Níže uvedené úlohy představují přehled otázek, které se vyskytly v tomto nebo v minulých semestrech ve cvičení nebo v minulých semestrech u zkoušky. Mezi otázkami semestrovými a zkouškovými není žádný rozdíl, předpokládáme, že připravený posluchač dokáže zdárně zodpovědět většinu z nich.

Tento dokument je k dispozici ve variantě převážně s řešením a bez řešení.

Je to pracovní dokument a nebyl soustavně redigován, tým ALG neručí za překlepy a jazykové prohřešky, většina odpovědí a řešení je ale pravděpodobně správně :-).

---------- RECURSION ----------

1.
Určete, jakou hodnotu vypíše program po vykonání příkazu **`print(rekur(6));`**, když rekurzivní funkce **`rekur()`** je definována takto:

```
int rekur(int x) {
  if (x < 1) return 2;
  return (rekur(x-3)+rekur(x-4));
}
```

a) 6
b) 7
c) 8
d) 14
e) 16

2.
Určete, jakou hodnotu vypíše program po vykonání příkazu **`print(rekur(5));`**, když rekurzivní funkce **`rekur()`** je definována takto:

```
int rekur(int x) {
  if (x < 0) return 2;
  return (rekur(x-4)+rekur(x-2));
}
```

a) 5
b) 6
c) 8
d) 10
e) 16

3.

```
int fff (int x, int y) {
  if (x <= y) return x;
  return fff(y,x);
}
```

Uvedená funkce fff vrátí pro kladné hodnoty x a y:

a) max(x,y)
b) min(x,y)
c) vždy x
d) vždy y
e) nevrátí nic, bude volat stále sama sebe

4.

```
int ff(int x, int y) {
  if (x > 0) return ff(x-1, y)-1;
  return y;
}
```

Funkce ff provádí následující akci:

a) pro kladná x vrací 0, jinak vrací y
b) odečte x od y, pokud x je nezáporné
c) odečte y od x, pokud x je nezáporné
d) vrací –y pro kladné x, jinak vrací y
e) spočte zbytek po celočíselném dělení y%x

5.

```
int ggg(int x, int y) {
 if (x <= y) return y;
 return ggg(y,x);
}
```

Uvedená funkce ggg vrátí pro kladné hodnoty x a y:

a) max(x,y)
b) min(x,y)
c) vždy x
d) vždy y
e) nevrátí nic, bude volat stále sama sebe

6.

Určete, jakou hodnotu vypíše program po vykonání příkazu **`print(rekur(5));`**, když rekurzivní funkce **`rekur()`** je definována takto:

```
int rekur(int x) {
  if (x <= 0) return 1;
  return (rekur(x-2)+rekur(x-2));
}
```

a) 4
b) 7
c) 8
d) 15
e) 16

7.

Determine what value will be printed as result of the function call **`print(recur(4));`**, Recursive function **`recur()`** is defined as follows:

```
int recur(int x) {
  if (x < 0) return 1;
  return (recur(x-2)+recur(x-2));
}
```

a) 4
b) 6
c) 8
d) 16
e) 32

8.
Určete, jakou hodnotu vypíše program po vykonání příkazu **`print(rekur(2));`**, když rekurzivní funkce **`rekur()`** je definována takto:

```
int rekur(int x) {
  if (x < 0) return 1;
  return (rekur(x-2) + rekur(x-1));
}
```

a) 2
b) 3
c) 4
d) 5
e) 8

9.
The call of the function **`recur(2)`** produces the sequence:

```
void recur(int x) {
  if (x < 0) return;
  print(x);
  recur(x-1);
  recur(x-1);
}
```

a) 1 1 2
b) 2 1 1 0 0
c) 0 0 1 0 0 1 2
d) 0 1 0 2 0 1 0
e) 2 1 0 0 1 0 0

10.
The call of the function **`recur(2)`** produces the sequence:

```
void recur(int x) {
  if (x < 0) return;
  recur(x-1);
  recur(x-1);
  print(x);
}
```

a) 1 1 2
b) 2 1 1 0 0
c) 0 0 1 0 0 1 2
d) 0 1 0 2 0 1 0
e) 2 1 0 0 1 0 0

11.
Determine the exact number of calls of the **`xyz()`** function while perfoming the command **`print(recur(2));`**. Recursive function **`recur()`** is defined as follows:

```
int recur(int x) {
  if (x < 1) return 2;
  xyz();
  return (recur(x-1)+recur(x-2));
}
```

a) 2
b) 3
c) 5
d) 6
e) 8

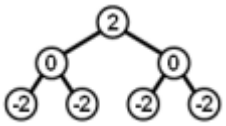
12.

```
void ff(int x) {
  if (x >= 0) ff(x-2) ;
  abc(x);
  if (x >= 0) ff(x-2) ;
}
```

