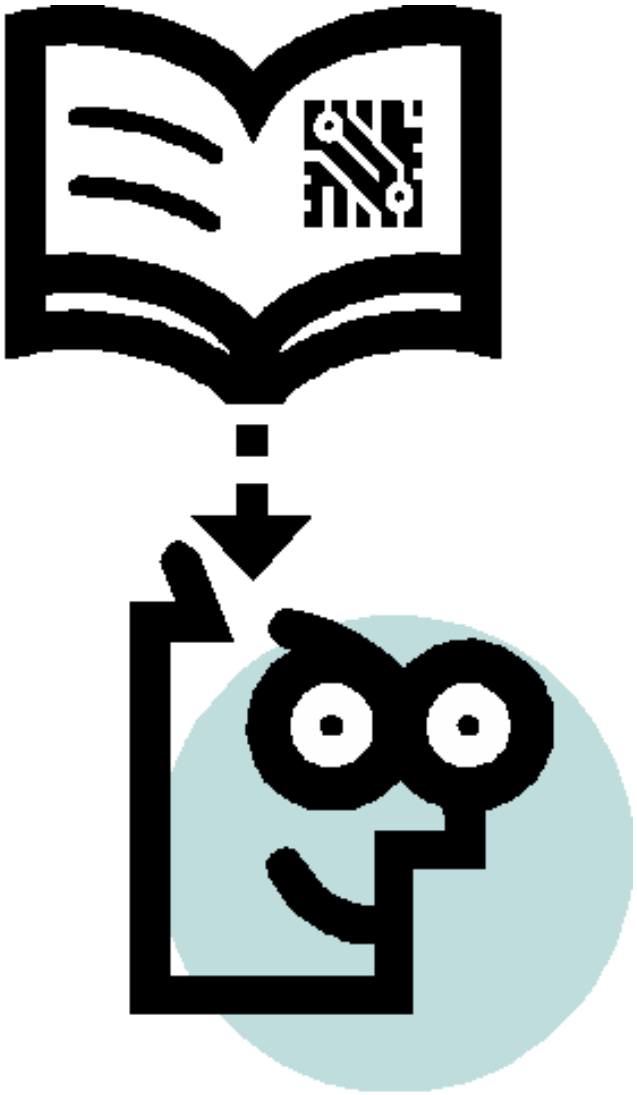
Preparing for Graduate School Examinations in Computer Science

Christopher Scaffidi



Version 2006.01.01

Shareware: $10 each

Preparing for Graduate School Examinations in Computer Science

Authored by
Christopher Scaffidi

Published by
Titanium Bits
5510 Forbes Ave
Pittsburgh, PA 15217
titanium.bits@gmail.com
http://www.geocities.com/titaniumbits/
http://www.titaniumbits.info/

Edition 2006.01.01
ISBN: 0-9727324-4-6

This booket is copyrighted shareware, not a public domain document. You may use the booklet at no charge for an evaluation period of 14 DAYS ONLY. To continue to use the booklet beyond the 14-day evaluation period, you must register it.

To register, send $10.00 to the author of this book by check, Amazon.com gift certificate, or other means. Once you have registered, the publisher grants you the right to use one copy of this booklet edition for YOUR OWN PERSONAL USE in perpetuity. You may send copies of this booklet to other people, but then they must pay to register those copies after the evaluation period. You may not make modifications of this booklet to share with other people.

Copyright ©2006 Christopher Scaffidi. All rights reserved. No part of this book, including interior design, cover design, and icons or images may be reproduced or transmitted in any form, by any means (electronic, photocopying, recording, or otherwise) without the prior written permission of the publisher.

Limit of liability/disclaimer of warranty: THIS BOOKLET IS PROVIDED STRICTLY ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, AND THE PUBLISHER AND AUTHOR HEREBY DISCLAIM ALL SUCH WARRANTIES, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, QUIET ENJOYMENT, OR NON-INFRINGEMENT.

The publisher and author have used their best effort to prepare this booklet. The publisher and author make no representations or warranties with respect to the accuracy or completeness of the contents of this booklet and disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranties exist beyond those expressed here. The accuracy and completeness of the advice, information and opinions provided in this booklet are not guaranteed or warranted to produce any particular result, including achieving any specific score on any specific exam. Neither the publisher nor author shall be liable for any loss or damages, including but not limited to incidental, consequential, or other damages.

Table of Contents

About this Booklet

The purpose of this booklet is to help you prepare for general introductory-level tests related to graduate school in computer science (CS). Subject test, preliminary examination —whatever you are facing, I hope this booklet helps you identify and correct gaps in your knowledge.

This booklet contains several sections. It begins with approximately 100 practice questions, followed by an answer key and a commentary on each problem. The questions cover hardware systems, software systems, algorithms & data structures, and the mathematics & theory of CS. The booklet closes with a list of supplementary resources that you should definitely check out.

Please note that these questions are not taken from any particular real test, nor are they intended to "give away" what will be on the actual exam. Who can predict the exact questions that will appear on your test? My hope is that this booklet will help you assess yourself in order to decide where to invest your time.

This booklet is shareware. You may use the booklet for free for 14 days. After that time, if you think the booklet is helpful and/or if you want to keep using the booklet, please send me $10. You can send an Amazon.com gift certificate to titanium.bits@gmail.com, or you can mail a check to me, Christopher Scaffidi, at 5510 Forbes Avenue, Pittsburgh, PA, 15217.

Acknowledgements

This booklet could not have been possible without the patience of my loving wife, Karen Needels. She never begrudges me the time I spend on software and CS; instead, she encourages me to seek out my dreams. I also appreciate the friendship of André Madeira, who has reviewed this booklet, done the lion's share of converting it to LaTeX, and provided valuable feedback. Thanks to Abhishek, Rodrigo, and Daniel for pointing out errors in previous versions; these have been corrected in version 2006.01.01. Of course, however, I am solely responsible for any mistakes or omissions, which are certainly possible. If you have any suggestions for how to improve this document, please email me so that I can update the text and include you in the next version's acknowledgements.

About the Author

As I write this, I am cruising on a train to my internship at Google. But in "real life," I am a second-year graduate student in the School of Computer Science at Carnegie Mellon University. My primary research interest is enabling software development by end users.

Though I had plenty of education prior to graduate school, little was in CS. For example, my Bachelor's from the University of Wisconsin (Madison) was in mathematics and physics. So I knew my CS subject test score would be an important part of my graduate school application, since it needed to demonstrate that I had a clue about CS.

Fortunately, I scored well on my examinations —97^{th} percentile on the subject test, and a perfect score on the quantitative —after hundreds of hours of study. At the time, I bemoaned the dearth of good practice problems for assessing my progress. Upon reflection, this ultimately inspired me to write this booklet.

I hope this booklet helps you get awesome scores so you can get into a great school like Carnegie Mellon University and eventually land a job at the company or university where you want to be.

Good luck!

Notation

This booklet relies on the following notations:

In arithmetic expressions...

mod	represents	modulo
$\log_b$	represents	logarithmic function base b
lg	represents	logarithmic function base 2 (unless otherwise stated)
ln	represents	natural logarithmic function

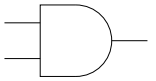
In Boolean expressions...

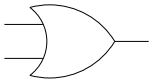
$*$	represents	conjunction (logical "and")
$+$	represents	disjunction (logical "or")
$\otimes$	represents	exclusive or (logical "xor")
$\rightarrow$	represents	logical implication
$\overline{\ }$	represents	negation (logical "not")

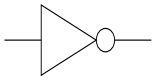
In set expressions...

$\cap$	represents	intersection
$\cup$	represents	union
$\overline{\ }$	represents	complement ("not")
$-$	represents	set difference

In circuit diagrams...


represents conjunction (logical "and")


represents disjunction (logical "or")

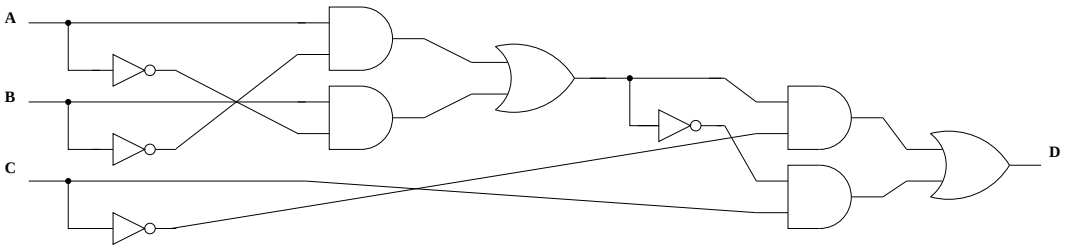

represents negation (logical "not")

In pseudocode...

new line	each line has one statement (i.e.: semicolons optional, as in Python)
braces { }	used to group statements that belong to the same basic block (as in C)
brackets [ ]	used to index into arrays (as in C)
`if, while, for`	control statements, though `for` uses semicolons (similar to C)
`Function`	defines a new procedure that can return a value (similar to JavaScript)
`return`	returns control to calling activation, perhaps with a value (as in C)
`int`	declares an integer variable or an integer formal parameter (as in C)
`Int8`	declares an 8-bit integer variable
`Int32`	declares a 32-bit integer variable
`Structure`	defines a structured data type (similar to structures in C)
`X.foobar()`	calls function `foobar` associated with object `X` (as in Java)

Hardware Systems — Questions

1. Consider the circuit below.



Which of the following statements *is/are true*?

I. The circuit above computes the (even) parity bit D of A, B, and C.

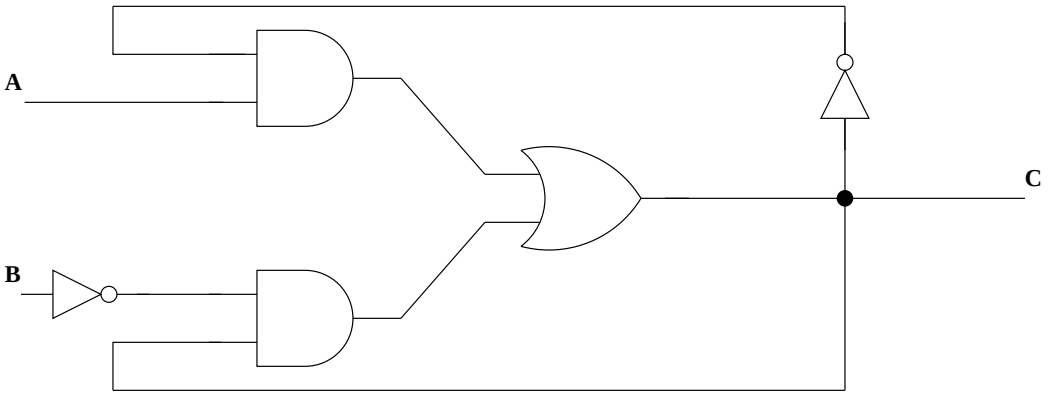
II. D will be true if and only if the following formula evaluates to true:

$$F(A, B, C) = \overline{A} * (\overline{B} * C + B * \overline{C}) + A * (B + \overline{C}) * (\overline{B} + C)$$

III. The reduced sum of products for this circuit has three or fewer items.

A. II only
B. I and II only
C. I and III only
D. I, II, and III
E. None of the above

2. Consider this circuit.



Suppose that the circuit is initialized by setting input $A = 0$ and $B = 0$, with $C = 0$. Assume that all inputs and outputs are clocked. Which of the following statements *is/are true*?

I. After the circuit is initialized, if A is set to 0 and B is set to 1, then output C becomes 1.
II. After the circuit is initialized, if A is set to 1 and B is set to 0, then output C becomes 1.
III. After the circuit is initialized, if A is set to 1 and B is set to 1, then output C becomes 1.

A. III only
B. I and II only
C. II and III only
D. I, II, and III
E. None of the above

3. Which of the following circuits is *not* combinational?

   A. Multiplexer
   B. Decoder
   C. Adder
   D. Barrel shifter
   E. Comparator

---

4. Consider the following Gray code sequence.

   ```
   000, 100, 101, 001, 011, X, 110, Y
   ```

   What are the missing entries, `X` and `Y`?

   A. `X=010, Y=010`
   B. `X=010, Y=100`
   C. `X=111, Y=010`
   D. `X=111, Y=100`
   E. `X=010, Y=111`

---

5. A manufacturer of spacecraft computer hardware needs to deal with the problem of occasional random bit flips in opcodes due to radiation. Which of the following statements *is/are true*?

   I. The minimum Hamming distance required to detect d errors is $d + 1$.
   II. The minimum Hamming distance required to correct d errors is $2 * d + 1$.
   III. The minimum Hamming distance required to prevent d errors is $3 * d + 1$.

   A. I only
   B. I and II only
   C. II and III only
   D. I, II, and III
   E. None of the above

6. Consider the following sequence of instructions intended for execution on a stack machine. Each arithmetic operation pops the second operand, then pops the first operand, operates on them, and then pushes the result back onto the stack.

```
push b
push x
add
pop c
push c
push y
add
push c
sub
pop z
```

Which of the following statements *is/are true*?

I. If push and pop instructions each require 5 bytes of storage, and arithmetic operations each require 1 byte of storage, then the instruction sequence as a whole requires a total of 40 bytes of storage.

II. At the end of execution, `z` contains the same value as `y`.

III. At the end of execution, the stack is empty.

A. I only

B. II only

C. II and III only

D. I, II, and III

E. None of the above

7. A certain architecture supports indirect, direct, and register addressing modes for use in identifying operands for arithmetic instructions. Which of the following *cannot* be achieved with a single instruction?

   A. Specifying a register number in the instruction such that the register contains the value of an operand that will be used by the operation.
   B. Specifying a register number in the instruction such that the register will serve as the destination for the operation's output.
   C. Specifying an operand value in the instruction such that the value will be used by the operation.
   D. Specifying a memory location in the instruction such that the memory location contains the value of an operand that will be used by the operation.
   E. Specifying a memory location in the instruction such that the value at that location specifies yet another memory location which in turn contains the value of an operand that will be used by the instruction.

---

8. The designers of a cache system need to reduce the number of cache misses that occur in a certain group of programs. Which of the following statements *is/are true*?

   I. If compulsory misses are most common, then the designers should consider increasing the cache line size to take better advantage of locality.
   II. If capacity misses are most common, then the designers should consider increasing the total cache size so it can contain more lines.
   III. If conflict misses are most common, then the designers should consider increasing the cache's associativity, in order to provide more flexibility when a collision occurs.

   A. III only
   B. I and II only
   C. II and III only
   D. I, II, and III
   E. None of the above

9. A system designer has put forth a design for a direct-mapped cache C. Suppose that reading a memory address A is anticipated to have an overall expected average latency $T(A, C)$ (including the average cost of cache misses on C). Which of the following statements *is/are true*?

   I. If C contains several words per cache line, then the index of the cache line for A is given by the rightmost bits of A.

   II. Suppose C is a unified cache and $C\prime$ is a comparable split cache with the same total capacity and cache line size as C. Then, generally, $T(A, C\prime) > T(A, C)$.

   III. Suppose $C\prime\prime$ is a two-way set associative cache with the same total capacity and cache line size as C. Then, generally, $T(A, C\prime\prime) < T(A, C)$.

   A. III only
   B. I and II only
   C. II and III only
   D. I, II, and III
   E. None of the above

---

10. A certain computer system design has a single CPU, a two-level cache, and supports memory-mapped I/O for output-only controllers (that is, controllers which only read from memory). Which of the following *is true*?

   A. This design is impossible, since memory-mapped I/O will prevent cache coherence.
   B. In two-level caches, the L2 cache is generally physically located on the same chip as the CPU.
   C. In two-level caches, the L1 cache is generally larger than the L2 cache.
   D. In two-level caches, the L2 cache generally has a lower latency than the L1 cache.
   E. In two-level caches, the L1 cache is generally built from SRAM.

11. The designers of a computer must select a cache system. They have two options.

    Design #1 uses a direct-mapped cache containing 2 words per cache line. It would have an instruction miss rate of 3% and a data miss rate of 8%.

    Design #2 uses a 2-way set associative cache containing 8 words per cache line. It would have an instruction miss rate of 1% and a data miss rate of 4%.

    For each design, there will be approximately 0.5 data references on average per instruction. The cache miss penalty in clock cycles is 8 + cache line size in words; for example, the penalty with 1-word cache lines would be 8 + 1 = 9 clock cycles.

    Let D1 = cycles wasted by Design #1 on cache miss penalties (per instruction)

    Let D2 = cycles wasted by Design #2 on cache miss penalties (per instruction)

    On average, how many clock cycles will be wasted by each on cache miss penalties?

    A. D1 = 0.45, D2 = 0.48
    B. D1 = 0.70, D2 = 0.40
    C. D1 = 0.70, D2 = 0.48
    D. D1 = 1.10, D2 = 0.40
    E. D1 = 1.10, D2 = 0.96

12. Consider the following two designs for a little cache.

    Design #1 is a direct-mapped cache of 8 1-word cache lines. The miss penalty is 8 clock cycles.

    Design #2 can store the same total number of items as Design #1, but it is a two-way associative cache of 1-word cache lines. Least-recently-used is utilized to determine which items should be removed from the cache. The miss penalty is 10 clock cycles.

    Suppose that the following eight memory references arrive.

    ```
    Memory References:  0, 3, 14, 11, 4, 11, 8, 0
    ```

    How much time will these designs spend on cache miss penalties, assuming that the caches start empty?

    A. Design #1 spends 56 cycles and Design #2 spends 60 cycles
    B. Design #1 spends 56 cycles and Design #2 spends 70 cycles
    C. Design #1 spends 48 cycles and Design #2 spends 70 cycles
    D. Design #1 spends 64 cycles and Design #2 spends 60 cycles
    E. Design #1 spends 64 cycles and Design #2 spends 80 cycles

---

13. A certain computer has a TLB cache, a one-level physically-addressed data cache, DRAM, and a disk backing store for virtual memory. The processor loads the instruction below and then begins to execute it.

    ```
    lw R3, 0(R4)
    ```

    This indicates that the computer should access the virtual address currently stored in register 4 and load that address's contents into register 3. Which of the following *is true* about what might happen while executing this instruction?

    A. If a TLB miss occurs, then a page fault definitely occurs as well.
    B. If a data cache miss occurs, then a page fault definitely occurs as well.
    C. No more than one data cache miss can occur.
    D. No more than one page fault can occur.
    E. If a page fault occurs, then a data cache miss definitely does not occur as well.

14. Twelve page requests occur in the following order: 9, 36, 3, 13, 9, 36, 25, 9, 36, 3, 13, and 25. Assume that physical memory initially starts empty and fully-associative paging is used. Which of the following statements *is/are true*?

    I. If the physical memory size is 3 pages, then most-recently-used (MRU) paging will result in the same number of faults as if the optimal algorithm was used.

    II. If first-in-first-out (FIFO) paging is used, and the physical memory size is raised from 3 pages to 4 pages, then Belady's anomaly will appear.

    III. If least-recently-used (LRU) paging is used, and the physical memory size is raised from 3 pages to 4 pages, then Belady's anomaly will appear.

    A. II only

    B. I and II only

    C. II and III only

    D. I, II, and III

    E. None of the above

15. Suppose that a direct-mapped cache has 2^9 cache lines, with 2^4 bytes per cache line. It caches items of a byte-addressable memory space of 2^{29} bytes.

    How many bits of space will be required for storing tags?
    (Do not include bits for validity or other flags; only consider the cost of tags themselves.)

    A. 2^8 bits
    B. 2^{11} bits
    C. 2^{13} bits
    D. 2^{25} bits
    E. 2^{32} bits

---

16. Suppose that a certain computer with paged virtual memory has 4 KB pages, a 32-bit byte-addressable virtual address space, and a 30-bit byte-addressable physical address space. The system manages an inverted page table, where each entry includes the page number plus 12 overhead bits (such as flags and identifiers).

    How big is the basic inverted page table, including page numbers and overhead bits?
    (Do not include any hash table for improving performance, nor any collision chains.)

    A. 2^{10} B
    B. 2^{20} B
    C. 2^{30} B
    D. 2^{32} B
    E. 2^{44} B

17. While designing a memory management subsystem, a manufacturer must decide whether to utilize a partitioning, segmentation, or demand paging strategy. Which of the following statements *is/are true*?

    I. Partitioning allocates one chunk of memory to each process, which helps to minimize internal fragmentation but can lead to substantial external fragmentation.

    II. Segmentation allocates one chunk of memory for each significant artifact within each process, which may burden developers or their tools with the task of associating artifacts with segments.

    III. Paging allocates many small chunks of virtual memory to each process, which helps to minimize external fragmentation but can lead to internal fragmentation.

    A. I only
    B. I and II only
    C. I and III only
    D. I, II, and III
    E. None of the above

---

18. A certain manufacturer has opted for a demand paging strategy in the memory management subsystem. Which of the following statements *is true*?

    A. Using smaller page sizes will generally lead to lower internal fragmentation.
    B. Using smaller page sizes will generally lead to a smaller page table.
    C. Using smaller page sizes will generally lead to lower external fragmentation.
    D. Compared to first-in-first-out paging, most-recently-used paging will generally lead to lower external fragmentation.
    E. Compared to first-in-first-out paging, most-recently-used paging will generally lead to lower job latency overall.

19. A certain hard drive rotates at 6000 rpm. It has 1 KB per sector and averages 128 sectors per track. Which of the following statements *is/are true*?

    I. The average latency of the drive is under 6 milliseconds.
    II. The burst data rate of the drive is over 10 MB/sec.
    III. The average capacity per track is over 1 MB.

    A. II only
    B. I and II only
    C. II and III only
    D. I, II, and III
    E. None of the above

---

20. A small disk drive with 80 cylinders (numbered 0 through 79) receives this batch of cylinder requests:

    ```
    4, 16, 3, 43, 60, 2, 79
    ```

    The head is initially at cylinder 40 and was headed in the direction of higher cylinder numbers (away from the spindle) when the batch of requests arrived. Assuming that seek time is proportional to seek distance (measured as number of cylinders traversed), which of these algorithms will result in the highest total seek time?

    A. FCFS
    B. SSTF
    C. SCAN
    D. LOOK
    E. C-LOOK

21. A super-simple RISC-like computer has three main computational elements (memory, registers, and ALU) which it uses to implement the four types of instructions shown below. This chart shows which main computational elements each instruction uses. For example, arithmetic instructions first require accessing memory, then accessing registers, then accessing the ALU, then accessing registers once more.

	Instruction Memory (Instruction Fetch)	Registers (Decode + Register Read)	ALU (Execute)	Data Memory (Data Access)	Registers (Write Back)
Branch	■	■	■		■
Arithmetic	■	■	■		■
Load	■	■	■	■	■
Store	■	■	■	■	

Suppose these computational elements require the following time to operate and settle (for either read or write operations):

```
Memory:     M ns
Registers:  R ns
ALU:        A ns
```

Which of the following statements *is/are true*?

I. If the machine has a single-cycle datapath with a constant cycle time, then the cycle cannot be faster than 2 * M + 2 * R + A ns.

II. If the machine has a multi-cycle datapath with a constant cycle time, then operations can be pipelined, but the cycle cannot be faster than max(M, R, A) ns.

III. Suppose that the machine has a single-cycle datapath, but the cycle length is varied as follows: Branch and arithmetic instructions will execute in M + 2 * R + A ns; load instructions will execute in 2 * M + 2 * R + A ns; store instructions will execute in 2 * M + R + A ns. In this case, the throughput will be better than in option I or II above.

A. I only

B. I and II only

C. II and III only

D. I, II, and III

E. None of the above

22. Suppose that the architecture outlined in the previous question, above, is implemented with a multi-cycle five-stage pipeline with no forwarding. The first stage corresponds to instruction fetch from memory, the second corresponds to data load from registers, the third corresponds to ALU operation, the fourth corresponds to memory access, and the fifth corresponds to register update. Each stage occupies one processor cycle.

    Suppose that a certain instruction in a program needs to read from a register that is written by the previous instruction in the program (read-after-write data hazard). How many cycles, if any, are wasted due to a pipeline stall?

    A. 0 cycles
    B. 1 cycle
    C. 2 cycles
    D. 3 cycles
    E. 4 cycles

---

23. Which of the following statements *is/are true*?

    I. Delayed control transfer involves starting the execution of the instruction after a branch or control instruction, regardless of whether the branch is taken.
    II. One way to implement branch prediction is to store the result of a branch condition in a branch target buffer to help guide instruction prefetching if the branch is encountered again later.
    III. If a multi-cycle, pipelined processor has N pipeline stages, then structural hazards can be avoided completely if at least N registers are available.

    A. I only
    B. I and II only
    C. II and III only
    D. I, II, and III
    E. None of the above

24. Which of the following statements *is/are true*?

    I. In-order retirement requires in-order starting of instructions, even if scoreboarding is used.

    II. In-order retirement means all instructions are committed in the order they appear in the program.

    III. In-order retirement facilitates the implementation of precise interrupts.

    A. I only

    B. II only

    C. II and III only

    D. I, II, and III

    E. None of the above

---

25. A processor manufacturer has found a nifty new way to parallelize a certain instruction so that the work of executing the instruction is split over a total of C copies of the datapath. Unfortunately, portions of the instruction's work are not parallelizable, so instructions cannot be sped up by a full factor of C.

    Suppose that this instruction usually takes 250 ns to execute when $C = 1$ but only 150 ns when $C = 3$. Approximately how much of the work is parallelizable?

    A. 40%

    B. 60%

    C. 70%

    D. 80%

    E. 90%

Software Systems — Questions

26. A certain text file is empty except for 2048 repetitions of a 16-byte sequence representing the following ASCII characters:

    "r", "a", "y", " ", "i", "s", " ", "y", "o", "u", "r", " ", "k", "e", "y", and ".".

    Note that " " is a blank/space character. Which of the following statements *is/are* true?

    I. Run-length encoding would provide significant compression of this file.
    II. Huffman encoding would provide significant compression of this file.
    III. Lempel-Ziv-Welch encoding would provide significant compression of this file.

    A. III only
    B. I and II only
    C. II and III only
    D. I, II, and III
    E. None of the above

---

27. While designing a file subsystem, an operating system manufacturer must decide how to organize files on disk. The subsystem must demonstrate very little internal fragmentation and must support fast random access for read operations; it is acceptable for file creation, updating, and deletion to run more slowly than reads.

    The manufacturer must select a file allocation strategy and a strategy for keeping track of free space. Which of the following options is most consistent with these requirements?

    A. Space should be allocated so each file's bytes are spread across a *linked list of large blocks*. The free space tracking strategy does not matter much and can be whatever is convenient.
    B. Space should be allocated so each file's bytes are stored *consecutively in a single block that is large enough* that the file's contents could never grow beyond the block's size. The free space tracking strategy does not matter much and can be whatever is convenient.
    C. Space should be allocated so each file's bytes are stored in small blocks scattered around the disk; keep an *index that maps from file locations to small blocks*. The free space tracking strategy does not matter much and can be whatever is convenient.
    D. The space allocation strategy does not matter much and can be whatever is convenient. The free space should be tracked with a *linked list of free blocks* on the disk.
    E. The space allocation strategy does not matter much and can be whatever is convenient. The free space should be tracked with a *bit vector*, where each bit indicates whether a certain block is free.

28. Suppose that on a certain system, every function call puts 10 bytes onto the stack plus another 4 bytes onto the stack for every integer formal parameter. Consider the following function definitions.

```
Function walk( int n ) {
    if ( n <= 1 ) then return
    run( n, n / 2 )
}

Function run( int x, int y ) {
    if ( x >= y ) then walk( y )
}
```

If n is a power of 2, then how much stack space must be available so that `walk(n)` can be called?

A. $14 * \log_2(n)$
B. $18 * \log_2(n)$
C. $32 * \log_2(n) - 18$
D. $32 * \log_2(n) + 14$
E. $32 * \log_2(n) + 18$

---

29. Consider the following function.

```
Function EXPL( int n ) {
    if ( n <= 2 ) then return 1
    return EXPL( n / 2 ) * lg( n )
}
```

What does `EXPL(n)` return, if n is a power of 2?

A. $\lg(n)$
B. $n * \lg(n)$
C. $n! * \lg(n)$
D. $(\lg(n))!$
E. $n! * (\lg(n))!$

30. Two 2D arrays of two-byte integers are stored in memory, but array A is stored in row-major order, while array B is stored in column-major order. Each array has 5 rows and 10 items in each row. Suppose array A is to be copied item by item into array B, beginning with item 0 in row 0 and ending with item 9 in row 4. A certain integer α occupies bytes 48 and 49 of array A and is then copied into array B following the copy strategy outlined above.

    Which of the following statements is true?

    A. α corresponds to item 4 of row 4, so it occupies bytes 48 and 49 of array B.

    B. α corresponds to item 4 of row 2, so it occupies bytes 44 and 45 of array B.

    C. α corresponds to item 4 of row 4, so it occupies bytes 88 and 89 of array B.

    D. α corresponds to item 2 of row 4, so it occupies bytes 28 and 29 of array B.

    E. α corresponds to item 2 of row 2, so it occupies bytes 44 and 45 of array B.

31. A certain language allows the declaration of data structures containing a variety of elements such as arrays, sub-structures and primitives. These primitives include single-byte "Int8" integers (which can be used to contain ASCII characters) and four-byte "Int32" integers (which can be used as pointers to memory locations). However, all multi-byte elements must be aligned on four-byte boundaries; if necessary, extra "padding" bytes are silently inserted by compilers before multi-byte elements to ensure this alignment.

    Consider the following data structure definitions. The `LittleString` structure represents a string of characters and contains an array of 50 single-byte integers and a single-byte integer that tells how many items in the array are actually in use. The second structure represents a node in a binary tree and contains one embedded `LittleString` structure as well as two pointers to child nodes.

```
Structure LittleString {
    Int8 length
    Int8 contents[50]
}

Structure TreeNode {
    LittleString key
    Int32 leftChild
    Int32 rightChild
}
```

    Suppose that a certain full tree of TreeNodes contains 20 leaf nodes. How many bytes of memory does the tree require, in total?

    A. 2301

    B. 2340

    C. 2360

    D. 2480

    E. 2496

32. Consider the following code.

```
int a = 1
int b = 1

Function foo( ) {
    print( a )
}

Function bar( int x, int y ) {
    int a = 0
    foo()
    x = 0
    b = b - y
    y = y + 1
}

Function foobar() {
    int r = 1
    bar( r, b )
    print( b )
    print( r )
}
```

Suppose that `foobar()` is called from the outermost scope. Which of the following statements *is/are true*?

I. If dynamic scoping is used, then the first digit printed is 1, regardless of how parameters are passed (by value, reference, or value-result).

II. If lexical scoping and pass-by-value-result are used, then the second digit printed is 1.

III. If lexical scoping and pass-by-value are used, then the third digit printed is 1.

A. III only

B. I and II only

C. II and III only

D. I, II, and III

E. None of the above

33. Which of the following statements *is/are true*?

    I. Stop-and-copy garbage collection may be preferable to reference counting because stop-and-copy can easily deal with circular references.

    II. Mark-and-sweep may be preferable to stop-and-copy because mark-and-sweep is more amenable to implementation as an incremental garbage collection algorithm.

    III. Stop-and-copy garbage collection may be preferable to mark-and-sweep because stop-and-copy affords the opportunity to periodically defragment the heap.

    A. I only
    B. III only
    C. II and III only
    D. I, II, and III
    E. None of the above

---

34. An operating system manufacturer must decide whether to support access control lists (ACLs) or capabilities. Which of the following statements *is/are true*?

    I. The ACL strategy may be preferable to the capabilities strategy because once a capability has been granted, it is difficult to revoke, and this may lead to additional code complexity.

    II. The capabilities strategy may be preferable to the ACL strategy because ACLs must be checked on every operation, and this may lead to poorer performance.

    III. The capabilities strategy may be preferable to the ACL strategy because the semantics of an ACL are typically hard to match to an API, and this may lead to poor code maintainability.

    A. II only
    B. I and II only
    C. I and III only
    D. I, II, and III
    E. None of the above

35. Which of the following statements *is not true*?

    A. Deadlock can never occur if all resources can be shared by competing processes.

    B. Deadlock can never occur if resources must be requested in the same order by processes.

    C. Deadlock can never occur if processes must request in advance all their resources that they will require (that is, they cannot hold some resources while waiting for the rest to become available).

    D. The Banker's algorithm for avoiding deadlock requires knowing resource requirements in advance.

    E. If the resource allocation graph depicts a cycle, then deadlock has certainly occured.

---

36. While designing a kernel, an operating system manufacturer must decide whether to support kernel-level or user-level threading. Which of the following statements *is/are true*?

    I. Kernel-level threading may be preferable to user-level threading because storing information about user-level threads in the process control block would create a security risk.

    II. User-level threading may be preferable to kernel-level threading because in user-level threading, if one thread blocks on I/O, then the process can continue.

    III. User-level threading may be preferable to kernel-level threading because in user-level threading, less expensive overhead is required when one thread blocks and another begins to run instead.

    A. III only

    B. I and II only

    C. II and III only

    D. I, II, and III

    E. None of the above

37. Consider the following code, which prints 100 integers. It uses semaphore `smx` to assure thread-safety, since the queue `Q` is not thread-safe. This queue provides two functions, `add()` and `remove()`, for first-in-first-out loading and unloading of items from a list. The program will crash if `remove()` is called while the queue is empty.

```
Function loadem() {
    for ( int x = 0; x < 100; x = x + 1 ) {
        Q.add( x )
        V( smx )
    }
}
Function printem() {
    while ( true ) {
        P( smx )
        int value = Q.remove()
        print( value )
    }
}
```

One thread executes `loadem()` at an arbitrary time, while another thread executes `printem()` at another arbitrary time. Which of the following statements *is/are true*?

I. If the `V` operation implements non-blocking atomic increment, and the `P` operation implements blocking atomic decrement, then the code above will work properly.

II. The code above may crash if multiple threads simultaneously execute `printem()`, even if only a single thread executes `loadem()`.

III. The code segment above is functionally equivalent to the following, where `mx` is a mutex...

```
Function loadem() {
    for ( int x = 0; x < 100; x = x + 1 ) {
        lock( mx )
        Q.add( x )
        unlock( mx )
    }
}
Function printem() {
    while ( true ) {
        lock( mx )
        int value = Q.remove()
        print( value )
        unlock( mx )
    }
}
```

A. II only

B. I and II only

C. I and III only

D. I, II, and III

E. None of the above

38. The single-CPU computer on a certain blimp has an operating system with a purely priority-driven preemptive scheduler that runs three processes.

    Process A is a low-priority process that runs once per hour; when it runs, it acquires an exclusive write lock on a log file, writes the current position and speed to the log file, closes the file, and then exits. Process B is a medium-priority video-streaming process that usually isn't running; when it runs, it does so continuously for an arbitrary length of time. Process C is a high-priority navigation process that runs once per minute; when it does, it acquires an exclusive write lock on the log file, issues some commands to the rudder, closes the file, and then exits.

    Which of the following statements *is/are true*?

    I. Starvation may occur for one or more processes for an arbitrary length of time.
    II. Priority inversion may occur for an arbitrary length of time.
    III. Deadlock may occur.

    A. II only
    B. I and II only
    C. I and III only
    D. I, II, and III
    E. None of the above

39. While designing a preemptive job scheduling subsystem, an operating system manufacturer must select a scheduling strategy. Which of the following requirements is a reason why round-robin scheduling might be preferred?

    A. The subsystem must minimize the number of context switches.
    B. The subsystem must achieve optimal throughput of jobs.
    C. The subsystem must guarantee that starvation will not occur.
    D. The subsystem must achieve the optimal response ratio.
    E. The subsystem must guarantee that if job J1 arrives before job J2, then J1 finishes before J2.

---

40. Multiple processes need to run on a non-preemptive system. Process A is currently active but needs to acquire an exclusive lock on a certain resource R in order to continue. Unfortunately, Process B currently has an exclusive lock on R.

    Process A *can spin-wait* until R becomes available, or A *can yield* so another waiting process may run. Which of the following is *not* a reasonable heuristic for maximizing the amount of work that can be completed by this machine?

    A. A should never yield to another process.
    B. If this is a uniprocessor machine, then A should yield to another process.
    C. If this is a multi-processor machine, but B is inactive, then A should yield to another process.
    D. If this is a multi-processor machine, and B is active, and the time to complete a context switch does not exceed the time that B will retain the lock, then A should yield to another process.
    E. If this is a multi-processor machine, and B is active, but the time to complete a context switch exceeds the time that B will retain the lock, then A should spin-wait.

---

41. Which of the following statements *is/are true*?

    I. Response time may be reduced if processes migrate from overloaded servers to other servers.
    II. Job response time may be improved if two processes that communicate with one another can migrate to a common host.
    III. Availability may be improved if processes can migrate from unstable hosts to more stable hosts.

    A. III only
    B. I and II only
    C. II and III only
    D. I, II, and III
    E. None of the above

42. Which of the following is *not* a possible way to reduce the overall latency of operations on distributed systems?

    A. Replicating computational units
    B. Prefetching data
    C. Multithreading
    D. Locking shareable resources with mutexes
    E. Utilizing non-blocking writes

---

43. Suppose that process P has been running for several days when a new process Q starts up and begins contending with P for resources. Which of the following *is true*?

    A. In a wait-die system, if P needs a resource held by Q, then P waits.
    B. In a wait-die system, if Q needs a resource held by P, then Q waits.
    C. In a wound-wait system, if P needs a resource held by Q, then Q yields and waits.
    D. In a wound-wait system, if Q needs a resource held by P, then P yields and waits.
    E. In a wound-wait system, if Q needs a resource held by P, then Q dies.

---

44. A firm with 200 web developers has a policy that their programmers must "work at workstations but develop on servers." What the management means by this is that developers must sit at workstations; however, they must store their source code on a server's file system, and the binaries they produce are written directly to the server's file system. Moreover, the server (rather than the workstations) runs the binaries that are produced.

    Which of the following statements *is/are true*?

    I. This policy could be problematic in the absence of a source code control system, since programmers might make incompatible changes to portions of the code base, resulting in a body of source code that cannot be compiled.
    II. This policy could be problematic in terms of overhead, since it may result in network traffic spikes every time that any programmer compiles any source code.
    III. This policy could be problematic in terms of tool costs, since the compiler, assembler, and linker tools must all be capable of opening a TCP/IP socket.

    A. III only
    B. I and II only
    C. II and III only
    D. I, II, and III
    E. None of the above

45. Which of the following requirements *is* a good reason why an organization might choose to use 100Base-T Fast Ethernet?

    A. Throughput rates in excess of 100 Mbps are required.
    B. Cable segments are expected to exceed 500 meters.
    C. A bus protocol is undesirable due to the possibility of packet sniffing.
    D. The protocol must be supported by major vendors and compatible with many products.
    E. Utilization in excess of 95% must be tolerated with minimal effect on overall latency.

---

46. Suppose a user turns on a computer, starts a browser, types `http://www.gre.com`, and hits `ENTER`. Which of the following protocols would probably *not* be used at any point to serve this request?

    A. HTTP
    B. TCP
    C. UDP
    D. IP
    E. SMTP

---

47. Suppose a program transmits 64 short UDP packets to a certain host. Which of the following statements *is not true*, assuming that none of the packets get dropped?

    A. If the packets are transmitted over a circuit-switched network, then the packets are guaranteed to arrive in order.
    B. If the packets are transmitted over a local area packet-switched network with fixed routing tables, then the packets are guaranteed to arrive in order.
    C. If the packets are transmitted over a wide area packet-switched network with virtual circuit routing, and all the packets are transmitted in the same session, then the packets are guaranteed to arrive in order.
    D. If the packets are transmitted over a wide area packet-switched network with dynamic routing, then the packets are guaranteed to arrive in order.
    E. Regardless of the routers or underlying network, there is a non-zero probability that the packets will arrive in order.

48. Which of the following will definitely result in datagram fragmentation?

    A. Transmitting datagrams longer than the physical layer's maximum transmission unit
    B. Transmitting datagrams with more than 4 bytes in the IP header
    C. Transmitting over a message-switched network
    D. Transmitting over a circuit-switched network
    E. Transmitting over a packet-switched network

---

49. A system administrator uses SNMP to transmit shut down commands to eight machines on a network... three servers, three clients, and two printers. For all practical purposes, when the commands arrive at their respective destinations, the machines shut down in random order (with a uniform probability distribution over all possible orderings).

    What is the probability that all three clients will shut down before a single server shuts down?

    A. $\frac{1}{40}$
    B. $\frac{1}{20}$
    C. $\frac{1}{10}$
    D. $\frac{1}{6}$
    E. $\frac{1}{3}$

---

50. A certain system has a reliability of 30 days between failures and a maintainability of 25% probability of being repaired per day (including the same day in which failure occurs). What is the approximate availability?

    A. 50%
    B. 80%
    C. 90%
    D. 99%
    E. 99.9%

Algorithms and Data Structures — Questions

51. Consider the following code, where `mod` is the modulo operator.

```
Function FMR ( int lowValue, int highValue ) {
    if ( lowValue == 0 ) then return highValue
    int modValue = ( highValue mod lowValue )
    return FMR ( modValue, lowValue )
}
```

Which of the following statements *is/are true*?

I. The `FMR` function requires the precondition that `lowValue` $\leq$ `highValue` in order to achieve the postcondition of returning the greatest common denominator of `lowValue` and `highValue`.

II. Due to recursion, the stack may grow to contain $O(\lg(\max(\texttt{lowValue}, \texttt{highValue})))$ activation records during evaluation of $\texttt{FMR}(\texttt{lowValue}, \texttt{highValue})$.

III. The `FMR` function requires $O(\lg(\max(\texttt{lowValue}, \texttt{highValue})))$ time to run.

A. I only

B. II only

C. II and III only

D. I, II, and III

E. None of the above

52. The intended purpose of this code is to precompute all the primes less than N. When it is finished executing, for $r \in [2, N)$, `bits`$[r]$ is supposed to equal 1 if and only if N is composite. Assume that the bits array is initialized to all zeroes.

```
for ( int x = 2; x < N; x = x + 1 ) {
    int y = x
    while ( y < N ) {
        bits[y] = 1
        y = y + x
    }
}
```

Which of the following statements *is/are true*?

I. The algorithm requires $O(N)$ temporary space.

II. The algorithm demonstrates an asymptotic algorithmic complexity of $O(N \ln(N))$.

III. After the initialization shown above, it is possible to determine in $O(1)$ time whether a natural number $n < N$ is prime.

A. I only

B. I and II only

C. II and III only

D. I, II, and III

E. None of the above

53. Suppose it is necessary to locate the zero of the function $f(x) = x^2 - 5$ using three algorithms: Newton's method, the method of false position, and the bisection method. Newton's method is initialized with $a = 1$, while the other two are initialized with bounds $a = 1$ and $b = 3$.

    Which of the following statements *is/are true*?

    I. In its first iteration, Newton's method estimates that $x = 2$; in its second iteration, Newton's method then estimates that $x = 2.5$.

    II. In its first iteration, the method of false position estimates that $x = 2$; in its second iteration, the method of false position estimates that $x = 2.5$.

    III. In its first iteration, the bisection method estimates that $x = 2$; in its second iteration, the bisection method estimates that $x = 2.5$.

    A. III only

    B. I and III only

    C. II and III only

    D. I, II, and III

    E. None of the above

54. Consider the following code, which operates on two 2-dimensional arrays to generate a third 2-dimensional array.

```
for ( int i = 0; i < N; i = i + 1 ) {
    for ( int j = 0; j < N; j = j + 1 ) {
        C[i][j] = 0
        for ( int k = 0; k < N; k = k + 1 ) {
            C[i][j] = C[i][j] + A[k][j] * B[i][k]
        }
    }
}
```

Which of the following statements *is/are true*?

I. The code above would produce the same result even if the first and second lines were swapped.

II. The code above could be used to multiply matrices A and B and store the result in matrix C.

III. The code above could be replaced by implementing a different algorithm that achieves the same result in $O(n^{2.81})$ worst-case asymptotic algorithmic complexity.

A. I only

B. II only

C. II and III only

D. I, II, and III

E. None of the above

---

55. Which of the following algorithms for operating on matrices does *not* demonstrate $O(n^3)$ worst-case asymptotic algorithmic complexity? (Here, n is the maximum number of rows or columns in the input matrix or matrices.)

A. Gaussian elimination

B. Multiplying two matrices

C. Simplex method

D. Matrix inversion

E. Calculating a matrix determinant

56. Suppose that two repositories exist for storing and searching through a certain type of record. The first repository uses a linked list; on average, it requires 10 ms to search 1024 records, or 10240 ms to search 1048576 records. The second repository uses a sorted array and a binary search; on average, it requires 400 ms to search to search 1024 records, or 800 ms to search 1048576 records.

    For what number of records do the two versions require approximately equal time, on average?

    A. 2048 records
    B. 4096 records
    C. 16384 records
    D. 65536 records
    E. 262144 records

---

57. Which of the following sorting algorithms has the lowest best-case asymptotic algorithmic complexity?

    A. Selection sort
    B. Insertion sort
    C. Quick sort
    D. Heap sort
    E. Merge sort

---

58. Which of the following sorting algorithms has the highest worst-case asymptotic algorithmic complexity?

    A. Radix sort
    B. Counting sort
    C. Randomized quick sort
    D. Shell sort
    E. Merge sort

59. A software vendor needs to choose two sorting algorithm implementations I1 and I2. I1 will be used in situations where item exchanges cost nothing but item comparisons remain expensive. Conversely, I2 will be used in situations where item comparisons cost nothing but item exchanges remain expensive.

    Suppose the vendor can only use the insertion, selection, or bubble sorts for these implementations, and suppose the vendor only cares about average-case asymptotic algorithmic complexity. Which algorithm should the vendor use for each implementation?

    A. Insertion sort for I1 and selection sort for I2
    B. Insertion sort for I1 and bubble sort for I2
    C. Selection sort for I1 and insertion sort for I2
    D. Selection sort for I1 and selection sort for I2
    E. Bubble sort for I1 and insertion sort for I2

---

60. Why might quick sort be preferred over insertion sort and merge sort?

    A. The worst-case asymptotic algorithmic complexity of quick sort is superior to that of insertion sort and merge sort.
    B. In situations where little temporary space is available, merge sort cannot be used, and in such cases, the average-case asymptotic algorithmic complexity of quick sort is superior to that of insertion sort.
    C. When the inputs are nearly sorted, the asymptotic algorithmic complexity of quick sort is superior to that of insertion sort and merge sort.
    D. When random access is very slow, as with sorting records on a long tape, the average run time of quick sort is superior to that of insertion and merge sort.
    E. Quick sort is a stable sort, whereas insertion sort and merge sort are not.

61. Which of the options shown below is an optimal Huffman code for the following distribution?

```
a occurs 10% of the time
b occurs 14% of the time
c occurs 16% of the time
d occurs 18% of the time
e occurs 42% of the time
```

A. `a =  00, b =  01, c = 110, d = 111, e =  10`

B. `a =   0, b = 100, c = 101, d = 110, e = 111`

C. `a = 000, b = 001, c = 010, d = 011, e =   1`

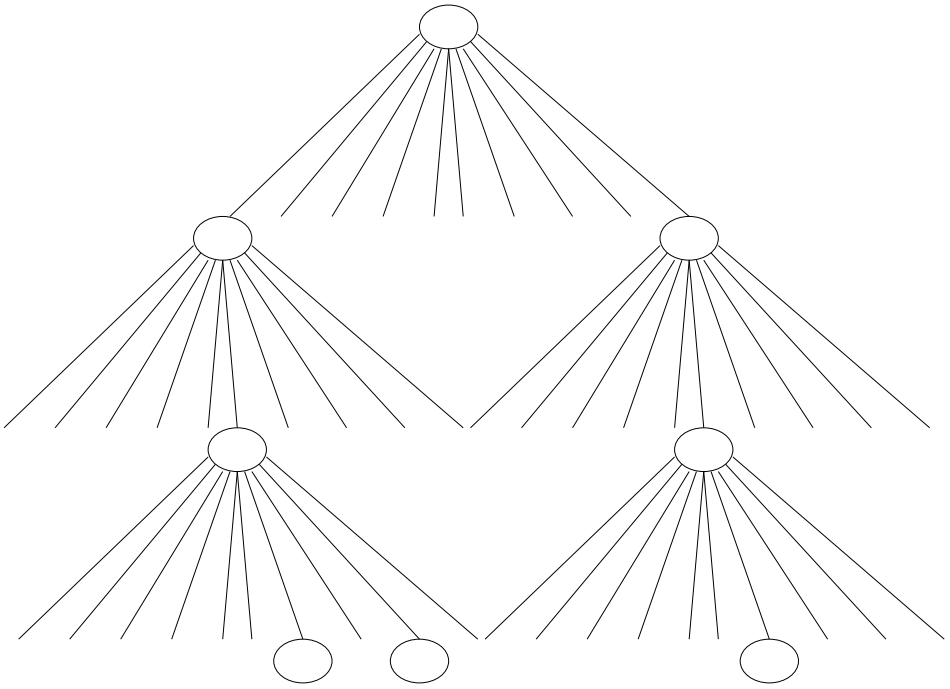
D. `a = 000, b = 001, c =   1, d = 011, e = 010`

E. `a =  00, b =  10, c = 010, d = 011, e =  11`

62. A programmer has "invented" a data structure that he calls the RD10 tree to replace the venerable hash table for storing 3-*digit integers* written in base-10.

    The root of the tree has 10 children, each of which has 10 children, each of which has 10 children (for a total of 1000 leaves). Each leaf corresponds to one possible 3-digit key, since the first digit of the key can be used to select an edge from the root, and so forth for the other two digits. Each leaf carries a "flag" attribute which is set to 0 to start with, but then is set to 1 if the corresponding key is stored in the RD10 tree.

    For example, if the values 056, 058, and 956 had been stored in the RD10 tree, it would look like the following. (To conserve paper, only sub-trees with a flagged descendant are showed here, even though *all* sub-trees and all 1000 leaves would actually be present in an RD10 tree.)



    Which of the following statements *is not true*?

    A. If N different keys are flagged in an RD10 tree, the same tree structure will result regardless of the order in which the N keys are flagged.

    B. If RD10 trees are generally sparsely populated with keys, as in the example above, then the programmer should consider using a left-child right-sibling representation to conserve memory.

    C. If insertions are much more common than searches, then in order to improve performance, the programmer should consider flagging interior nodes to indicate if they have any flagged descendants.

    D. The time it takes to flag a new key does not depend on the number of keys already flagged.

    E. The time it takes to find a key in the RD10 tree does not depend on the number of flagged keys.

63. Given a binary search tree T, what is the path from a node x to its successor y, assuming that both x and y exist in T?

A. y is the right child of x

B. y is the rightmost descendant of x

C. if x has a right child, then y is the right child of x; otherwise, if x is a left child, then y is the parent of x; otherwise, y is the left child

D. if x has a right child, then y is the leftmost descendant of x's right child; otherwise, if x is a left child, then y is the parent of x; otherwise, y is the parent of x's first ancestor z such that z is a left child

E. if x has a right child, then y is the right child of x; otherwise, y is the parent of x

---

64. Suppose that six keys are inserted into an unbalanced binary search tree in the following order: 4, 6, 3, 8, 2, and 5. Which of the following statements *is/are true*?

I. Finding a key in the resulting tree requires examining 1, 2, or 3 nodes.

II. The resulting tree has equal numbers of interior and leaf nodes.

III. The key 7 can now be inserted without adding another level to the tree.

A. I and II only

B. I and III only

C. II and III only

D. I, II, and III

E. None of the above

65. Which of these statements *is not true* about an AVL tree T containing n nodes?

    A. Rotations may be required during key insertion to keep T balanced.

    B. The height of T cannot exceed $1.5 * \log_2(n)$.

    C. The number of interior nodes in T cannot exceed the number of leaves in T.

    D. If each node in T is augmented with an integer showing the size of that node's sub-tree, then T can be used to perform order statistic searches in $O(\lg n)$ asymptotic algorithmic complexity.

    E. It is possible to insert nodes into T in $O(\lg n)$ asymptotic algorithmic complexity.

---

66. Which of these statements *is not true* about a Red-Black tree T containing n nodes?

    A. Rotations may be required during key deletion to maintain the Red-Black tree properties.

    B. If P is the distance from the root to the most distant leaf, and p is the distance from the root to the nearest leaf, then $P < 1.5 * p$.

    C. The number of nil leaves in T equals 1 plus the number of interior nodes in T.

    D. If each node in T is augmented with an integer showing the size of that node's sub-tree, then T can be used to find the rank of a key in $O(\lg n)$ asymptotic algorithmic complexity.

    E. It is possible to delete nodes from T in $O(\lg n)$ asymptotic algorithmic complexity.

---

67. Which of these statements *is not true* about a B-Tree with height h and n nodes, assuming that each node takes exactly 1 disk operation to read?

    A. Rotations may be required during insertion to keep T balanced.

    B. h cannot exceed $\log_t((n+1)/2)$, where t is the minimum node degree.

    C. If each node in T is augmented with an integer showing the size of that node's sub-tree, then n additional nodes can be inserted into T in a total of $O(n * h)$ CPU operations.

    D. Finding a node in T cannot require more than $O(h)$ disk operations (in other words, $O(h)$ time, if only disk reads and writes are counted).

    E. Finding a node in T cannot require more than $O(h)$ CPU operations (in other words, $O(h)$ time, if only instruction executions on the CPU are counted).

68. A hash table H with $m = 8$ slots uses open addressing. Six keys are inserted into H in the following order: 14, 23, 0, 6, 3, and 11. Let $h(k, i)$ be the slot to be probed on the i^{th} attempt (where i is numbered from 0) for key k.

    Which of the following statements *is/are true*?

    I. After all values are inserted, H will contain [0, 6, empty, 3, empty, 11, 14, 23] if linear probing is used with $h(k, i) = (k + i) \mod m$.

    II. After all values are inserted, H will contain [0, 6, empty, 3, 11, empty, 14, 23] if quadratic probing is used with $h(k, i) = (k + i + 2 * i^2) \mod m$.

    III. After all values are inserted, H will contain [0, empty, empty, 3, 11, 6, 14, 23] if double hashed probing is used with $h(k, i) = (k + i * d(k)) \mod m$, and $d(k)$ is the sum of the decimal digits in k.

    A. I and II only

    B. I and III only

    C. II and III only

    D. I, II, and III

    E. None of the above

69. Given a graph G with vertex set V and edge set E, which of the following statements *is/are true*?

    I. If G is directed and acyclic, the asymptotic algorithmic complexity of topological sort on G is $O(|V| + |E|)$, assuming edges are represented in adjacency lists rather than an adjacency matrix.

    II. If G is directed or undirected, the asymptotic algorithmic complexity of breadth-first search on G is $O(|V| + |E|)$, assuming edges are represented in adjacency lists rather than an adjacency matrix.

    III. If G is undirected, during a depth-first search of G, exactly four types of edges might be identified: tree edges (members of the spanning forest), back edges (linking descendants to ancestors), forward edges (non-tree edges linking ancestors to descendants), and cross edges (all other remaining links).

    A. I only

    B. III only

    C. I and II only

    D. I, II, and III

    E. None of the above

---

70. Given an undirected graph G with vertex set V and edge set E, which of the following statements about single-source shortest path algorithms *is not true*?

    A. Dijkstra's algorithm is different than Prim's algorithm in that Dijkstra's algorithm can use a priority queue, but Prim's algorithm cannot use a priority queue.

    B. Dijkstra's algorithm runs on G in asymptotic algorithmic complexity $O(|E| + |V| \lg |V|)$, assuming that a Fibonacci heap is used for the priority queue of waiting vertices and edges are represented in adjacency lists (rather than an adjacency matrix).

    C. Dijkstra's algorithm possesses lower average asymptotic algorithmic complexity than the Bellman-Ford algorithm, especially if G is so dense that $|E| \approx |V|^2$.

    D. The Bellman-Ford algorithm can handle edges with negative weights, whereas Dijkstra's algorithm may fail if G possesses edges with negative weights.

    E. Dijkstra's algorithm resembles breadth-first search in that each algorithm grows a tree of "finished" nodes (none of which are modified again later by the algorithm).

71. Given an undirected graph G with vertex set V and edge set E, which of the following statements *is not true*?

    A. Kruskal's algorithm is different than Prim's algorithm in that Kruskal's algorithm relies heavily on disjoint set operations, but Prim's algorithm does not involve keeping track of many disjoint sets.

    B. Kruskal's algorithm runs on G in asymptotic algorithmic complexity $O(|E|\alpha(|V|))$, assuming that the Union-Find data structure is used (and α is the inverse Ackermann's function).

    C. If G is so sparse that $|E|$ is essentially a constant, Kruskal's algorithm is preferable to Prim's algorithm in terms of asymptotic algorithmic complexity, even if Fibonacci heaps are used in implementing Prim's algorithm.

    D. Kruskal's algorithm can handle edges with negative weights, whereas Prim's algorithm may fail if G possesses edges with negative weights.

    E. Prim's algorithm resembles depth-first search in that when each finishes running on a strongly connected graph, the result is a single tree.

---

72. Given an undirected graph G with vertex set V and edge set E, which of the following statements *is not true*?

    A. Johnson's algorithm differs from the Floyd-Warshall algorithm in that Johnson's algorithm uses adjacency lists whereas the Floyd-Warshall algorithm uses an adjacency matrix.

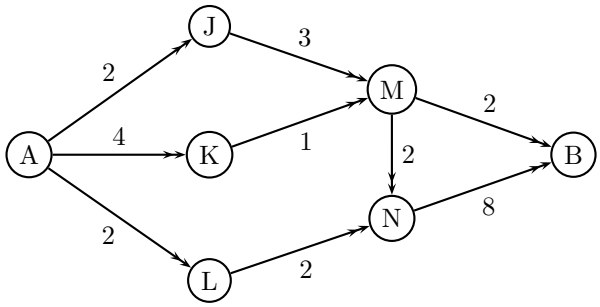
    B. The Floyd-Warshall algorithm runs on G in asymptotic algorithmic complexity $O(|V|^3)$.

    C. The Floyd-Warshall algorithm is preferable to Johnson's algorithm in terms of asymptotic algorithmic complexity, particularly if G is so sparse that $|E|$ is a negligibly small constant.

    D. The Floyd-Warshall algorithm and Johnson's algorithm can handle graphs with negative weights, as long as no negative weight cycles are present.

    E. Johnson's algorithm utilizes a single-source shortest-path algorithm such as Dijkstra's.

73. What is the maximum possible flow from point A to point B through the directed graph below? (Note that the number near each edge indicates that edge's capacity.)



A. 2

B. 5

C. 8

D. 10

E. 11

---

74. Which of the following algorithms *is* a dynamic programming algorithm?

A. Dijkstra's single-source shortest-paths algorithm

B. Kruskal's minimum spanning tree algorithm

C. Prim's minimum spanning tree algorithm

D. Depth-first search

E. The Floyd-Warshall all-pairs shortest-paths algorithm

75. Which of the following statements *is not true*?

    A. Greedy algorithms do not depend on examining sub-problems in order to make a locally optimal choice.

    B. Dynamic programming algorithms are bottom-up rather than top-down.

    C. Divide-and-conquer constitutes a top-down approach to solving problems.

    D. Dynamic programming generally runs much slower than an equivalent memoization approach if every sub-problem must be solved.

    E. Memoization requires memory for storing solutions to sub-problems.

---

76. Which of the following statements *is not true*?

    A. Dynamic programming and greedy algorithms depend on optimal substructure: Problem P contains two sub-problems P_1 and P_2 that are structurally similar to P but smaller in size.

    B. Dynamic programming depends on overlapping sub-problems: Problem P contains sub-problems P_1 and P_2, and P_1 and P_2 in turn share numerous sub-problems Q_1 through Q_n.

    C. Greedy algorithms depend on existence of an equivalent matroid representation: Problem P can be solved with a greedy algorithm if and only if P can be represented as finding a maximum-weight independent subset within a weighted matroid.

    D. Dynamic programming depends on sub-problem utility: Examination of all sub-problems of P is worthwhile in the process of determining the solution to P.

    E. Greedy algorithms depend on the greedy choice property: The optimal solution to problem P can be determined without first obtaining optimal solutions to sub-problems of P.

Mathematics and Theory — Questions

77. Which of the following expressions evaluates to the largest number?

    A. The prefix expression `+ * - 2 3 5 7`
    B. The infix expression `2 + 3 * 5 - 7`
    C. The infix expression `(2 + 3) * (5 - 7)`
    D. The postfix expression `2 3 + 5 7 - *`
    E. The postfix expression `2 3 + 5 * 7 -`

---

78. A function returns a twos-complement two-byte integer. On success, the return value exceeds 1024. On error, the return value is in the range -1 through -10 to represent ten different error conditions.

    Which of the following hexadecimals could *not* be returned by this function?

    A. 0401
    B. 3840
    C. FFED
    D. FFF6
    E. FFFD

79. Which of the following pairs of twos-complement two-byte integers, written in hexadecimal, would result in overflow if added?

    A. 4650, 2340
    B. FFED, FFFF
    C. 787A, E3E0
    D. 1010, 0101
    E. 878A, E0E3

---

80. A binary single-precision floating point number contains the sequence of bits `10001111100000000001000000000000`. Information is stored in the following left-to-right order: sign bit, exponent (bias -127), and mantissa (with an implied unit bit).

    Which of the following representations in decimal is equivalent?

    A. $2^{31} * (1 + 2^{-12})$
    B. $-1 * 2^{31} * (1 + 2^{-12})$
    C. $-1 * 2^{-65} * (1 + 2^{-10})$
    D. $-1 * 2^{-96} * (1 + 2^{-11})$
    E. $-1 * 2^{-112} * (1 + 2^{-12})$

81. Which of the following statements *is/are true*?

I. Floating point addition is always associative.

II. Shifting a twos-complement integer right by one bit, and filling from the left with 0, is always equivalent to dividing by 2.

III. An integer's ones-complement representation is never identical to its twos-complement representation.

A. I only
B. I and II only
C. II and III only
D. I, II, and III
E. None of the above

---

82. Which of the following statements *is/are true*?

I. $a \otimes b$ always equals $(\overline{a} * b) + (a * \overline{b})$

II. $a \otimes (b \otimes c)$ always equals $(a \otimes b) \otimes c$

III. $(a + \overline{b}) * (b + c)$ always equals $a + c$

A. III only
B. I and II only
C. II and III only
D. I, II, and III
E. None of the above

83. Which of the following statements *is/are true*?

I. $(a \rightarrow b)$ always equals $\overline{a} + b$
II. $(\overline{a} + b) + (a * \overline{b})$ is a tautology
III. $(a \rightarrow b) * (a * \overline{b})$ is satisfiable

A. III only
B. I and II only
C. II and III only
D. I, II, and III
E. None of the above

---

84. Which of the following statements *is/are true*?

I. $A \cup (B - C)$ always equals $(A \cup B) - (A \cup C)$
II. $A \cap (B - C)$ always equals $(A \cap B) - (A \cap C)$
III. $A - (B \cap C)$ always equals $(A - B) \cup (A - C)$

A. I only
B. I and II only
C. II and III only
D. I, II, and III
E. None of the above

85. Consider a binary function $g : M \times M \rightarrow \{\texttt{true, false}\}$, where M is a non-empty subset of the natural numbers that contains an even number of distinct elements.

    Which of the following statements *might* be *true* about g?

    A. g is symmetric and antisymmetric
    B. g defines a total order and an equivalence relation with at least two equivalence classes
    C. g defines a total order but not a partial order
    D. g is reflexive and antisymmetric but not a surjection
    E. g is an injection

---

86. Consider two natural-valued functions $f : N \rightarrow N$ and $g : N \rightarrow N$.

    Which of the following statements *cannot* be *true*?

    A. $f \in O(g)$ and $g \in O(f)$
    B. $f \in \Theta(g)$ and $g \in \Theta(f)$
    C. $f \in \Omega(g)$ and $g \in \Omega(f)$
    D. $f \in O(g)$ but $g \notin \Omega(f)$
    E. $f \notin O(g), \Theta(g)$, or $\Omega(g)$, and $g \notin O(f), \Theta(f)$, or $\Omega(f)$

87. For all natural numbers $n \in N$, define f and g as follows:

$$f(n) = \sum_{k=1}^{n} \frac{1}{k} \qquad\qquad g(n) = \sum_{k=0}^{n-1} e^k$$

Which of the following statements *is not true*?

A. $f \in O(\log_2(n))$
B. $f \in \Omega(\ln(n))$
C. $f \in O(\ln(g))$
D. $f \in \Omega(\ln(g))$
E. $f \in \Theta(\ln(\ln(g)))$

---

88. Suppose c is an integer greater than 1, and some function f is defined such that $f(0) = f(1) = 1$. Then which of the following statements must *not* be *true*?

A. If $f(n) = f(n/2) + c$, then $f(n) \in O(\lg n)$
B. If $f(n) = f(n-2) + c$, then $f(n) \in O(n)$
C. If $f(n) = f(n-2) + n^c$, then $f(n) \in O(n^c)$
D. If $f(n) = c * f(n-2)$, then $f(n) \in O(c^n)$
E. If $f(n) = c * f(n/2)$, then $f(n) \in O(n^{\lg c})$

89. A certain function $g : N \times N \rightarrow N$ is defined as follows for x and $y \geq 0$:

$$\begin{cases} g(0,y) = y+1 & \text{for } y \geq 0 \\ g(x,0) = g(x-1,1) & \text{for } x > 0 \\ g(x,y) = g(x-1,g(x,y-1)) & \text{for } x \text{ and } y > 0 \end{cases}$$

Which of the following statements *is/are true*?

I. $g(1,n) = n-3$
II. $g(2,n) = 2n-3$
III. $g(3,n) = 2^{n+3}-3$

A. III only
B. I and III only
C. II and III only
D. I, II, and III
E. None of the above

90. Consider the non-singular matrices shown below.

$$A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 2 \\ -1 & 1 & 1 \end{bmatrix} \qquad A^{-1} = \frac{1}{6}\begin{bmatrix} 2 & 1 & -4 \\ 2 & -2 & 2 \\ 0 & 3 & 0 \end{bmatrix} \qquad x * A = \begin{bmatrix} 2 & 4 & 2 \\ 0 & 0 & 4 \\ -2 & 2 & 2 \end{bmatrix}$$

$$X * A = \begin{bmatrix} 0 & 0 & 2 \\ 1 & 2 & 1 \\ -1 & 1 & 1 \end{bmatrix} \qquad Y * A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 2 \\ -2 & 2 & 2 \end{bmatrix} \qquad Z * A = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 2 \\ -1 & 1 & 5 \end{bmatrix}$$

Which of these statements *is not true*?

A. $det(A) = 1/det(A^{-1})$
B. $det(x * A) = x^3 * det(A)$
C. $det(X * A) = det(A)$
D. $det(Y * A) = 2 * det(A)$
E. $det(Z * A) = det(A)$

---

91. Given two square, non-singular matrices A and B, which of the following statements is *not* always *true*?

A. $(A * B)^{-1} = B^{-1} * A^{-1}$
B. $(A * B)^T = B^T * A^T$
C. $det(A * B) = det(A) * det(B)$
D. $(A + B)^{-1} = A^{-1} + B^{-1}$
E. $(A + B)^T = A^T + B^T$

92. Given a graph G with vertex set V and edge set E, which of the following *is not true if and only if* G is a tree?

    A. For all vertices u and $v \in V$, there exists a unique simple path from u to v.

    B. G is connected, but if any edge $e \in E$ is removed, the resulting graph is not connected.

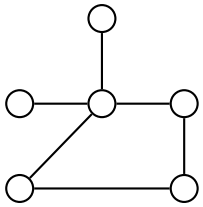
    C. G is acyclic, but if any edge $e \notin E$ is added, the resulting graph is not acyclic.

    D. $|E| = |V| - 1$ and either G is acyclic or G is connected.

    E. G is strongly connected, and every vertex is a member of exactly one clique.

---

93. Consider the following undirected graph G. Which of the statements below *is true*?



    A. G has a clique of size 4

    B. G is strongly connected

    C. G has a Hamiltonian circuit

    D. G has an Eulerian circuit

    E. G is complete

94. A certain programming language supports declaration and assignment of variables (including integers, and character string variables), blocking console I/O to/from variables, arithmetic operations, and *forward* branches. The language does not permit branches or jumps from address A to address B if $B \leq A$, nor does it support functions.

    Which of the following statements *is true*?

    A. The language is Turing-complete.

    B. For any program P, there exists a constant p such that for all console inputs into P, P is guaranteed to halt within p seconds of starting.

    C. For any program P, there exists a constant p such that for all console inputs into P, P is guaranteed to execute no more than p instructions before halting.

    D. This language could be used to input a number q and then print all the Fibonacci numbers less than q.

    E. This language could be used to input a regular expression e and a string of characters s, and then determine if e could generate s.

---

95. Which of the following languages over the alphabet $A = \{0, 1\}$ *is* regular?

    A. $\{w \in A^* : w$ contains equal numbers of 1's and 0's$\}$

    B. $\{w \in A^* : w$ contains a prime number of 1's$\}$

    C. $\{w \in A^* : \exists u \in A^*$ such that $w = uu\}$

    D. $\{w \in A^* : w$ does not contain any 1's in even positions, where the leftmost is position 1$\}$

    E. $\{w \in A^* : w$ contains a 1 in every position that is a power of 2$\}$

96. Consider these three grammars.

Grammar G1:	Grammar G2:	Grammar G3:
E → E + T | T	E → E + T | T	E → T R | R
T → T * v | v	T → v R	R → + T R | ε
	R → * v R | ε	T → T * v | v

Which of the following statements *is not true*?

A. If w can be generated by G1, then it can be generated by G2.
B. If w can be generated by G2, then it can be generated by G3.
C. If w can be generated by G3, then it can be generated by G1.
D. If w can be generated by G2, then it can be generated by G1.
E. If w can be generated by G1, then it can be generated by G3.

---

97. Consider the following grammars:

Grammar G1:	Grammar G2:
S → 0 T | ε	S → T S
T → 1 S	S → ε
	T → X Y
	X → 0
	Y → 1

Which of the following statements *is not true*?

A. Grammar G1 can generate any string that G2 can, and G1 can do so in fewer steps than G2 can.
B. Grammar G2 is ambiguous.
C. Grammar G1 corresponds to a regular language.
D. Grammar G1 corresponds to a language that can be recognized with an LR parser.
E. Grammar G2 is in Chomsky normal form.

98. Which of the following problems *is* decidable?

    A. Determining whether two nondeterministic finite automata accept the same language.
    B. Determining whether two context-free grammars represent exactly the same language.
    C. Determining whether a predicate expression is satisfiable.
    D. Determining whether a Turing machine will halt for any input.
    E. Determining whether a Turing machine decides an $\mathcal{NP}$-hard language.

---

99. Consider a language L that is recognized by a machine M. Which of the following statements might *not* be *true*?

    A. If M is a deterministic finite automaton, then L can be represented by a regular expression.
    B. If M is a non-deterministic finite automaton, then L can be represented by a context-free grammar.
    C. If M is a deterministic pushdown automaton, then L can be represented by a context-free grammar.
    D. If M is a non-deterministic pushdown automaton, then L is recursively enumerable.
    E. If M is a Turing machine, then L is recursive.

---

100. Which of the following statements *is not true*?

    A. The class of regular languages is closed under union, intersection, concatenation, and Kleene star.
    B. The class of context-free languages is closed under union, intersection, concatenation, and Kleene star.
    C. The class of languages Turing-decidable in polynomial time ($\mathcal{P}$) is closed under union, intersection, concatenation, and Kleene star.
    D. The class of recursive languages is closed under union, intersection, concatenation, and Kleene star.
    E. The class of recursively enumerable languages is closed under union, intersection, concatenation, and Kleene star.

101. Which of the following problems is not $\mathcal{NP}$-complete?

A. Determining whether a Boolean proposition is satisfiable
B. Determining the clique of maximum size within a graph
C. Determining whether a directed graph contains a Hamiltonian circuit
D. Determining whether an undirected graph contains an Eulerian circuit
E. Determining the shortest round-trip route that visits all vertices in a graph

---

102. Suppose that $\mathcal{NP}$ is a strict superset of $\mathcal{P}$. Consider a language A that is reducible in polynomial time to a language B. Which of the following statements *must* be *true*?

A. If A is in $\mathcal{NP}$, then B is in $\mathcal{NP}$.
B. If A is in $\mathcal{P}$, then B is in $\mathcal{P}$.
C. If A is in $\mathcal{P}$, then B is $\mathcal{NP}$-hard.
D. If B is in $\mathcal{NP}$-complete, then A is in $\mathcal{P}$.
E. If B is in $\mathcal{P}$, then A is in $\mathcal{P}$.

Answer Key

The following sections represent a commentary on these answers, rather than an attempt to formally prove the answers correct. Hopefully, this commentary will deliver insights for how to approach each problem, or at least provide keywords for use in finding information in the Resources listed at the end of this booklet.

1. B
2. C
3. D
4. C
5. B
6. C
7. C
8. D
9. A
10. E
11. C
12. B
13. C
14. B
15. C
16. B
17. D
18. A
19. B
20. A
21. B
22. D
23. B
24. C
25. B
26. C
27. C
28. D
29. D
30. B
31. E
32. A
33. D
34. B
35. E
36. A
37. A
38. B
39. C
40. A
41. D
42. D
43. A
44. B
45. D
46. E
47. D
48. A
49. B
50. C
51. C
52. B
53. A
54. D
55. C
56. D
57. B
58. C
59. A
60. B
61. C
62. C
63. D
64. A
65. C
66. B
67. A
68. E
69. C
70. A
71. D
72. C
73. B
74. E
75. D
76. C
77. E
78. C
79. E
80. D
81. E
82. B
83. B
84. C
85. A
86. D
87. D
88. C
89. A
90. C
91. D
92. E
93. B
94. C
95. D
96. C
97. B
98. A
99. E
100. B
101. D
102. E

Hardware Systems — Comments

1. **Parity circuit**

   Examination of a truth table is often the most educational test for determining whether functions are equal. It may seem tempting to rapidly guess the answer, but building a truth table and/or Karnaugh map can significantly speed up the process as well as provide an easy tool for double-checking the answer.

A	B	C	Circuit	Parity	F
0	0	0	0	0	0
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	0	0	0
1	0	0	1	1	1
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	1	1	1

   Clearly, the circuit, parity, and F all serve to produce the same outputs.

   Algebraically, the equivalence of the circuit and the parity function can be seen by recognizing that the parity of A, B, and C can be written as $XOR(XOR(A,B),C)$, and $XOR(A,B)$ can be written as $A * \overline{B} + \overline{A} * B$. The equivalence of these and formula F can be seen by applying De Morgan's laws:

   $\overline{A+B} = \overline{A} * \overline{B} \qquad \overline{A*B} = \overline{A} + \overline{B}$

   The reduced sum of products for a truth table is most easily ascertained by drawing the Karnaugh map for the truth table and then circling adjacent groups of 1's.

	$\boldsymbol{B}C = 00$	$\boldsymbol{B}C = 01$	$\boldsymbol{B}C = 11$	$\boldsymbol{B}C = 10$
$\boldsymbol{A} = 0$	0	1	0	1
$\boldsymbol{A} = 1$	1	0	1	0

   Since no adjacent entries contain 1 entries (diagonal does not count!), the reduced sum of products contains four terms, $A * \overline{B} * \overline{C} + \overline{A} * B * \overline{C} + \overline{A} * \overline{B} * C + A * B * C$.

---

2. **Flip-flop circuit**

   This circuit is essentially a J-K flip-flop, with a third input for the clock, which is not depicted in the diagram. Thus, the circuit demonstrates the truth table below:

A	B	C_{old}	C_{new}
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

   This truth table can be used to verify that I is false, but II and III are true.

3. **Combinational?**

   The barrel shifter can rotate a string of bits by an arbitrary number of positions in a single step. As such, the "next" state of this circuit is a function of the "current" state, so it is sequential. All of the other circuits listed are combinational, meaning that their outputs are not influenced by the previous state of the circuit.

---

4. **Gray code**

   When you list 2^n distinct n-bit binary numbers in a Gray code, adjacent numbers differ by exactly one bit. Thus, `X` must be either `111` or `010`; `Y` must be either `010` or `100`. Now, in order to list all 2^3 numbers, no number can appear twice. Note that `100` already appeared in the list, so `Y` cannot be `100`. Thus, `Y = 010`. Then, by a similar reasoning, `X` must be `111`. Hence, choice `C` is correct.

   If the 2^n numbers are represented with a Gray code (rather than with the usual binary representation), then hardware for incrementing a number always flips exactly one bit. For example, if 3 (base 10) is represented as `001` and 4 (base 10) is `011` (as in this problem), then incrementing `001` requires only one bit flip to generate `011`.

---

5. **Hamming distance**

   The Hamming distance between two bit strings is the number of bits that would have to flip to make the strings identical.

   Consider a single bit flip. This could turn 001 into 011, for example. If both 001 and 011 are valid strings, then there is no way to detect that an error occurred when 011 is observed. So strings must differ by at least 2 bits in order for 1 bit errors to be detectable. In general, to detect d errors requires a minimum Hamming distance of d + 1.

   However, just detecting a bit flip does not necessarily make it feasible to figure out what the original value was. If both 001 and 100 are valid bit strings, but 101 is observed, how is the observer to know whether an 001 or a 100 was intended? In order to correct a 1 bit error, the minimum necessary Hamming distance is 3 bits. So, for example, if only 111 and 000 are valid, but an 001 arrives, then the receiver can correct this to 000 under the assumption that only one bit flipped. Correcting d bit flips requires a minimum Hamming distance of 2 * d + 1.

   There is no way to prevent bit flips by adjusting the Hamming distance.

6. **Stack machine**

   There are 7 pushes/pops for a cost of 35 bytes, plus 3 arithmetic instructions, for a total of 38 bytes.

After Instruction	Stack Contains	New Variables
push b	b	
push x	b, x	
add	b + x	
pop c		c = b + x;
push c	b + x	c = b + x;
push y	b + x, y	c = b + x;
add	b + x + y	c = b + x;
push c	b + x + y; b + x	c = b + x;
sub	y	c = b + x;
pop z		c = b + x; z = y;

---

7. **Addressing modes**

   Choice A is "register" addressing, which is supported by this architecture. Choice B is also typically covered when manufacturers speak of "register" addressing, which is supported by this architecture. Choice C is "immediate" (or "literal") addressing, which is not supported by this architecture. Choice D is "direct" (or "absolute") addressing, which is supported by this architecture. Choice E is "indirect" (or "memory indirect") addressing, which is supported by this architecture.

---

8. **Types of cache misses**

   Increasing the cache line size brings in more from memory when a miss occurs. If accessing a certain byte suggests that nearby bytes are likely to be accessed soon (locality), then increasing the cache line essentially prefetches those other bytes. This, in turn, forestalls a later cache miss on those other bytes.

   If misses occur because the cache is too small, then the designers should increase the size!

   Conflict misses occur when multiple memory locations are repeatedly accessed but map to the same cache location. Consequently, when they are accessed, they keep kicking one another out of the cache. Increasing the associativity implies that each chunk of the cache is effectively doubled so that more than one memory item can rest in the same cache chunk.

9. **Types of caches**

   In a direct-mapped cache with many words per cache line, the offset is given by the rightmost bits and the cache line's index is given by some of the middle bits. (Since a program's code and data each occupy small areas of memory, using the leftmost bits for the cache index would cause numerous cache collisions.)

   A split cache is essentially two small caches, one for instructions and one for data. The benefit is that the processor can read instructions while the data cache is busy, which generally improves overall latency on average rather than worsens it.

   Increasing the associativity of a cache often improves the hit rate of the cache, since there generally will be fewer conflict misses due to collisions. This will thus reduce the latency of reads overall.

---

10. **L1 and L2 caches**

    Choice A *is not true*. Cache coherence problems only arise when multiple CPUs attempt to modify memory in the same system; in this system, only one CPU exists, and the controllers only listen for writes to portions of the memory space, since they are not allowed to modify memory. Consequently, no cache coherence problem results. Besides, even if a problem did exist, solutions do exist, as discussed in [Stallings].

    In general, the Level-1 (L1) cache is a small, fast, expensive cache located either on or fairly close to the CPU chip. The Level-2 (L2) cache is somewhat larger, slower, and less expensive (per bit) than the L1 cache; the L2 is generally located off the CPU chip. L1 and L2 both usually use SRAM. Thus, choices B, C, and D are false but E *is true*.

---

11. **Miss penalties and cycles per instruction**

    For these designs, let p equal the cache miss penalty, in clock cycles, and let m_i and m_d indicate the instruction and data miss rates, respectively. Then the total time spent on penalties, for an average instruction, is $p * (1 * m_i + 0.5 * m_d)$, since there are about 0.5 data references per instruction. Consequently, the total penalty for D1 and D2, are 0.70 and 0.48, respectively.

12. **Least-recently used and miss penalties**

Consider Design #1 first. The memory references will map to the following respective cache lines: `0, 3, 6, 3, 4, 3, 0`, and `0`. After each memory reference, the cache will look as follows (where * indicates empty), for a total of 7 misses (56 cycles):

After reference to 0...	`0 * * * * * * *`	(this caused a miss)
After reference to 3...	`0 * * 3 * * * *`	(this caused a miss)
After reference to 14...	`0 * * 3 * * 14 *`	(this caused a miss)
After reference to 11...	`0 * * 11 * * 14 *`	(this caused a miss)
After reference to 4...	`0 * * 11 4 * 14 *`	(this caused a miss)
After reference to 11...	`0 * * 11 4 * 14 *`	
After reference to 8...	`8 * * 11 4 * 14 *`	(this caused a miss)
After reference to 0...	`0 * * 11 4 * 14 *`	(this caused a miss)

Now, Design #1 can store 8 items, and Design #2 can store the same number of items. However, in Design #2, these are grouped in pairs, so the conversion from memory addresses to cache locations is modulo 4 rather than modulo 8. Hence, the memory references map to the following cache lines: `0, 3, 2, 3, 0, 3, 0`, and `0`. So after each memory reference, the cache will look as follows (where * indicates empty, and hyphens join cache lines in the same associative block), for a total of 7 misses (70 cycles):

After reference to 0...	`0--* *--* *--* *--*`	(this caused a miss)
After reference to 3...	`0--* *--* *--* 3--*`	(this caused a miss)
After reference to 14...	`0--* *--* 14--* 3--*`	(this caused a miss)
After reference to 11...	`0--* *--* 14--* 3-11`	(this caused a miss)
After reference to 4...	`0--4 *--* 14--* 3-11`	(this caused a miss)
After reference to 11...	`0--4 *--* 14--* 3-11`	
After reference to 8...	`8--4 *--* 14--* 3-11`	(this caused a miss)
After reference to 0...	`8--0 *--* 14--* 3-11`	(this caused a miss)

13. **Cache misses and page faults**

For the details of executing this operation, see [Patterson]. The basic process starts with converting the virtual address to a physical address; if possible, this relies on the TLB cache's stored entry that maps virtual address to physical address, but if this fails, then the page table must be consulted in physical or virtual memory. Once the physical address is available, the processor can attempt to retrieve the actual value from the data cache, but if this fails, then the value must be read from physical memory.

Choice A *is not true*, since a TLB miss can occur without a subsequent page fault. The TLB is typically much smaller than physical memory. It is quite possible that a page's virtual-to-physical mapping will get forced out of the TLB without getting forced out of physical memory.

Choice B *is not true*, since a data cache miss can occur without a subsequent page fault. Like the TLB, the data cache is much smaller than the physical memory. It is quite possible that page content will get forced out of the cache without getting forced out of physical memory.

Choice C *is true*, since virtual-to-physical address translation relies on testing the TLB cache rather than the data cache. Hence, the translation cannot generate a data cache fault. Although the ensuing load from the physical address can still generate a data cache miss, this still only totals one data cache miss.

Choice D *is not true*, since multiple page faults can occur. For example, one page fault could occur if a TLB miss occurs and the page table for the process is no longer in physical memory. Even after the physical address is available, a second page fault could occur if that physical address's page is no longer in physical memory.

Choice E *is not true*. Because retrieving data from disk takes milliseconds, whereas retrieving data from cache takes microseconds, computers will never generate page faults until after unsuccessfully testing the cache. Hence, if a page fault occurs, then a cache miss must have preceded it.

14. **Paging algorithms**

The optimal algorithm is to remove the page that will be needed farthest in the future. If this algorithm is used with 3 pages, then physical memory contains the following values:

After 9	(which caused a fault):	9	
After 36	(which caused a fault):	36, 9	
After 3	(which caused a fault):	3, 36, and 9	[from farthest to soonest used]
After 13	(which caused a fault):	13, 36, and 9	
After 9	(which hit):	13, 9, and 36	
After 36	(which hit):	13, 36, and 9	
After 25	(which caused a fault):	25, 36, and 9	
After 9	(which hit):	9, 25, and 36	
After 36	(which hit):	9, 36, and 25	
After 3	(which caused a fault):	3, 36, and 25	
After 13	(which caused a fault):	13, 3, and 25	
After 25	(which hit):	13, 3, and 25	→ Total of **7 faults**

Suppose that MRU is used with 3 pages. Then physical memory contains the following values:

After 9	(which caused a fault):	9	
After 36	(which caused a fault):	36, and 9	
After 3	(which caused a fault):	3, 36, and 9	[from most to least recently used]
After 13	(which caused a fault):	13, 36, and 9	
After 9	(which hit):	9, 13, and 36	
After 36	(which hit):	36, 9, and 13	
After 25	(which caused a fault):	25, 9, and 13	
After 9	(which hit):	9, 25, and 13	
After 36	(which caused a fault):	36, 25, and 13	
After 3	(which caused a fault):	3, 25, and 13	
After 13	(which hit):	13, 3, and 25	
After 25	(which hit):	25, 13, and 3	→ Total of **7 faults**

Clearly, MRU and the optimal algorithm happen to generate the same number of faults in this case (though this is not a universal truth for memory reference strings in general). Thus, I *is true*.

LRU never demonstrates Belady's anomaly, which is when increasing the number of frames also increases the number of faults. In contrast, FIFO sometimes results in Belady's anomaly, so it is necessary to check. Suppose that FIFO is used with 3 pages. Then physical memory contains the following values:

After 9	(which caused a fault):	9	
After 36	(which caused a fault):	9, 36	
After 3	(which caused a fault):	9, 36, and 3	[from first to last loaded]
After 13	(which caused a fault):	36, 3, and 13	
After 9	(which caused a fault):	3, 13, and 9	
After 36	(which caused a fault):	13, 9, and 36	
After 25	(which caused a fault):	9, 36, and 25	
After 9	(which hit):	9, 36, and 25	
After 36	(which hit):	9, 36, and 25	
After 3	(which caused a fault):	36, 25, and 3	
After 13	(which caused a fault):	25, 3, and 13	
After 25	(which hit):	25, 3, and 13	→ Total of **9 faults**

Suppose that FIFO is used with 4 pages. Then physical memory contains the following values:

After 9	(which caused a fault):	9	
After 36	(which caused a fault):	9, 36	
After 3	(which caused a fault):	9, 36, and 3	[from first to last loaded]
After 13	(which caused a fault):	9, 36, 3, and 13	
After 9	(which hit):	9, 36, 3, and 13	
After 36	(which hit):	9, 36, 3, and 13	
After 25	(which caused a fault):	36, 3, 13, and 25	
After 9	(which caused a fault):	3, 13, 25, and 9	
After 36	(which caused a fault):	13, 25, 9, and 36	
After 3	(which caused a fault):	25, 9, 36, and 3	
After 13	(which caused a fault):	9, 36, 3, and 13	
After 25	(which caused a fault):	36, 3, 13, and 25	→ Total of **10 faults**

Thus, FIFO does demonstrate Belady's anomaly in this case. As noted earlier, LRU never does. So II *is true* and III *is false*.

---

15. **Size of cache**

Of the 29 bits of the address space, 9 indicate the cache line index, and 4 indicate the offset within the cache line. That leaves $16 = 2^4$ bits for the tag. There are 2^9 tag entries, for a total of $2^{9+4} = 2^{13}$ bits.

---

16. **Size of page table**

The page table typically contains one entry for each virtual page. That entry lists the physical frame's number along with some overhead bits (such as flags for indicating whether the entry is valid and whether it needs to be copied to disk eventually). But with one entry for each virtual page, the table becomes extraordinarily big for large address spaces.

Several strategies are available for coping with this, such as only including entries for a subset of the virtual address space, or paging the page table, or inverting the page table. The essence of an inverted page table is that there is one row for each frame, and that row contains the number of the virtual page stored at that frame.

While this last strategy does an effective job of reducing the size of the table, it requires a hash table to keep performance acceptable (for details, see [Silberschatz] and [Tanenbaum]).

Each page is 4 KB = 2^{12} B. Since physical memory is 2^{30} B, it includes $2^{30-12} = 2^{18}$ frames. Likewise, virtual memory is 2^{32} B, and it includes $2^{32-12} = 2^{20}$ pages. Thus, the virtual page number must be 20 bits long, and the overhead costs an additional 12 bits, for a total of $32 = 2^5$ bits per entry. Since there are 2^{18} inverted page table entries, this amounts to $2^{5+18} = 2^{23}$ bits, or 2^{20} B.

17. **Partitioning, segmentation, and paging**

Partitioning does allocate one contiguous piece of memory to each process. Consequently, as processes start, execute, and exit, the free space of memory can get broken into many useless little chunks; this is called "external fragmentation." Fortunately, on the assumption that the operating system allocates each process approximately the amount of space actually used by the process, there is little waste within each partition ("internal fragmentation").

At the other extreme, paging views physical memory as a long series of small equally-sized pieces, and the operating system allocates pieces of physical memory to processes on an as-needed basis. An extra level of indirection is needed so that the memory space seen by the process appears contiguous, even if it actually resides in non-contiguous pieces of physical memory. Thus, any free piece of physical memory can be assigned when a request occurs, so there is no external fragmentation. On the other hand, processes can never receive less than a page of memory, so there can be non-negligible internal fragmentation.

Segmentation represents a compromise between these two extremes. Each artifact of a program corresponds to a specific piece of contiguous memory. For example, the stack might be in one piece, while an array might be in another, and the text/code for a function might reside in another. For the most part, this keeps the chunks allocated to a fairly small size, so there is usually a use for every chunk of free memory; this helps to reduce external fragmentation somewhat. Also, the artifacts generally fill most or all of their respective memory chunks, which helps to moderate internal fragmentation. However, the compiler (or programmer) must now keep track of the relationship between artifacts and segments.

---

18. **Page sizes**

On average, each process uses only half of its last page. Consequently, minimizing page sizes helps minimize internal fragmentation, as indicated by choice A. However, using smaller pages means that the page table must contain more entries, so choice B *is false*. Choices C and D *are false* because external fragmentation never exists when paging is used, so adjusting page size or page replacement strategy has no effect on external fragmentation. It might be possible to construct a situation where MRU paging yields better job throughput than FIFO, but it is certainly not universally true. MRU throws out the page that was most recently used, which is not consistent with the goal of taking advantage of temporal locality.

---

19. **Disk drive speed**

Average latency owes to the time for a point on the disk to rotate under the head, or about half of the rotation time. The latency in milliseconds is given by the formula 30000 / (speed in rpms), which works out to 5 ms in this case.

The capacity of each track is given by the average number of sectors per track times the size per sector. (Note that on modern disk drives, there are more sectors per track near the outer edge of the disk, and fewer sectors per track near the spindle.) This works out to 128 KB in this case.

The burst data rate is the maximum amount of data that can stream off the disk per unit time. It is roughly equal to the average track capacity times rotations per second, which works out to 12.5 MB per second in this case.

---

20. **Disk scheduling algorithms**

FCFS ("first come first served") provides service to seeks in order:

Seek From	Seek To	Cost	Running Total
40	4	36	36
4	16	12	48
16	3	13	61
3	43	40	101
43	60	17	118
60	2	58	176
2	79	77	253

SSTF ("shortest seek time first") handles seeks by constantly going to whatever is nearest.

Seek From	Seek To	Cost	Running Total
40	43	3	3
43	60	17	20
60	79	19	39
79	16	63	102
16	4	12	114
4	3	1	115
3	2	1	116

SCAN reads from end to end, then back again, over and over. The head will continue reading toward higher cylinder numbers, traversing 39 cylinders before reaching cylinder 79, and then it will start toward 0, reading another 77 by the time it hits cylinder 2. This totals 116 cylinder traversals.

LOOK reads from end to end, then back again, over and over, but it never goes farther than the outermost outstanding read request. In this case, however, the outermost request is at 79, so it needs to go to the last cylinder, anyway. This yields the same 116 cylinder traversals.

C-LOOK resembles LOOK, but the head only reads while moving in one direction. Hence, it will traverse 39 cylinders on the way to the end, then another 79 going back to cylinder 0, then another 16 as it makes its way to cylinder 16 (which it skipped over on the way back down to 0!). This totals 134 cylinder traversals.

21. **Classic RISC cycle length**

The design in this problem broadly resembles the classic hypothetical RISC system discussed in [Patterson] but is somewhat simplified.

I *is true* because the cycle must be long enough to accommodate load instructions, which use memory twice, registers twice, and the ALU once.

II *is true* because the cycle only needs to be long enough that the slowest functional unit (typically memory) can settle. That way, instructions can be pipelined as in the diagram below, which depicts a series of four instructions, one from each type (BR, AR, LD, and ST):

BR Fetch	BR Decode	BR Execute	BR (Pause)	BR Write			
	AR Fetch	AR Decode	AR Execute	AR (Pause)	AR Write		
		LD Fetch	LD Decode	LD Execute	LD Memory	LD Write	
			ST Fetch	ST Decode	ST Execute	ST Memory	ST (Pause)

III *is false*, since the proposed change would probably demonstrate better throughput than the implementation in I and somewhat worse throughput than the implementation in II. It improves on I because now the faster instructions are executed in less time than the slowest instructions. However, it is not as good as II, since II can complete one instruction every $\max(M, R, A)$ ns, whereas III can only complete an instruction, at best, every $M + 2 * R + A$ or $M * 2 + R + A$ ns.

---

22. **Classic RISC stall length**

Suppose that an arithmetic instruction AR2 follows another arithmetic instruction AR1, and AR2 uses the register written by AR1. Then the decode stage of AR2 must follow the writeback stage of AR1:

AR1 Fetch	AR1 Decode	AR1 Execute	AR1 (Pause)	AR1 Write				
	AR2 Fetch	AR2 (Stall)	AR2 (Stall)	AR2 (Stall)	AR2 Decode	AR2 Execute	AR2 (Pause)	AR2 Write

Note that AR2 is effectively delayed for three cycles. In practice, it is silly to wait so long before beginning the decode stage of AR2, since the requisite operand already exists at the end of the execute stage of AR1 (though it has not yet arrived back in the register file). This value can then be fed directly into the execute stage of AR2. This is the essential insight of forwarding, which completely eliminates the three-stage bubble shown above. For the details of read-after-write (RAW) and other data hazards, refer to [Patterson].

23. **Handling hazards**

I *is true*: delayed control transfer, also known as delayed branching, is an attempt to cope with control hazards. II *is also true*: the branch target buffer stores the previous target address for the current branch, though other algorithms for branch prediction also exist. These are discussed in more detail by [Stallings] and [Patterson].

III *is definitely not true*. For any given instruction set architecture implemented on an N-stage pipelined processor, N registers probably is not enough registers to completely prevent structural hazards involving a shortage of register hardware. Besides this, structural hazards can result from an undersupply of other computational elements, such as ALUs.

---

24. **In-order retirement**

In-order retirement definitely does not require in-order scheduling of instructions. Scoreboarding is a technique whereby instructions can be started out of order and then stalled as needed until necessary operands become available.

In-order retirement by definition means that all instructions are committed in the same order they appear. This is useful for the implementation of precise interrupts, because upon resuming from the interrupt, the processor can easily identify which instructions need to be restarted from scratch.

---

25. **Amdahl's Law**

According to Amdahl's Law, if a fraction F of the work can be sped up by a factor of C, then the net speedup is $S = \dfrac{1}{(1-F) + \frac{F}{C}}$. If S is $250/150 = 5/3$, and C is 3, then solving for F produces 0.60.

Software Systems — Comments

26. **Compression**

Roughly speaking, run-length encoding (RLE) replaces a string of one repeated character with the value of the character and an integer showing how many times that character occurs. So, for example, RLE would replace "xxxxxxxx" with something like "x ⊢ 8 ⊢". (The '⊢' character would have to be a special control character that never appears in the file, or it might be a regular character like a backslash that must be escaped when not used as a control chararcter.) Several variations on this method exist, but none of them would provide significant compression of the file described by this problem, since the file does not contain any spot where a single character appears many times in a row.

Huffman encoding uses an alternate representation of each character such that common characters require fewer bits than uncommon characters. In this case, of 256 possible one-byte characters, only a dozen or so distinct characters appear in this file. Consequently, each of these could be represented with four or fewer bits, with the remaining 244 using five or more bits. In fact, some of those characters will take more than eight bits, but since they do not appear in this file, there is no cost to using such a lengthy representation for these rare characters. A lookup table will need to be prepended to the file's contents, in order to provide a mapping between the bit-level representations and the corresponding characters, but the significant savings from Huffman encoding should more than compensate for the overhead of this header.

Lempel-Ziv Welch achieves compression by replacing each recurring string of characters with a reference to the previous string. Again, exact implementations differ, but a file containing "abcdefghijklm abcde" might be replaced with "abcdefghijklm ⊢ 14 ⊢ 5 ⊢", where the "⊢ 14 ⊢ 5 ⊢" indicates a repetition of the string that starts 14 characters back and runs for 5 characters. A scheme like this will provide significant compression. Moreover, little if any space must be spent on a new header containing a lookup table.

---

27. **File management**

Three file allocation strategies are under consideration: storage in a linked list of large blocks, consecutive storage in a single block, and storage in a bunch of small blocks tracked by an index.

Clearly, random access reads will be terribly slow on a linked list (so choice A is wrong). Thus, the space allocation strategy clearly does matter (so choice D and E are wrong). In terms of supporting fast reads, the remaining two options are fairly workable, since the physical location of data can be ascertained in constant time.

However, the strategy must also demonstrate very little internal fragmentation. This precludes allocating a single chunk of disk to accommodate each file's maximum possible size. So choice B is wrong.

Two free space tracking strategies are under consideration: linked list and bit vector. Neither of these is important during a read, since the operating system has no reason to figure out what space is free during a read. Thus, choice C is correct.

For further discussion of these strategies, and others, refer to [Silberschatz] and [Tanenbaum].

28. **Recursion and stack size**

If `walk()` is called with $n = 1$, then no recursion occurs. If `walk()` is called with $n = 2$, then one level of indirect recursion occurs. If `walk()` is called with $n = 2^p$, then p levels of recursion occur.

So if `walk()` is called with $n = 1$, then the stack hits 14 bytes. If `walk()` is called with $n = 2$, then `run()` is called once and `walk()` is called twice, for a total of $18 + 14 * 2$ bytes. If `walk()` is called with $n = 4$, then `run()` is called twice and `walk()` is called three times, for a total of $18 * 2 + 14 * 3$ bytes.

Hence, if `walk()` is called with n in general, then the stack hits $18 * \log_2(n) + 14 * (\log_2(n) + 1)$ bytes.

---

29. **The what-does-this-recursive-function-return game**

Perhaps the easiest way to see the answer to this problem is simply to write out `EXPL(n)` for a few n:

n	`EXPL(n)`
1	1
2	1
4	2
8	6
16	24
32	120

Another convenient way to arrive at the same answer is to let $n = 2^p$ and then construct an equivalent function $\texttt{EXPL}_p$ that operates on a value of $p \geq 0$:

```
Function EXPLp( int p ) {
    if ( p <= 1 ) then return 1
    return EXPLp( p - 1 ) * p
}
```

This function represents the canonical recursive implementation of the factorial function.

---

30. **Row-major versus column-major**

Each integer is 2 bytes long, and there are 10 items in each row, so each row occupies 20 contiguous bytes in row-major order. Thus, row 0 occupies byte 0 through byte 19, row 1 occupies byte 20 through byte 39, and row 2 occupies byte 40 through byte 59. In particular, bytes 48 and 49 correspond to item 4 of row 2. (Note that rows and columns are indexed from 0 in this problem.)

In column-major order, since each column spans 5 rows, each column occupies 10 contiguous bytes. Thus, column 0 occupies byte 0 through byte 9, column 1 occupies byte 10 through byte 19, and so forth. In particular, item 4 at row 2 occupies bytes 44 and 45.

31. **Alignment**

The *length* element of a `LittleString` begins at byte 0 and ends just before byte 1. But the *contents* element is a multi-byte element and must be word-aligned, so it cannot begin until byte 4. It then ends just before byte 54. But 54, in turn, is not a multiple of 4, so an additional two bytes of padding must be added after the *key* element of the `TreeNode`. Thus, the `leftChild` element begins at byte 56, and the `rightChild` element begins at byte 60. Thus, each `TreeNode` requires a total of 64 bytes.

Full binary trees have two children under every non-leaf. So if the tree has 20 leaves, then it must have 19 interior nodes, for a total of 39 nodes. Thus, the tree as a whole requires 2496 bytes.

---

32. **Scoping and parameter passing**

Dynamic scoping means that non-local variables are resolved by looking at the bindings of variables in the stack of calling activation records. Consequently, stepping through this program would look like the following:

- The `foobar()` activation record inherits the binding of a, which equals 1.
- The `bar()` activation record then overrides this binding; here, $a = 0$.
- The `foo()` activation record inherits the binding of a from `bar()`, so $a = 0$ when it is printed.

Thus, I *is false*. (Note that the value printed did not depend on any parameters passed through function signatures. Consequently, it does not matter how parameters are passed.)

Lexical scoping means that non-local variables are resolved by looking at the bindings of variables in the enclosing scope. Pass-by-value-result means that variable values are copied as actual parameters, but then the variables are updated when the function completes. (This is a blend of pass-by-value, where the actual parameters are a copy of the caller's variables, and pass-by-reference, where the actual parameters point to the memory locations of the caller's variables during the entire function call.) Consequently, stepping through this program would look like the following:

- In `foobar()`, b corresponds to the b in the outermost scope, and r is a local variable equal to 1.
- When `bar()` is called, x is initialized to 1 (the value of r), and y is initialized to 1 (the value of b). In addition, b is the same as b in the outermost scope.
- The call to `foo()` does not change anything.
- Back in `bar()`, x is then set to 0. (This does not affect r immediately.)
- Then, $y = 1$ is subtracted from $b = 1$. This immediately changes the global b variable to 0.
- Lastly, y is incremented to equal 2. (This does not immediately affect b.)
- When `bar()` returns, r and b are updated to reflect the new values of x and y. Since x last equaled 0 and y last equaled 2, r becomes 0 and b becomes 2.

Thus, when b is printed, a 2 appears. II *is false*.

Finally, as noted earlier, pass-by-value simply copies the value in without updating the caller's variables. Thus, the call to `bar()` cannot affect the value of r in `foobar()`. Therefore, when r is printed, it still equals 1, and III *is true*.

33. **Garbage collection**

Stop-and-copy garbage collection allocates objects in one half of the total available memory space. These objects can refer to one another. Once in a while, the virtual machine stops executing instructions and starts copying live objects into the other half of memory (beginning with a collection of root objects). Any object referenced by a live object is also deemed live and recursively copied as well. As the objects are copied into the other half of memory, they can be adjoined to one another, thereby compacting them into an efficient piece of memory. Once all the live objects have been copied, everything still in the old half of memory is thrown away.

Mark-and-sweep garbage collection allocates objects throughout the entire available memory space. These objects can refer to one another. Once in a while, the virtual machine stops executing instructions and starts flagging live objects (beginning with a collection of root objects). Any object referenced by a live object is also deemed live and recursively flagged as well. After no more objects can be marked, then all unmarked objects are swept away so their memory may be reused. Mark-and-sweep can be implemented as an incremental algorithm by assuming that all objects created since the mark-and-sweep started are still live; if any objects violate this assumption, they can be collected during the next mark-and-sweep.

Reference counting tags each object with an integer indicating how many other objects refer to the object. (Root objects start with a tag equal to 1.) When a reference is created, this integer is incremented, and when a reference is released, this integer is decremented. When the integer hits zero, the object is destroyed. Unfortunately, if two objects mutually refer to one another, then their reference counts can never hit zero, so they will never be destroyed.

---

34. **ACLs versus capabilities**

A capability is essentially a handle or pointer to a specific object. By keeping processes from acquiring handles on resources, an operating system can prevent unauthorized access. On the positive side, once a process has a handle, it can access the object directly without troubling the operating system further; on the negative side, once a process has a handle, it knows about the object, making revocation difficult. One strategy is to use an extra level of indirection, so that handles reference intermediate objects.

Access Control Lists (ACLs) are quite flexible and generally demonstrate easy-to-understand semantics (so III *is false*). However, they must be checked by the operating system upon every operation, which can be a performance problem if the ACLs are complex and long. Consequently, many systems implement a simplified ACL scheme, where instead of specifying what operations every single user can do on each object, they instead specify what operations can be executed by groups of users on each object.

35. **Conditions for deadlock**

Deadlock requires the following four conditions to hold true:

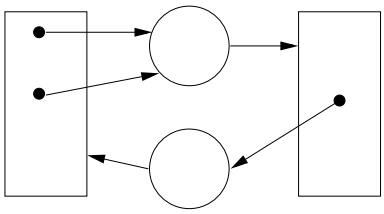
- Mutual exclusion: resources are not shareable.
- Hold and wait: a process can grab some of its required resources opportunistically and hold them until it can acquire other required resources.
- No preemption: the system cannot kill any process or take away its resources.
- Circular wait: not all processes request resources in the same order, so it is possible for them to be waiting in a circle for one another (A waits for B, B waits for C, and C waits for A).

Based on these considerations, choices A, B, and C are true.

The Banker's algorithm distinguishes between deadlock-free (safe) and deadlock-prone (unsafe) states. Processes must declare in advance what resources they may require in the future. The system can track its inventory of each resource with the goal of having enough resources to supply the future needs of running processes. That way, when a new process wants to start up, the system can decide whether it has enough resources to serve this new process without putting existing processes in jeopardy of deadlock.

The resource allocation graph depicts processes and the resources they depend on. One circle appears for each process. One little box appears for each type of resource, with a dot in the box for each instance of the resource. Arrows appear from circles to boxes if the process is waiting for that resource; arrows appear from dots to circles if that resource instance is allocated to that process.

Cycles are a necessary but not sufficient precondition for deadlock to exist. However, if each box in the cycle contains only one dot, then deadlock has occurred. Deadlock may exist in other circumstances, as well, such as in the graph below.



---

36. **User-level versus kernel-level threading**

In kernel-level threading, the kernel manages the threads; in user-level threading, a software library rather than the kernel manages the threads.

Consequently, I *is false*, since in user-level threading, the kernel does not even keep track of individual threads within the process control block. Likewise, II *is wrong*, since the process is "opaque" to the kernel in the sense that the kernel cannot determine whether which thread is blocked, let alone whether another thread is runnable. Thus, in user-level threading, if one thread blocks on I/O, then the whole process blocks, including any other threads.

However, III *is true* because in kernel-level threading, an interrupt must fire and receive service from the kernel before one thread can take over from another.

37. **Semaphores and mutexes**

I *is false*. Although V does need to implement non-blocking increment, and P must block until it can decrement the semaphore, that alone does not assure success. Note that in the specified implementation, it is possible for one thread to call `add()` while another calls `remove()`. Since the queue is not thread-safe, the program may crash.

II *is true* because the underlying queue's methods are not thread-safe. Imagine if thread T0 completely finished executing `loadem()` before threads T1 and T2 began to execute `printem()`. Suppose that T1 and T2 then start reading `printem()`. Each would successfully decrement *smx* and then could begin calling `Q.remove()`. Unfortunately, as stated in the problem, the queue is not thread-safe, so the program could crash if both T1 and T2 attempted to execute `Q.remove()` at the same time.

III *is false*. Suppose that thread T0 completes one loop of `loadem()` (and releases the mutex), but then thread T1 executes two loops of `printem()`. T1 will attempt to call `Q.remove()` twice on a queue that only contains one element!

---

38. **Priority inversion**

In priority inversion, a high-priority process ends up waiting for a low-priority process to complete. It usually results from a low-priority process grabbing some resource and getting preempted by a medium-priority process; at some point, a high-priority process needs that resource. The high-priority process cannot continue until the low-priority process releases it which cannot happen until the medium-priority process completes.

The result, as in the case of this problem, is starvation of one or more processes for a long period of time. Deadlock will not occur, however, as the medium-priority process eventually does complete, allowing the other processes to continue and complete. Note that if the operating system was a starvation-free operating system, then the low-priority process would occasionally run, which might allow it to release the resource well in advance of the completion of the medium-priority process.

Another solution is the priority-inheritance protocol, in which any process (even the low-priority process) gains a temporary priority boost if it holds a resource required by a higher priority process. That way, if the operating system is preemptive, it will block the medium-priority process until the resource is released.

39. **Round-robin job scheduling**

Round-robin scheduling splits up CPU time into little slices, and processes wait in a circular queue for a chance to use a slice of CPU time. This strategy and others are discussed by [Silberschatz] and [Tanenbaum]. Round-robin has the following properties:

- Starvation cannot occur. That is, if a job waits long enough, it will receive some service and eventually be completed.
- Round-robin does not produce optimal throughput of jobs. Instead, the shortest-job-first (SJF) algorithm completes the largest number of jobs in a given amount of time.
- Round-robin involves a good deal of context switching overhead. After each little slice of CPU time, the operating system must switch to another process (if more than one process is running).
- Round-robin does not deliver the optimal response ratio, which is defined to equal the execution time divided by the total time (from arrival to finish). SJF generally gives a superior average response ratio.
- Round-robin does not guarantee that jobs will finish in the order that they arrive. For example, if job J1 requires sixty slices of CPU time and J2 requires only one slice, then J2 will finish approximately 59 time slices before J1.

---

40. **Spin-wait**

To intuitively understand this problem, suppose that process B has the lock on R but will be done with it in 1 microsecond. Suppose that process A wants the lock, but doing a context switch to some other process C will require a million years. It would make sense to let A loop (spin-wait) until B releases the lock, since it clearly be inefficient to try switching to another process.

That is the insight required to solve this problem. But the challenge is now to make sure that process B can actually continue to make progress on its job so that it can release the lock quickly.

As in the case of choice B, if process A hogs the system's only CPU while spin-waiting, then B will never get a chance to run. In this case, A should yield. Likewise, as in choice C, if there are multiple CPU's but B is not running (with the implication that it might not run any time soon), then A should yield.

As in the case of choice D, if process B will take a million years to complete, but a context switch takes only a microsecond, then a multi-processor system should switch to another process on the free CPU. But as in the case of choice E, if process B will only take a microsecond, but a context switch takes a million years, then a multi-processor system should allow A to spin-wait on one CPU.

In short, it is too simple to say that spin-waiting is always good or always bad, as in choice A.

41. **Process migration**

[Sinha] lists a number of potential benefits owing to process migration:

- If some machines are too heavily utilized, then jobs will stack up on those machines. Moving jobs to another machine will improve response time as well as throughput.
- If a process needs a certain piece of hardware (such as a graphics card) or needs to communicate with another program, why not co-locate that process with the resource it requires? It reduces network overhead, which helps speed up this process; in addition, it reduces network utilization, which might improve communication between other processes running on other machines.
- Reliability and availability can be improved by migrating processes to more reliable machines, or by migrating copies of the program to multiple machines (so as to support fail-over in case one machine crashes).

---

42. **Latency on distributed systems**

If a process is significantly delayed because it spends significant time waiting for a certain heavily utilized computational resource, then adding extra copies of that resource may improve latency.

If a process is significantly delayed due to the network or other latency of doing reads from remote data sources, then prefetching data may improve latency. This prefetching could be done in a separate thread from the main computational thread; in general, if one thread could get a worthwhile operation done while waiting for another thread to complete some lengthy operation, then performing both operations in parallel through multithreading may improve the process's overall latency.

If a process is significantly delayed due to waiting for a write to complete, then utilizing non-blocking writes may improve latency. The process could continue with other work while another process (perhaps on a remote machine) completes the write.

Locking more resources with mutexes will generally not improve latency. In fact, it usually will worsen latency, since processes will now need to take turns executing code, rather than proceeding as quickly as possible. This can cause particular problems if slow operations are attempted while holding the mutex.

---

43. **Wait-die versus wound-wait**

The wait-die and wound-wait schemes prevent deadlock. The following chart summarizes what happens in various circumstances:

	Wait-die	**Wound-wait**
Young process Q needs resource held by old process P	Q dies	Q waits
Old process P needs resource held by young process Q	P waits	Q dies

44. **Remote development and development tools**

Source code control systems help track source code files as developers simultaneously add onto different parts of the application. Most source code control systems allow each developer to take a copy of the entire application's source code, which enables them to make small changes that are then merged back into the official copy once the code is working. One major problem with having all developers work on the same code base is that source code does not work properly while a new feature is still being developed; in fact, the code may not even compile. Consequently, the workers at this company may find themselves continually frustrated, since it will be difficult to add new features while other workers are also adding features.

Moreover, because the source code is located on a remote machine, the compiler and other tools must read files across the network. For large projects, this network overhead can easily double the time it takes to perform a build. This, in turn, will reduce the number of edit-compile-test cycles that each developer can complete each hour, thereby lowering overall productivity.

Finally, some tools will be more expensive because they must open sockets to the remote machine rather than connecting locally. This is not the case with compilers, assemblers, or linkers (so III *is false*), since these see a *file system interface* and have thus have no reason to open a socket. *However, some other tools* may prove problematic. For example, among the top-tier testing and debugging tools (those with component server integration and a GUI), support for remote operation currently costs a good deal.

In short, making all developers work like this is generally poor software engineering, since it can reduce productivity and increase costs. One redeeming aspect of web development is that many artifacts (such as JavaScript and HTML pages) are generally interpreted and very loosely coupled. This can help overcome some of the problems discussed above.

45. **Fast Ethernet**

Like earlier versions of Ethernet, the 100Base-T Fast Ethernet data link protocol is a bus protocol based on CSMA/CD. Consequently, the protocol involves collisions on the common bus, which can lead to poor latency as hosts attempt to resend their messages; it also leaves the door open to packet sniffing by hosts on the same segment. The widespread twisted-pair variety of Fast Ethernet (100Base-TX) only supports up to 100 meters and 100 MBps. Later versions of Ethernet deal with many of these weaknesses. Ethernet is the most widely deployed data link protocol, and consequently, it enjoys broad support from vendors.

---

46. **Networking protocols**

To serve the request, the client must first resolve the host name using the domain name service (DNS) protocol, which in turn relies on the user datagram protocol (UDP) and the internet protocol (IP) at a still lower layer. Once the hostname is converted to an IP address, the client opens a transmission control protocol (TCP) connection to the server, again using IP at a lower layer. The browser transmits a hypertext transmission protocol (HTTP) request, which the server interprets before sending an HTTP reply. At no point is the sendmail transport protocol (SMTP) an important part of this process.

The top layer of protocols is the application layer. Typical application layer protocols are HTTP (for web content), HTTPS (for secure web content), FTP (for file transfer), SMTP (for sending mail), telnet (for text based logins), and SNMP (for managing networks).

The next layer is the transport layer. The two crucial protocols in this layer are TCP (for reliable, in-order delivery of packets) and UDP (for unreliably transmitting short snippets of information).

The next lower layer is the network layer. The crucial protocol in this layer is IP (for addressing of datagrams over the internet).

The next lower layer is the data link layer. The crucial protocol in this layer is Ethernet (mainly for sending data over twisted-pair cables), but another is FDDI (for sending data over fast fiber channels).

The lowest layer is the physical layer. One important specification in this layer is RS-232.

47. **Switching, routing, and UDP**

In general, UDP packets can arrive out of order when transmitted over the internet. This occurs for several reasons. One is that they may be dropped (since successful transmission is not guaranteed for UDP), forcing the application layer to resend some packets; this issue can be ignored in this problem. The other main reason why packets arrive out of order is because different packets might take different routes.

Circuit-switched networks establish a dedicated physical connection between the sender and receiver. Thus, all packets take the same route and cannot get out of order. Packet-switched networks do not physically dedicate hardware in this manner. Instead, all the packets ride on a shared network of arteries, just as all cars share the highway rather than each getting their own personal road.

With fixed routing tables, all the intervening routers know of only one way for the packets to get to the destination. Consequently, all packets take the same route.

In a virtual circuit routing scheme, the routers identify a route at the beginning of the session. This route is then used to move all packets within that session. Consequently, they all take the same route.

In a dynamic routing scheme, each packet can take a different route. The actual algorithm chosen for routing the packets varies but typically attempts to optimize the network's average overall throughput or latency, or a combination of the two.

---

48. **Datagram fragmentation**

From [Sinha]: "If a datagram is found to be larger than the allowable byte size for network packets, the IP breaks up the datagram into fragments and sends each fragment as an IP packet. When fragmentation does occur, the IP duplicates the source address and destination address into each IP packet, so that the resulting IP packets can be delivered independently of each other. The fragments are reassembled into the original datagram by the IP on the receiving host and then passed on to the higher protocol layers." Note that each network can have a different maximum transmission unit (MTU).

---

49. **Probability and SNMP**

The question is indifferent to when the printers shut down, so they can be ignored, leaving only the three servers and three clients. There are $6! = 720$ possible orderings over these six machines. Of these, only $(3!) * (3!) = 36$ orderings would have three clients followed by three servers. Because all orderings are equally probable, the requested probability is $36/720 = 1/20$.

50. **Availability**

The per-day probability of repair must be converted into a mean time to repair (MTTR) as follows:

Days till repair	Probability of repair on these days
0 to 1	0.25
1 to 2	$0.75 * 0.25$
2 to 3	$0.75^2 * 0.25$
3 to 4	$0.75^3 * 0.25$

Hence, $\boldsymbol{Pr}[\text{day } n-1 \text{ to } n) = 0.75^{n-1} * 0.25$

$$\langle \text{days} \rangle = \sum_{n=0}^{\infty} (0.25) * n * (0.75)^{n-1}$$

If you have forgotten how to evaluate this sum, consider some function

$$F(x) = \sum_{n=0}^{\infty} x^n = \frac{1}{1-x}$$

Note $F\prime(x) = \sum_{n=0}^{\infty} n * x^{n-1} = \frac{1}{(1-x)^2}$

Hence, with $x = 0.75$,

$$\langle \text{days} \rangle = \frac{0.25}{(1-0.75)^2} = 4$$

However, if the repair occurs at the start of the day, then MTTR = 3; but if the system is not functional until the end of the day, then MTTR = 4.

Mean between failures (MTBF) was specified to be 30 days. Using MTTR = 3 yields Availability = 91%, and using MTTR = 4 yields Availability = 88%. In either case, choice C is closest.

$$\text{Availability} = \frac{MTBF}{MTBF + MTTR}$$

Algorithms and Data Structures — Comments

51. **Euclid's algorithm**

FMR implements Euclid's algorithm for finding the greatest common denominator of two positive integers. It does not require that the arguments be passed in the "right" order, since the algorithm simply swaps their order on the first recursion if `lowValue > highValue`. (Try out `lowValue = 18` and `highValue = 12`.)

Here is how to see that the algorithm runs in logarithmic asymptotic algorithmic complexity. Suppose that the arguments are now in the "right" order so that `lowValue` $\leq$ `highValue`. If `lowValue` $\geq$ `highValue / 2`, then `modValue = lowValue`, and the algorithm terminates after one recursion, which would be $O(1)$ time.

So suppose that `lowValue` does not exceed `highValue / 2`. This `lowValue` becomes the `highValue` on the next recursive call, so `highValue` is slashed by at least a factor of 2. Consequently, the algorithm cannot take more than logarithmic time to drive `highValue` down to 0.

---

52. **Sieve of Eratosthenes**

This "sieve of Eratosthenes" does require $O(N)$ storage, with one cell for each number to be considered for prime-ness. Since y grows by at least x per repetition of the inner loop, the run time of the inner loop is $O(N/x)$. Hence, the overall execution time is equal to $\sum_{x=2}^{N} O\left(\frac{N}{x}\right)$. Since the harmonic series converges to $\ln(N)$, the run time of the sieve is $O(N \ln(N))$.

Note, however, that this implementation has a defect. Specifically, y should be initialized to $2 * x$, not to x. The algorithm as implemented will claim that every integer greater than 1 is a composite number!

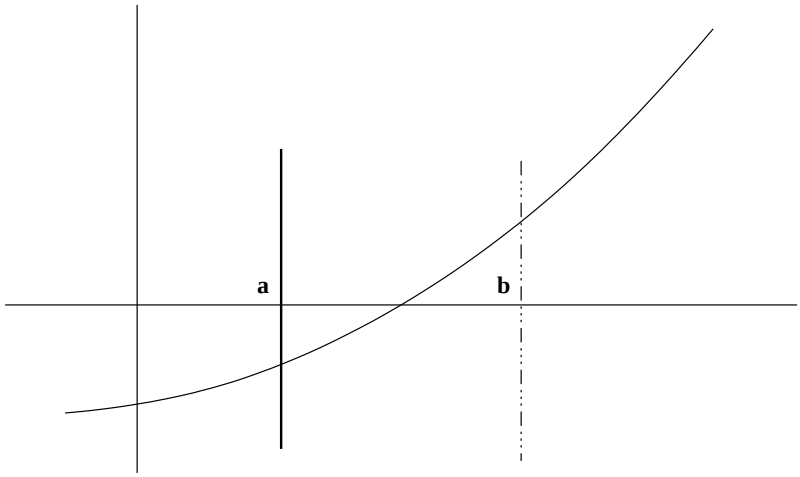
---

53. **Methods for finding a function's root**

The goal is to find where a function $f(x)$ crosses the x axis. The method of false position and the bisection method try to bracket the target value of x with a "high" value of b and a "low" value of a. Note that $f(a) < 0$ and $f(b) > 0$. These methods guess a new value x and compute $f(x)$. If $f(x) < 0$, then x is used to overwrite a for the next iteration of the algorithm; conversely, if $f(x) > 0$, then x is used to overwrite b for the next iteration of the algorithm. In this manner, the boundaries constantly close in on the goal.

Newton's method is similar in its goal, but it only uses a single value of x rather than two bounds. It drives this value of x up or down depending on whether $f(x) < 0$ or $f(x) > 0$, respectively. Although Newton's method tends to converge rather rapidly, it can fail to converge at all in some situations. At each iteration, Newton's method and the method of false position each look to see how far $f(x)$ is away from 0. They then divide this distance by an estimate of the slope to tell how big of a step to take.

Newton's method works by examining the derivative of the function and using it to estimate how far to walk along the x axis in order to zero out $f(x)$. Here, $f'(x) = 2x$. If Newton's method is initialized with $a = 1$ and $f(a) = -4$, then it notes that the derivative is $f'(a) = 2$, meaning



that a step of $4/2 = 2$ should be taken to the right. Thus, the next estimate is $x = a + 2 = 3$. Here, $f(x) = 4$ and $f'(x) = 6$, so the next step is 4/6 to the left, yielding $x = 2.3333$.

The method of false position estimates the slope of f between two points, rather than using the derivative. If it is initialized here with $a = 1$ and $b = 3$, with $f(a) = -4$ and $f(b) = 4$, then the slope is $8/2 = 4$, so the next guess is $4/4 = 1$ to the right of a, yielding $x = 2$. Here, $f(x) = -1$. Thus, the root of f must be between x and b, so x is used to replace a. Now the slope is estimated as $5/1 = 5$, so the next guess is $1/5$ to the right of a, yielding $x = 2.2$.

The bisection method simply guesses that x is the average of a and b. If the method is initialized with $a = 1$ and $b = 3$ here, with $f(a) = -4$ and $f(b) = 4$, then the method guesses $x = (1+3)/2 = 2$, for which $f(x) = -1$. Thus, x is used to overwrite a. The next estimate is $(2 + 3)/2 = 2.5$.

To summarize, Newton's method first guesses $x = 3$, and then it guesses $x = 2.3333$. The method of false position first guesses $x = 2$, and then it guesses $x = 2.2$. The bisection method first guesses $x = 2$, and then it guesses $x = 2.5$. So only III *is true*. Interestingly, although Newton's method generally does well, the method of false position has come closest to the correct answer of ≈ 2.236.

54. **Multiplying matrices**

It is true that swapping the first and second lines would have no effect on the correctness of the result. However, depending on whether this array spans multiple memory pages, and depending on the operating system's paging algorithm, swapping the first and second lines can have a gross effect on performance.

The code multiplies A with B and stores the result in C. Strassen's divide-and-conquer algorithm improves on this algorithm by recognizing that each matrix can be divided into smaller pieces; the algorithm calculates 14 helper matrices then recombines them to form C, achieving $O\left(n^{\log_2 7}\right)$ worst-case asymptotic algorithmic complexity.

---

55. **Matrix algorithms**

Gaussian elimination transforms a system of linear equations into reduced row echelon form. It does so by adding a scaled copy of each equation to other equations in order to progressively eliminate variable coefficients from left to right. It runs in $O(n^3)$ asymptotic algorithmic complexity.

Matrix multiplication loops through all the cells of the destination matrix, filling in each cell with the dot product of one row from the first matrix and one column from the second matrix. It runs in $O(n^3)$, where n is the maximum dimension of the matrices.

The simplex method explores the boundaries of a multi-dimensional "feasible" subspace within a larger space. It seeks a point within the space that maximizes a linear objective function subject to linear constraints. It usually runs in polynomial time (where n is the number of dimensions), but its worst case asymptotic algorithmic complexity is exponential.

Matrix inversion is no more difficult than conversion to row echelon form, and Gaussian elimination can serve as the workhorse for either purpose. The naïve algorithm for calculating a determinant, using the definition provided by Leibniz, leads to $O(n!)$ asymptotic algorithmic complexity. However, since adding a multiple of one row to another does not change the determinant, a procedure like Gaussian elimination can be used to transform the matrix in $O(n^3)$ time so that all elements are on the diagonal, making it easy to then calculate the determinant in linear time.

---

56. **Big-O**

Searching linked lists runs in average algorithmic complexity of $O(n)$. That can be verified here, where it takes 10 ms to search 2^{10} records and $10 * 2^{10}$ ms to search 2^{20} records. Thus, it appears that $T(n) = 10 * 2^{-10} * n$, where $T(n)$ is the time in milliseconds to search through n records.

Binary search runs in average algorithmic complexity of $O(\log_2 n)$. That can be verified here, where it takes 40 * 10 ms to search 2^{10} records and 40 * 20 ms to search 2^{20} records. Thus, it appears that $U(n) = 40 * \log_2 n$, where $U(n)$ is the time in milliseconds to search through n records.

The goal is now to find a value of n for which $T(n) = U(n)$, that is, $10 * 2^{-10} * n = 40 * \log_2 n$.

If $n = 2^{16}$, then $T(n) = 10 * 2^6 = 640$ ms, and $U(n) = 40 * 16 = 640$ ms.

57. **Sorts' best-case run time**

Here are the big-O asymptotic algorithmic complexities for some popular sorting algorithms:

	Best	**Average**	**Worst**	**Notes**
Bucket	n	n	n^2	
Counting	$k+n$	$k+n$	$k+n$	integers only in $[0,k)$
Heap	$n*\log_2 n$	$n*\log_2 n$	$n*\log_2 n$	
Insertion	n	n^2	n^2	
Merge	$n*\log_2 n$	$n*\log_2 n$	$n*\log_2 n$	
Quick	$n*\log_2 n$	$n*\log_2 n$	n^2	
Quick-Random	$n*\log_2 n$	$n*\log_2 n$	n^2	
Radix	$d*(n+2^d)$	$d*(n+2^d)$	$d*(n+2^d)$	d digits; ints only in $[0,2^d)$
Selection	n^2	n^2	n^2	
Shell	$n^{1.25}$	$n^{1.5}$	$n^{1.5}$	approximate big-O values

Bucket: Break the possible range of input values into n sub-ranges. For each sub-range, create a linked list. For each element in the array, add the item to the corresponding linked list. Use insertion sort on each linked list. Concatenate the linked lists.

Counting: For each possible element value $[0,k)$, count how many elements have that value. Then, "roll up" these totals so that they now refer to the number of elements with less than or equal to that value. Finally, beginning with the last element and moving toward the beginning of the array, use the element's value to index into the counts array, which will tell where the element should end up in the output; put it there, and then decrement the corresponding value in the counts array.

Heap: Treat the array as a binary tree. Beginning with the midpoint of the array and going toward the beginning of the array, examine element x. Then "heapify" x: If x is smaller than either of its children, swap it with its child; repeat the examination of x in its new position and repeat the swap as long as necessary so that x is bigger than both its children (or it ends up in a leaf node). Heap sort then pulls items out of the heap by swapping the root with the last node and repeating the "heapify" process for just the root.

Insertion: For each element x in the array, read backwards (toward the array's start) until finding an element y that is smaller than x. Insert x to the right of y, shifting to the right all of the elements between y and x's old position.

Merge: Divide the array in half. Recursively sort the left half. Recursively sort the right half. Now treat each half as a queue. While both queues are non-empty, examine the head element of each queue and choose the lower of the two heads; remove that element from its queue and put it into the output. Once only one queue still has elements, stream them to the output.

Quick: Partition the elements so that all elements to the left of some slot are less than all the elements to the right of the slot. Recursively sort the left half. Recursively sort the right half.

Quick (Randomized): Same as quick sort, except that some elements are randomly exchanged prior to partitioning. The worst case of quick sort occurs when the elements start out sorted, and random swapping helps to prevent that from happening very often.

Radix: Use a stable sort (e.g.: counting sort) to order all elements according to their rightmost digit. Then repeat for the second to last digit. Repeat for the remaining digits.

Selection: Find the smallest element and exchange it with the element at slot 0. Find the next smallest, and exchange it with the element at slot 1. Continue for the rest of the slots.

Shell: Beginning with $k \approx n/3$, make sure that each element x is sorted with respect to the elements in positions $x + k$ and $x - k$. (Use insertion sort or equivalent.) Then divide k by 3 and repeat. Keep repeating until $k < 1$. Now every element x is sorted with respect to its immediate neighbors.

---

58. **Sorts' worst-case run time**

See the comments on the previous problem.

---

59. **Counting compares and exchanges in simple sorts**

As noted in [Sedgewick], selection sort uses about $N^2/2$ comparisons and N exchanges on average. Insertion sort uses about $N^2/4$ comparisons and $N^2/8$ exchanges on average. Bubble sort uses about $N^2/2$ comparisons and $N^2/2$ exchanges on average.

So if exchanges cost nothing and only comparisons count, then insertion sort edges out the others. If exchanges count but comparisons cost nothing, then selection sort beats the others asymptotically.

---

60. **Tradeoffs among sorting algorithms**

Choice A *is false* because the asymptotic algorithmic complexity of quick sort is $O(n^2)$, whereas that of insertion sort is $O(n^2)$ and that of merge sort is $O(n \log_2 n)$.

Choice B *is true*. The average asymptotic algorithmic complexity of quick sort is $O(n \log_2 n)$, whereas that of insertion sort is $O(n^2)$ and that of merge sort is $O(n \log_2 n)$. But when merge sort cannot be used, quick sort is certainly a reasonable alternative.

Choice C *is false* because when the inputs are sorted, the asymptotic algorithmic complexity of quick sort is $O(n^2)$, whereas that of insertion sort is $O(n)$ and merge sort is $O(n \log_2 n)$.

Choice D *is false* because quick sort really needs to do many random accesses. In contrast, merge sort needs very few random accesses.

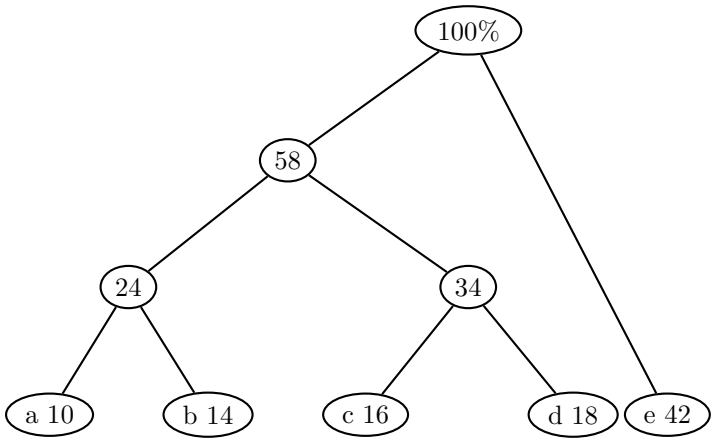
Choice E *is false* because quick sort is not a stable sort, but insertion sort and merge sort are stable. That is, quick sort may invert equal elements during partitioning, whereas insertion and merge sort never invert equal elements at any step.

61. **Huffman code**

Build an optimal Huffman code as follows:

- Put each of the characters in a box at the bottom of the paper. These boxes will end up being the leaves of a tree.
- Group the two least-occurring characters together by drawing a parent box with the two character boxes as children. This new box represents "either of these two children," and its occurrence equals the sum of the occurrences of its children.
- Repeatedly group together the two boxes with the lowest occurrence, then draw a parent that merges them. When only one box is left, that is the root of the tree.
- Finally, trace the path from the root to each node. Whenever a left branch is taken, write a 0; whenever a right branch is taken, write a 1. For example, in the tree below, d is reached with one left branch and then two right branches, so its code is `011`.

Here is the tree that results from applying this algorithm to the given character occurrences.



62. **Radix-like tree**

Choices A, D, and E *are true* because the structure just contains a flag for each present entry. The structure does not change as keys are stored. Insertion and search both take time proportional to the depth of the tree, which has a constant three levels here.

Choice B *is true* because the left-child right-sibling representation would allow the programmer to represent the same set of entries without as many links. Instead, each parent would have a pointer to the head of a linked list of extant children, and portions of the tree could be omitted unless they actually had entries beneath them. (Incidentally, if this optimization was implemented, then the ordering of insertions may affect the order of keys within the tree.)

Choice C *is false* because the problem specifies that insertions occur much more often than searches. The proposed flag would be beneficial during searches, since the search algorithm could be optimized to avoid looking in sub-trees that have no entries. However, the proposed flag would need to be set at each level on every insertion.

Note that this so-called RD10 tree is actually just an inefficient radix tree where values only appear at the leaf level. To conserve memory, radix trees are generally constructed so that empty sub-trees are omitted.

---

63. **Successor in binary tree**

This question is more tricky than hard. One thing is to remember that all leftward descendants of x's right child r have keys between x and r; finding the least of these gives the successor of x.

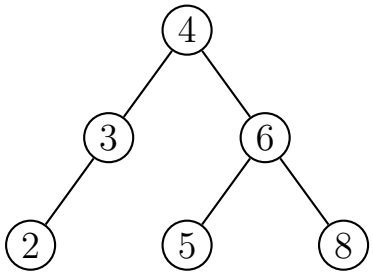
The other trick is that x might not have a right child. In that case, x is the rightmost descendant of some ancestor z, which is the left child of a node y (or z is the root, in which case x has no successor in T). The thing to remember here is similar to the thing to remember in the paragraph above. Specifically, all rightward descendants of y's left child z, including x, have keys between z and y. So y is the successor to x.

---

64. **Binary search tree**

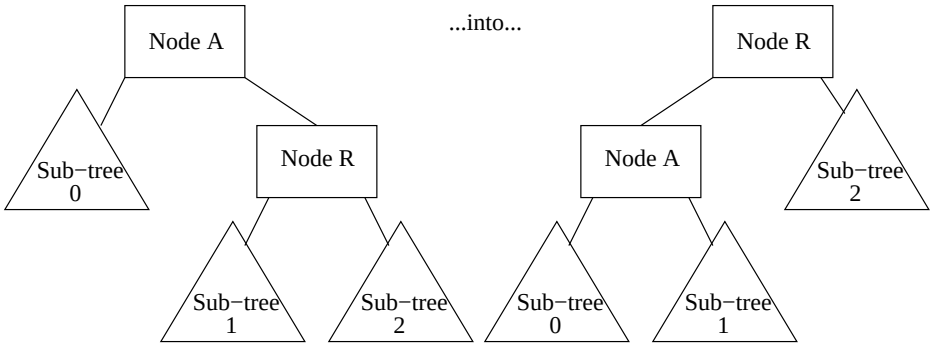
The resulting tree appears below:

Finding 4 now takes one comparison (with the root), finding 3 and 6 take two comparisons, and finding the other nodes takes three comparisons. There are three leaf nodes and three interior nodes. If 7 is inserted, it will go onto a new level, as the left child of 8.

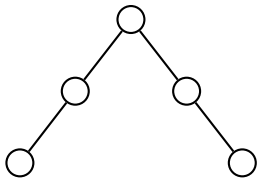


65. **AVL tree**

During insertion, rotations might occur to keep the tree balanced. (That is, the two sub-trees of any node cannot differ in height by more than 1.) For example, a left rotation turns a tree like the following...



These manipulations ensure that the height of T cannot exceed $1.44 * \log_2 n$. Insertion, deletion, and search can proceed in $O(\log_2 n)$ asymptotic algorithmic complexity. Note that the number of interior nodes can exceed the number of leaves. An example appears below:



If the height of T is $O(\log_2 n)$, and the nodes are augmented with an attribute γ such that the value of γ at a node only depends on the value of γ at the node's children, then γ can be maintained in $O(\log_2 n)$ time during insertions and deletions. The reason is that if the value of γ changes at some node, then only the $O(\log_2 n)$ ancestors of that node need to be tweaked so that their value of γ becomes correct again.

An order statistic search means finding the k^{th} largest node in tree T. If each node in T contains an attribute γ which tells the size of that node's sub-tree, then finding the k^{th} largest node is straightforward. (Assume that $\gamma = 0$ for empty/non-existent nodes, i.e.: nil. $\gamma = 0$.)

```
x = root
while ( x.γ != 0 ) {
     g = x.left.γ + 1
     if ( g == k ) then return x
     if ( g > k ) then x = x.left
     if ( g < k ) then {
          k = k - g
          x = x.right
     }
}
return nil
```

66. **Red-black tree**

As with AVL trees, rotations must occur during insertion and deletion. In this case, however, the properties to maintain are the following:

- Every node is assigned a color, either red or black.
- The root and leaves are always black. (Leaves are placeholder "nil" values in a red-black tree.)
- If a node is red, then its children are black.
- Every path from a leaf to the root contains the same number of black nodes.

These properties, particularly the last one, help to ensure that the tree is always $O(\log_2 n)$ in height. In fact, the distances from two leaves to the root cannot differ by more than a factor of 2. Insertion, deletion, and search all proceed in $O(\log_2 n)$ asymptotic algorithmic complexity.

Because the tree is full (every leaf is occupied by a "nil"), the number of leaves equals 1 plus the number of interior (data) nodes.

If each node is augmented with an integer γ showing the size of that node's sub-tree, then the rank r of a node x can be determined with the pseudo-code below. (Assume that $\gamma = 0$ for empty/non-existent nodes.)

```
r = x.left.γ + 1
while ( x is not the root ) {
     if ( x is a right child ) then {
          r = r + x.parent.left.γ + 1
     }
     x = x.parent
}
```

67. **B-Tree**

[Cormen] discusses B-Trees in depth. The basic idea is that each node has somewhere between $t-1$ and $2t-1$ keys that segregate children into between t and $2t$ corresponding groups. Each group is placed in a sub-node. Thus, B-Trees are a generalization of binary trees (which use a single key in each node to segregate children into two groups). However, like AVL and Red-Black trees, B-trees implement a form of balancing.

Nodes may be split during key insertion (beginning with the root, if necessary). This has the interesting effect that younger nodes tend to appear near the top of the tree, whereas with binary trees, younger nodes appear near the bottom of the tree. Rotations are not used during insertion.

Since each node has a minimum of t children, the height of the tree is $O(\log_t n)$ and works out to the exact formula given by the problem. Thus, searching for a node takes no more time than reading down the tree in $O(h)$ steps and reading across each node in $O(t)$ steps for a total of $O(t*h)$ CPU operations and $O(h)$ disk operations, where t is a constant. It turns out that even though the tree may grow due to splits during insertion, an insertion still takes no more than $O(t*h)$ CPU operations, so inserting n nodes takes $O(n*t*h) = O(n*h)$ CPU operations.

68. **Hash table**

Consider the linear probing strategy in I. Here is the hash table after each insertion:

After inserting 14							14	
After inserting 23							14	23
After inserting 0	0						14	23
After inserting 6	0	6					14	23
After inserting 3	0	6		3			14	23
After inserting 11	0	6		3	11		14	23

Consider the quadratic probing strategy in II. Here is the hash table after each insertion:

After inserting 14							14	
After inserting 23							14	23
After inserting 0	0						14	23
After inserting 6	0	6					14	23
After inserting 3	0	6		3			14	23
After inserting 11	0	6		3		11	14	23

Consider the double hash probing strategy in II. Here is the hash table after each insertion:

After inserting 14							14	
After inserting 23							14	23
After inserting 0	0						14	23
After inserting 6	0				6		14	23
After inserting 3	0			3	6		14	23
After inserting 11	0			3	6	11	14	23

So it turns out that none of the above *is true*!

69. **Breadth-first and depth-first searches**

Breadth-first and depth-first search each explore a graph. They maintain a collection of nodes waiting to be visited; while the collection is non-empty, these algorithms pull another node out of the collection and put the node's children into the collection. (Note that for a directed graph, the edge from a source node x to a target node y must point the "right way" in order for y to be added to the collection when x is visited.) For a breadth-first search, the collection is a queue, while for depth-first search, the collection is a stack. Since each node is touched twice (once on entering the collection and once when it is visited), and each edge is touched once, the algorithms run in $O(|V| + |E|)$ asymptotic algorithmic complexity.

During a depth-first search of a directed tree, four edge types (listed in the problem) could be encountered. The non-tree edges mainly result from encountering edges that were previously seen but not traversed because they pointed the "wrong way" when they were first encountered. On an undirected graph, edges can be traversed in either direction, so only tree edges and back edges appear in undirected graphs. Moreover, in acyclic graphs (either directed or undirected), back edges cannot appear.

The topological sort algorithm presented in [Cormen] uses a depth-first search. As nodes are finished, they are added to the front of a linked list. Since no node can be finished after its ancestors are, nodes will appear later in the final list than their ancestors. (Ancestors, meaning sources of incoming edges on a node x, are like "dependencies" of x.) Since depth-first search runs in $O(|V| + |E|)$, so does topological sort.

---

70. **Single-source shortest paths**

Dijkstra's algorithm starts by marking all vertices as infinitely far from the source vertex, then marks the source as being zero distance from itself. All vertices are then loaded into a priority queue and removed one at a time (in order of increasing distance from the source). As each vertex is removed, its edges are examined to see if any adjacent vertices are closer to the source than initially thought; this is called relaxation, and it may result in a new ordering of vertices within the priority queue. A Fibonacci heap is useful because it supports priority queue operations in amortized $O(\lg n)$ time. Dijkstra's algorithm removes $|V|$ vertices from the queue and processes all $|E|$ edges, so the asymptotic algorithmic complexity is $O(|E|+|V| \lg |V|)$. Dijkstra's algorithm cannot handle edges with negative weights because that could break the invariant that when a vertex leaves the priority queue, it is "done" (meaning that its true distance from the source has been discerned).

The Bellman-Ford algorithm also begins by marking all vertices as infinitely far from the source vertex and marks the source as being zero distance from itself. However, Bellman-Ford simply loops over all the vertices in the graph, and for each vertex, it loops over all the edges to see if the presence of that edge helps reduce the expected distance to the vertex (by providing a shortcut through that edge). Thus, the Bellman-Ford algorithm runs in $O(|E| * |V|)$ time, which is somewhat poor, but Bellman-Ford can readily handle negative-weight edges.

Prim's algorithm for finding a minimum spanning tree grows a single tree. At each step, it picks the cheapest edge that hooks another vertex into the tree. Prim's algorithm runs in $O(|E| + |V| \lg |V|)$ time if a Fibonacci heap is used for managing the priority queue.

71. **Minimal spanning tree**

Kruskal's algorithm for finding a minimum spanning tree builds a forest of trees. At each step, it picks the cheapest edge that joins two trees together. Hence, Kruskal's algorithm relies on keeping track of disjoint sets (with one set per tree that is in the forest). The asymptotic algorithmic complexity is indeed $O(|E| * \alpha(|V|))$ if these disjoint sets of vertexes are managed through a Union-Find data structure (which has operations that run in amortized $\alpha(n)$ time, essentially constant).

Prim's algorithm for finding a minimum spanning tree grows a single tree. At each step, it picks the cheapest edge that hooks another vertex into the tree. Prim's algorithm runs in $O(|E| + |V| \lg |V|)$ time if a Fibonacci heap is used for managing the priority queue. Note that if $|E|$ is a constant, then Prim's algorithm takes $O(|V| \lg |V|)$ time, whereas Kruskal's only takes $O(\alpha(|V|))$.

---

72. **All-pairs shortest paths**

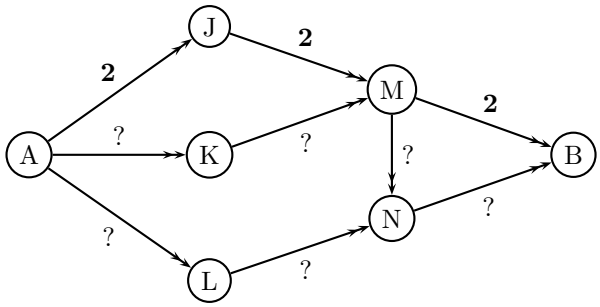
Johnson's algorithm solves the all-pairs shortest paths problem. It does so by running Bellman-Ford to check for negative weight cycles, and then repeatedly runs Dijkstra's algorithm to find the single-source shortest paths from each node. Repeatedly running Dijkstra's algorithm is the limiting factor in terms of asymptotic algorithmic complexity; if Dijkstra's algorithm is implemented with Fibonacci-heaps, then Johnson's algorithm can run in $O(|V|^2 \lg |V| + |V| * |E|)$. (Before running Dijkstra's algorithm, it shifts the weight of each edge, if needed, to eliminate negative weight edges.) Like Bellman-Ford and Dijkstra's algorithm, Johnson's algorithm relies on adjacency lists for the input graph.

The Floyd-Warshall algorithm is a dynamic programming approach to the all-pairs shortest paths problem. For each pair of nodes x and y, it computes the distance between x and y without going through any other intermediate nodes (so many nodes may at this point have an apparently infinite distance between them). Then, for each pair of nodes, it computes the distance between x and y if the intermediate path is allowed to traverse node 0. Then, for each pair, it recomputes the distance if the path is allowed to go through nodes 0 or 1. The algorithm continues adding nodes to the set of allowed intermediate nodes until all paths are allowed. The asymptotic algorithmic complexity is $O(|V|^3)$, if adjacency matrices are used, regardless of how many edges are actually present in the graph.

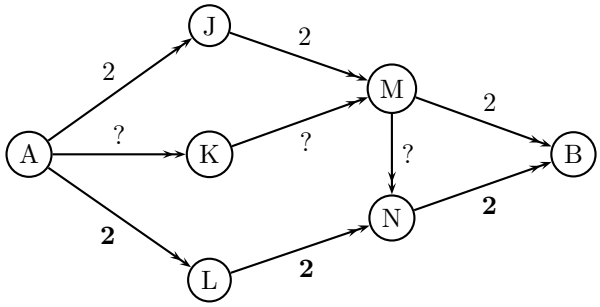
73. **Maximum network flow**

One way to find the maximum flow in a network with integral edge capacities is the Ford-Fulkerson method. This method examines all the paths from the source to the sink and identifies paths that are useful for augmenting the total flow through the network. There are a variety of different implementations that look a bit different graphically, but here is an illustrative walkthrough:

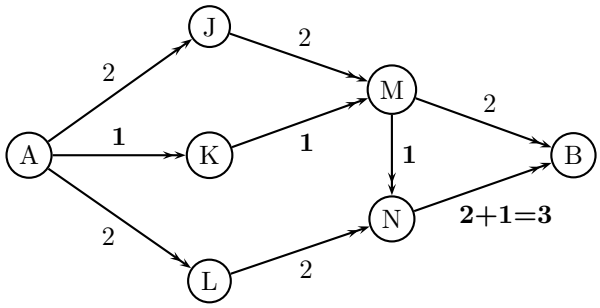
Consider $A \rightarrow J \rightarrow M \rightarrow B$, which has maximum capacity of 2 (because of the $A \rightarrow J$ and $M \rightarrow B$ links). After sending as much flow as possible through this channel, the flow looks like the following:



Next consider $A \rightarrow L \rightarrow N \rightarrow B$ with a maximum capacity of 2 (due to the $A \rightarrow L$ and $L \rightarrow N$ links):



Next consider $A \rightarrow K \rightarrow M \rightarrow N \rightarrow B$, with a maximum capacity of 1 (due to the $K \rightarrow M$ link):



It might be tempting to try augmenting by examining the path $A \rightarrow J \rightarrow M \rightarrow N \rightarrow B$, but the $A \rightarrow J$ link is already fully utilized. No additional flow-augmenting paths exist, so the maximum flow is 5.

74. **Dynamic programming examples**

Only Floyd-Warshall uses a dynamic programming approach. All of the others are greedy, since they make choices without considering sub-problems first.

---

75. **Strategies: greedy, dynamic programming, memoization, divide-and-conquer**

Divide-and-conquer problems first break the problem into sub-problems. They then solve each subproblem. Finally, they combine the sub-problem solutions into a solution to the main problem. Note that these sub-problems are treated as independent: although a given parent problem can use the results of its sub-problems, nothing that is learned while solving one sub-problem is used in the course of solving the other sub-problem(s). Many implementations depend on recursion.

Dynamic programming algorithms first compute the solutions to all possible sub-problems, and then use these solutions to construct the solutions to larger problems. Generally, when compared to a divide-and-conquer algorithm, a dynamic programming algorithm will demonstrate superior asymptotic algorithmic complexity if the sub-problems overlap a great deal.

Greedy programming algorithms simply choose the solution that looks best at each step. For example, Prim's minimal spanning tree algorithm extends the tree at each step by selecting the cheapest edge that hooks a new node into the tree.

Memoization is a way of blending the efficiency of dynamic programming with the top-down strategy of divide-and-conquer. The control flow looks like a divide-and-conquer algorithm, but the solution to subproblems are stored in memory. That way, the algorithm can check to see if it has already solved a certain sub-problem and reuse the answer rather than wasting time repeating work.

If only a few sub-problems need to be solved, then memoization is faster than a bottom-up dynamic programming approach. However, if every single sub-problem must be solved, it is generally faster to precompute the sub-problems' solutions rather than solve them on an as-needed basis. The reason is that most memoization and divide-and-conquer implementations make use of recursion, which can be slow; there may also be benefits due to less virtual memory page swapping in dynamic programming.

---

76. **Strategies (2): greedy, dynamic programming, memoization, divide-and-conquer**

Refer to the comments in the previous problem to understand why choices A, B, D, and E *are all true.*

Matroids constitute an elegant theory for representing most —though not all—problems for which a greedy solution exists. (The activity scheduling problem is a notable exception, so C *is not true.*) A matroid is a pair (S, Γ), such that S is a set and Γ is a collection of subsets of S. (That is, Γ is a set of sets.) If set $A \in \Gamma$, and $B \subseteq A$, then $B \in \Gamma$. Also, if $A \in \Gamma$ and $B \in \Gamma$, such that $|A| > |B|$, then there exists an $x \in A - B$ such that $B \cup \{x\} \in \Gamma$.

If a positive weight function is defined on every element of S, then finding the element of Γ with the maximum total weight is a problem that can be solved with a greedy algorithm. This algorithm builds up the optimal $M \in \Gamma$ in a greedy fashion from scratch, rather than constructing each possible $A \in \Gamma$, computing its weight, and then picking the biggest. In fact, this greedy algorithm can be reused on any problem that can be put into a matroid form.

Mathematics and Theory — Comments

77. **Prefix, infix, and postfix**

Choice A evaluates to `( (2 - 3) * 5) + 7 = (-1 * 5) + 7 = -5 + 7 = 2`
Choice B evaluates to `2 + (3 * 5) - 7 = 2 + 15 - 7 = 17 - 7 = 10`
Choice C evaluates to `(2 + 3) * (5 - 7) = 5 * (-2) = -10`
Choice D evaluates to `(2 + 3) * (5 - 7) = 5 * (-2) = -10`
Choice E evaluates to `((2 + 3) * 5) - 7 = (5 * 5) - 7 = 25 - 7 = 18`

So the largest of the five expressions is choice E.

---

78. **Twos-complement**

The first choice (0401) equals 1025 in decimal, while the second (3840) equals 14400 in decimal. Both are return codes indicating success. The other three choices indicate negative numbers.

To negate a number, flip the bits and add 1. Examples:

> 1 in decimal is 0001, flipping the bits gives FFFE, and adding 1 yields −1 as FFFF
>
> 10 in decimal is 000A, flipping the bits gives FFF5, and adding 1 yields −10 as FFF6

Hence, FFF6 and FFFD fall into the range −1 through −10 and indicate valid error return values. The only choice that could not be returned by this function is choice C, which corresponds to −19 in decimal.

---

79. **Arithmetic overflow**

Overflow occurs in twos-complement when two numbers x and y of the same sign are added, but the result has the opposite sign.

Choice A equals 6990 in hexadecimal, which has the same sign as the operands.
Choice B equals FFEC in hexadecimal, which has the same sign as the operands.
Choice C cannot overflow, since the operands have different signs.
Choice D equals 1111 in hexadecimal, which has the same sign as the operands.
Choice E equals 686D in hexadecimal, which has a different sign than the operands.

---

80. **Floating point (IEEE-754)**

One bit is required for the sign. A bias-127 number requires 8 bits for the exponent. That leaves 23 bits for the mantissa.

The first bit is a 1, indicating that the sign is set. This is a negative number.

The exponent's bit sequence 00011111 corresponds to 31, which is biased by −127, so the exponent is −96.

The mantissa's bit sequence 00000000001000000000000 has a single bit set 11 places from the left. This is the fractional component of a number with an implied unit bit to the left. So the mantissa is $1 + 2^{-11}$.

In other words, the entire number equals $-1 * 2^{-96} * (1 + 2^{-11})$.

Note that in IEEE-754, certain combinations of bits are set aside to represent special cases, such as infinity and NaN ("not a number").

---

81. **Limitations of finite precision**

I *is false* because of round off error. II *is false* because negative numbers have a leading 1 bit in two's complement, so the left should be filled with a 1 if the number is negative and a 0 if it is positive.

To compute the one's complement representation of a negative number, write the number as a positive number but then flip the bits. Transforming this to two's complement then requires adding 1. In general, this means that the one's and two's complement representations of a negative number differ. However, for positive integers, the one's and two's complement representations are the same, so III *is false*.

---

82. **XOR**

Examining the truth table of XOR will reveal that I *is true*. This is a very good formula to memorize.

The XOR operator is commutative, so II *is true*, as the truth table will reveal.

III *is not true*. For example, if $a = 0, b = 1, c = 1$, then $(a + \overline{b}) * (b + c)$ equals 0, but $a + c$ equals 1.

---

83. **Implication**

I *is true* thanks to a quirky little convention. The "$\rightarrow$ " symbol is read as "implies." The only time $a \rightarrow b$ is not true is if a *is true* but b *is not*. This matches the truth table for $\overline{a} + b$.

II *is true*, as revealed by a truth table: $(\overline{a} + b) + (a * \overline{b})$ is always true, regardless of the values of a and b.

A more intuitive way to see why this is true is to recognize that the first term $\overline{a} + b$ equals $a \rightarrow b$, and the second term $(a * \overline{b})$ equals the negation of $a \rightarrow b$. Since it is always true that $a \rightarrow b$ is either true or false, it stands to reason that $(\overline{a} + b) + (a * \overline{b})$ *is also always true*.

III *is false*, since $(a \rightarrow b) * (a * \overline{b})$ is equivalent to asking for $a \rightarrow b$ to be true and false at the same time.

84. **De Morgan's laws**

Note that $X - Y = X \cap \overline{Y}$. De Morgan's laws will also be useful:

$$\overline{X \cup Y} = \overline{X} \cap \overline{Y} \qquad \overline{X \cap Y} = \overline{X} \cup \overline{Y}$$

Consider proposition I, which *is false*:

$$A \cup (B - C) = A \cup (B \cap \overline{C})$$

But

$$(A \cup B) - (A \cup C) = (A \cup B) \cap \overline{A \cup C} = (A \cup B) \cap (\overline{A} \cap \overline{C}) = B \cap \overline{A} \cap \overline{C}$$

Consider proposition II, which *is true*:

$$A \cap (B - C) = A \cap (B \cap \overline{C})$$

And

$$(A \cap B) - (A \cap C) = (A \cap B) \cap (\overline{A} \cup \overline{C}) = A \cap B \cap \overline{C}$$

Consider proposition III, which *is true*:

$$A - (B \cap C) = A \cap \overline{B \cap C} = A \cap (\overline{B} \cup \overline{C}) = (A \cap \overline{B}) \cup (A \cap \overline{C})$$

And

$$(A - B) \cup (A - C) = (A \cap \overline{B}) \cup (A \cap \overline{C})$$

---

85. **Binary functions**

Choice A is definitely possible. For example, let $M = \{1, 2\}$, $g(1, 1) = g(2, 2) =$ `true`, and $g(1, 2) = g(2, 1) =$ `false`. Note that for any x and y, if $g(x, y) =$ `true`, then $g(y, x) =$ `true`, thus meeting the definition of symmetric. Note also that for any x and y, if $g(x, y) = g(y, x) =$ `true`, then $x = y$, thus meeting the definition of antisymmetric.

Choice B is not possible. To prove this by contradiction, select an x and y from different equivalence classes. Since g is a total order, either $g(x, y) =$ `true` or $g(y, x) =$ `true`. Since g is an equivalence relation, which is symmetric, it follows that both $g(x, y) =$ `true` and $g(y, x) =$ `true`. Since g is a total order, which is antisymmetric, it follows that $x = y$. But then x and y would be in the same equivalence class, violating the initial selection criteria.

Choice C is not possible, since a total order must be a partial order (meaning reflexive, antisymmetric, and transitive), and also total (either $g(x, y)$ or $g(y, x)$ must be true for all x and y).

Choice D is not possible, since set M contains at least two elements. The reasoning is as follows. Reflexivity implies that for all x, $g(x, x) =$ `true`. Antisymmetry implies that for any x and y such that $x \neq y$, $g(x, y) =$ `false` or $g(y, x) =$ `false`. Hence, both `true` and `false` are in the range of g, which implies that g must be a surjection, meaning that for any value z in the co-domain {`true, false`}, there exists some (x, y) in the domain $M \times M$ such that $g(x, y) = z$.

Choice E is not possible, since the domain of g contains a perfect square number of elements (such as 4, 16, 36, ...). But the range contains only two elements (`true` and `false`). Hence, the domain is larger than the range in size (and both are finite), so g cannot be an injection.

Incidentally, this implies that g cannot be a bijection, either, which is a function that is both an injection and a surjection.

---

86. **Big-O and friends**

Choices A through C are possible, for example if f and g are the same function.

Choice D is not possible. If $f \in O(g)$, then there exists some constants N and $C > 0$ such that for all $n > N$, $f(n) \leq C * g(n)$. But by definition, this implies that $g \in \Omega(f)$.

Choice E is possible, but hard to imagine in practice. Such a pair of f and g functions must "take turns" being greater than one another. There can be no constants N and $C > 0$ such that for all $n > N$, $f(n) \leq C * g(n)$; conversely, there can be no constants N and $C > 0$ such that for all $n > N$, $g(n) \leq C * f(n)$. For example, let $g(n) = n$, and let $f(n) = n^2$ when n is composite but $f(n) = 1$ when n is prime.

---

87. **The guess-that-sum game**

First, note that $f(n)$ roughly equals $\ln(n)$ for large n, though it slightly exceeds $\ln(n)$ for finite n (but never by more than a factor of 3 for $n > 1$). Second, $g(n)$ exactly equals $\frac{e^n - 1}{e - 1}$. Thus, for large n, $f(n)$ is approximately proportional to $\ln(\ln(g))$.

Note that if $h(x) = \ln(x)$, then $h(x) \in \Theta(\log_D x)$ for any constant base D.

---

88. **The guess-that-recurrence game**

Choices A and E can be analyzed using the Master's Theorem. Given recursion $T(n) = a * T(n/b) + g(n)$, let $r = log_b a$ and $a, b > 0$.

- If $g(n) \in O(n^{r-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) \in \Theta(n^r)$.
- If $g(n) \in \Theta(n^r)$, then $T(n) \in \Theta(n^r * log_2 n)$
- If $g(n) \in \Omega(n^{r+\varepsilon})$ for some $\varepsilon > 0$, and $a * g(n)/b < f(n)$ for large n, then $T(n) \in \Theta(g(n))$.

Choice A: Covered the middle case of the Master's Theorem.
Choice E: Covered by the first case of the Master's Theorem.
Choice B: Verify by substitution that $f(n) = c * \lfloor n/2 \rfloor + 1$.
Choice D: Verify by substitution that $f(n) = c^{\lfloor n/2 \rfloor}$.
Choice C: Actually, $f(n) \in O(n^{c+1})$ but not $O(n^c)$.

89. **Ackermann's function**

This function is called Ackermann's function, which is one of the fastest-growing functions ever imagined. It turns out that it does have some application (in its inverse form) in the analysis of the Union-Find data structure and algorithm (see question 71).

The correct values for $g(1,n), g(2,n)$, and $g(3,n)$ are probably worth memorizing...

$$g(1,n) = n+2$$

$$g(2,n) = 2n+3$$

$$g(3,n) = 2^{n+3} - 3$$

This can be verified with the following chart:

	$x=0$	$x=1$	$x=2$	$x=3$
$Y=0$	1	2	3	5
$Y=1$	2	3	5	13
$Y=2$	3	4	7	29
$Y=3$	4	5	9	61

---

90. **Determinants**

Choice A *is always true*, provided A is not singular. Choice B *is true*: $det(x * A) = x^n * det(A)$, where n is the number of columns in the matrix.

Choice D *is correct*, since multiplying a single row by 2 doubles the determinant. Likewise, choice E *is true*, since adding a multiple of one row to another does not change the determinant.

However, choice C is wrong because swapping two rows multiplies the determinant by -1.

---

91. **Transposes and inverses**

These and other important but basic properties of matrices are summarized in [Rothenberg] and similar introductory linear algebra texts.

---

92. **Trees and graphs**

Note that this question asks for the option that is *not* true *if and only if* the graph is a tree.

The brief proofs for these appear in [Cormen]. The basic idea is that a tree is the most efficient way to connect all the elements of a graph in the sense that adding additional edges will result in a cycle.

Note that all the nodes in a clique are directly connected to one another. So if there are $|V|$ nodes, then there would need to be $|V|^2$ edges in a directed graph—that certainly does not look like a tree!

93. **Graph properties**

Choice A *is false*. A clique is sub-graph $H \subseteq G$ such that for any vertices u and v in H, u and v are directly connected. The biggest clique in G is of size 2.

Choice B *is true*. A graph is strongly connected if for any vertices u and v in G, there exists a path from u to v.

Choice C *is false*. A Hamiltonian circuit is a path that visits every vertex, then ends on the vertex where it started. Determining whether a graph has a Hamiltonian circuit is difficult in general, but here it is easy because the central node in G must be visited at least one extra time because it adjoins two vertices that have only one edge each.

Choice D *is false*. An Eulerian circuit is a path that visits every edge, then ends on the vertex where it started. A graph has an Eulerian circuit *if and only if* it is connected and every vertex has an even number of edges, which is not the case for G.

Choice E *is false*. A graph is complete by definition if for any vertices u and v in G, u and v are connected.

---

94. **Turing-incompleteness and halting**

Choice A *is false*, since it is impossible to create programmatic loops using this language.

Choice B *is almost true*. It falls apart because if the program accepts console inputs during execution, then the user could take an arbitrarily long amount of time to provide an input.

Choice C *is true*. As each instruction executes, the program counter must move toward the end of the program (since backward branches are forbidden). Consequently, if the program contains p instructions, then no more than p instructions can be executed before the program counter falls off the end of the program.

Choice D *is false*. Imagine a program with p total instructions that needs x instructions to compute each Fibonacci number. Then all the user has to do is enter $q = 1 + (p/x)$ to break the system. The program must finish within p instructions, so it only has time to compute p/x Fibonacci numbers. (In addition, any implementation of such a program must have at least one loop, since q is arbitrarily large; however, as mentioned for choice A, this language does not support looping.)

Choice E *is false*. Imagine a program with p total instructions that uses x instructions to check whether the next character is consistent with a given regular expression. The argument proceeds as in choice D.

95. **Regular languages**

Choices A and C are popular "well-known" non-regular languages. One straightforward way to show that a language is non-regular is to use the pumping lemma. This lemma says that if L is a regular language, there exists a positive integer p such that for any string $w \in L$, there exist strings x, y, and z such that $w = xyz$ and $|x| < p$ and $|y| > 0$ and $xy^n z \in L$ for any non-negative integer n. The procedure is to suggest some string w that is in L and then to argue that no strings x, y, and z exist that satisfy these requirements.

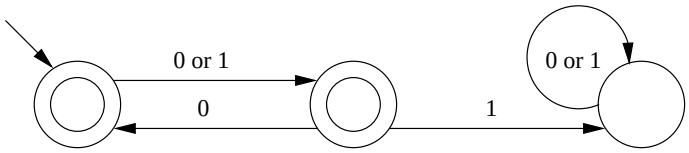
For choice A, use $w = 0^p 1^p$.

For choice B, use $w = 1^r$ where r is the first prime number such that the next prime number exceeds $p + r$.

For choice C, use $w = 0^p 110^p$.

For choice E, use $w =$ the element in L which contains 1 in exactly p positions.

In contrast, choice D can be recognized with the following deterministic finite state automaton:



---

96. **Elimination of left recursion**

In G1, `T` can take the form of `v * v * v * v * v ....` In G2, `R` can take the form `* v * v * v ...`, so `T` can take the form of `v * v * v * v ....` Thus, in both G1 and G2, `E` can take the form `v * v * v + v * v * v * v + v * v ....` These are the algebraic expressions involving only addition and multiplication.

In G3, `T` can take the form `v * v * v * v`, so `R` can take the form `+ v * v + v * v`. Thus, through the production `E` $\rightarrow$ `T R`, `E` can take the form of any arithmetic expression. However, `E` can also take the form of non-arithmetic expressions, as with `E` $\rightarrow$ `R` $\rightarrow$ `+ T R` $\rightarrow$ `+ v R` $\rightarrow$ `+ v + T R` $\rightarrow$ `+ v + v R` $\rightarrow$ `+ v + v`. In addition, G3 can generate ε, which G1 and G2 cannot.

Thus, while G1 and G2 generate the same language, G3 generates a somewhat larger language.

Incidentally, G1 demonstrates left-recursion because it contains productions of the form $X \rightarrow X\alpha$ (in addition to non-problematic productions of the form $X \rightarrow \beta$). Left recursion can be eliminated by introducing a helper non-terminal `R` and changing the grammar to read

$X \rightarrow \beta R$

$R \rightarrow \alpha R \mid \varepsilon$

97. **Chomsky normal form**

As discussed by [Sipser], a context-free grammar has Chomsky normal form if every rule has either the form `A` $\rightarrow$ `B C`, or `A` $\rightarrow$ `a`, where `A`, `B`, and `C` are non-terminals and `a` is a terminal. Also, `S` $\rightarrow \varepsilon$ is allowed if `S` is the start symbol. Grammar G2 has Chomsky normal form. When a grammar in Chomsky normal form generates a non-empty string of length n, then $2 * n - 1$ steps are required. In contrast, G1 would require only n steps.

Neither grammar is ambiguous. In general, proving this is tough. However, proving it in the case of G2 is made easier by the fact that the left hand variable in each production can be substituted into the ones above it to produce a grammar with a single production, `S` $\rightarrow$ `0 1 S` (along with `S` $\rightarrow \varepsilon$), which is clearly not ambiguous.

---

98. **Decidability**

Rice's theorem shows that it is impossible to examine a Turing machine and decide anything nontrivial about that machine. That is, for any language L and a serialized representation $\langle M \rangle$ of a machine M, it is impossible to decide whether $\langle M \rangle \in L$ unless (a) all machine representations are in L, or (b) no machine representations are in L, or (c) the membership of $\langle M \rangle$ in L depends on something other than the language that M recognizes (for example, the length of $\langle M \rangle$). With this theorem in hand, it is clear that choices D and E are *not decidable*.

Choices B and C are not decidable because a context-free grammar can be used to represent the allowable configuration histories of a Turing machine as it computes, and predicate expressions can be used to represent the language that a Turing machine recognizes. Consequently, if either of these were decidable, it would be feasible to decide whether a Turing machine will halt for a given input, which is a problem that is known to be undecidable. Therefore, neither choice B nor choice C is decidable.

Choice A, on the other hand, is decidable. Given two nondeterministic finite automata, compute the equivalent deterministic finite automata, A and B, which recognize languages L_A and L_B, respectively. Next, build a nondeterministic finite automaton that decides the language $L = (L_A - L_B) \cup (L_B - L_A)$. Note that L is empty *if and only if* A and B are equivalent. Convert that automaton to a deterministic finite automaton M_L. Finally, check to see if any accept states in M_L can be reached from the start state to determine whether L is empty.

---

99. **Chomsky hierarchy**

Here is the Chomsky hierarchy of language classes and the corresponding machines that decide them.

Regular languages are the simplest. L is regular *if and only if* there exists a deterministic finite state automaton M that decides L. Also, L is regular *if and only if* there exists a nondeterministic finite state automaton M that decides L. Finally, L is regular *if and only if* there exists a regular expression E such that E generates a string w *if and only if* w is an element of L.

All regular languages are context free. L is context free *if and only if* there exists a nondeterministic pushdown automaton M that decides L. (Note: The class of languages decided by deterministic pushdown automata is a strict subset of the class of context free languages.) L is context free *if and only if* there exists a context free grammar G such that G generates a string w *if and only if* w is an element of L.

All context free languages are recursive. L is recursive *if and only if* there exists a Turing machine M that decides L. (It appears that the decidable languages are not technically a member of the Chomsky hierarchy, but they fit in there nicely.)

All recursive languages are recursively enumerable. L is recursively enumerable *if and only if* there exists a Turing machine M that recognizes L.

Choice E *is not true* because if L is recognized by Turing machine M, then this only implies that L is recursively enumerable. It does not imply that L is recursive, since M might not be able to decide L.

---

100. **Closure properties**

Choice B *is not true*, since the class of context-free languages is not closed under intersection. Most of the other closure issues can be addressed with a fairly straightforward construction. Here are a few that might not be obvious:

- Intersection between two regular languages can be achieved by representing each language as a deterministic finite-state automaton with state sets D1 and D2, and then constructing a new automaton with state set D1 $\times$ D2. This new automaton essentially runs the first two automata on the input string and makes sure that they both accept.
- Concatenation of two languages in $\mathcal{P}$ can be achieved by letting a machine for the first language run on the first i characters of the n input characters, and then letting a machine for the second language run on the last $n - i$ characters of the input. Both machines must halt in polynomial time. If either fails to accept, then increment i and try again. By looping from $i = 0$ through $i = n$, the concatenated language can be decided.
- Kleene star of a decidable (recursive) language L can be achieved by counting the number of characters n in the input, then looping from $i = 1$ through $i = n$. If i does not evenly divide n, then continue to the next value of i. Second, if i does evenly divide n, but the input is not equal to n/i repetitions of the first i characters, then continue to the next value of i. Thirdly, if a Turing machine for L does not accept the first i characters, then continue to the next value of i. But if all three conditions are met for a certain i, then accept the input.
- Concatenation of two recognizable (recursively enumerable) languages L1 and L2 proceeds as with two languages in $\mathcal{P}$ (choice C above), with one caveat. Since the machines for L1 and L2 might not terminate, it is necessary to run copies of the machines in parallel for all possible values of i. Fortunately, the input is finite in length, so a finite number of i values is possible ($i = 0$ through $i = n$).

101. **$\mathcal{NP}$-completeness**

The $\mathcal{NP}$ problems have solutions that can be checked in polynomial time. (Such a language can be recognized by a non-deterministic Turing machine in polynomial time.) $\mathcal{NP}$ complete problems have the additional property that they are $\mathcal{NP}$-hard; that is, any other $\mathcal{NP}$ problem is reducible to the $\mathcal{NP}$ complete problem in polynomial time. Intuitively, $\mathcal{NP}$ complete problems are "at least as hard" as any other $\mathcal{NP}$ problem.

Of the problems listed here, only checking for an Eulerian circuit is not $\mathcal{NP}$ complete. (An Eulerian circuit is a path that visits each edge exactly once, then terminates on the vertex where it started.) In fact, checking whether an undirected graph contains an Eulerian circuit is in $\mathcal{P}$, since an undirected graph has an Eulerian circuit *if and only if* the graph is connected and every node adjoins an even number of edges.

Perhaps the most famous $\mathcal{NP}$ complete problem is the "traveling salesman" problem, which is described by choice E.

---

102. **Reducibility**

Keep in mind that if A can be reduced to B, then B is "at least as hard" as A. See question 99 for the hierarchy of languages.

Choices A, B, and C *are false*. Suppose that A is decidable on a nondeterministic Turing machine in polynomial time ($\mathcal{NP}$). This does not imply that B is also in $\mathcal{NP}$ (or $\mathcal{P}$ or $\mathcal{NP}$-hard). For example, if A = traveling salesman problem, then A is reducible in constant time to B = traveling salesman problem plus also decide if two Turing machines recognize the same language. But B is now not decidable, so it is not in $\mathcal{NP}$ (nor in $\mathcal{P}$ or $\mathcal{NP}$-hard).

Choice D *is false*. If B is $\mathcal{NP}$-complete, then that at least implies that A must be $\mathcal{NP}$, since if B is decidable on a nondeterministic Turing machine in polynomial time ($\mathcal{NP}$), so must A. However, there is no reason to believe that A is in $\mathcal{P}$. For example, if A is the traveling salesman problem and B is the same problem (reduction by identity), then B is $\mathcal{NP}$-complete but A is not in $\mathcal{P}$.

Choice E *is true*. If B can be decided in polynomial time with a deterministic Turing machine, and if A can be reduced to B in polynomial time, then A can also be decided in polynomial time with a deterministic Turing machine.

Resources

Practice questions and practice tests

You can never have too many practice problems, so you may want to consider borrowing or buying two additional resources to accompany this booklet.

The first is *GRE: Practicing to Take the Computer Science Test* (published by ETS), which contains approximately 100 practice problems plus a full-length test with real questions. Unfortunately, it seems to be in short supply and was recently running well over $50 online (used).

The other good supplementary resource may be *GRE Computer Science - The Best Test Preparation for the Graduate Record Examination Subject Test* (published by REA), though I have not yet seen a copy, since it was published in 2005. People generally criticized the previous version by REA as very unlike the actual computer science subject test. However, the current version is written by a professor who has taught courses that prepare students for their upcoming exam. At around $20, it might be a good investment.

Educational textbooks

Some of these are pretty ~~ancient~~ timeless and are available in several editions.

Cormen, T., et al. *Introduction to Algorithms.*
Patterson, D., and Hennessy, J. *Computer Architecture: A Quantitative Approach.*
Patterson, D., and Hennessy, J. *Computer Organization & Design: The Hardware / Software Interface.*
Rothenberg, R. *Linear Algebra with Computer Applications.*
Sedgewick, R. *Algorithms.*
Silberschatz, A., and Galvin, P. *Operating System Concepts.*
Sinha, P. *Distributed Operating Systems: Concepts and Design.*
Sipser, M. *Introduction to the Theory of Computation.*
Stallings, W. *Computer Organization and Architecture: Designing for Performance.*
Tanenbaum, A. *Operating Systems Design and Implementation.*

Index

Note that the index entries below indicate question numbers rather than page numbers.

ISBN 0-9727324-4-6