Daná funkce ff je volána s parametrem 2: **ff(2);**. Funkce **abc(x)** je tedy celkem volána

a) 1 krát
b) 3 krát
c) 5 krát
d) 7 krát
e) 8 krát

Jedno volání funkce reprezentujeme uzlem ve stromu rekurzivního volání, hodnotou uzlu bude hodnota parametru funkce v daném volání.



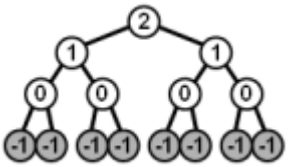
Protože při každém volání funkce ff se zavolá funkce abc() právě jednou, je počet volání funkce abc() právě 7. Platí varianta d).

13.

```
void gg(int x) {
  if (x < 0) return;
  abc(x);
  gg(x-1);
  gg(x-1);
}
```

Daná funkce gg je volána s parametrem 2: **`gg(2);`**. Funkce **`abc(x)`** je tedy celkem volána

a) 1 krát
b) 3 krát
c) 4 krát
d) 7 krát
e) 8 krát



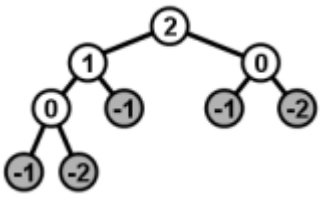
Strom rekurzivního volání funkce gg vidíme na obrázku (jenž není součástí zadání úlohy), v každém uzlu je vepsána hodnota parametru x při odpovídajícím volání funkce gg. Při volání, kdy je hodnota x = –1, nastává okamžitý návrat z funkce gg a funkce abc v takovém případě již volána není. To znamená že funkce gg bude volána jen v bílých uzlech stromu na obrázku, jichž je dohromady 7. Platí varianta d).

14.

```
 void fff(int x) {
  if (x < 0) return;
  abc(x);
  fff(x-1);
  fff(x-2);
}
```

Daná funkce fff je volána s parametrem 2: **`fff(2);`**. Funkce **`abc(x)`** je tedy celkem volána

a) 1 krát
b) 3 krát
c) 4 krát
d) 7 krát
e) 8 krát



Strom rekurzivního volání funkce fff vidíme na obrázku (jenž není součástí zadání úlohy), v každém uzlu je vepsána hodnota parametru x při odpovídajícím volání funkce fff. Při volání, kdy je hodnota x je menší než 0 (x = –1 nebo x = –2) , nastává okamžitý návrat z funkce fff a funkce abc v takovém případě již volána není. To znamená že funkce fff bude volána jen v bílých uzlech stromu na obrázku, jež jsou dohromady 4. Platí varianta c).

15.
Vypočtěte, kolik celkem času zabere jedno zavolání funkce **`rekur(4);`** za předpokladu, že provedení přikazu **`xyz();`** trvá vždy jednu milisekundu a že dobu trvání všech ostatních akcí zanedbáme.

```
void rekur(int x) {
  if (x < 1) return;
  rekur(x-1);
  xyz();
  rekur(x-1);
}
```

Řešení: Strom rekurzivního volání funkce **`rekur`** je binární vyvážený strom. při volání **rekur(4)** bude mít tento strom hloubku 5, uzly posledního (=nejhlubšího) „patra“ však budou odpovídat pouze provedení řádku **`if (x < 1) return;`** a příkaz **`xyz()`** se tu neprovede.
V každém uzlu stromu rekurzivního volání do hloubky 4 bude tedy jednou proveden příkaz **`xyz()`**. Počet těchto uzlů je 1 + 2 + 4 + 8 = 15. Stejněkrát bude proveden i příkaz **`xyz()`.**

16.
Určete, jakou hodnotu vypíše program po vykonání příkazu **`print(rekur(4));`**, když rekurzivní funkce **`rekur()`** je definována takto:

```
int rekur(int x) {
  if (x < 1) return 2;
  return (rekur(x-1)+rekur(x-1));
}
```

Nedokážete-li výsledek přímo zapsat jako přirozené číslo, stačí jednoduchý výraz pro jeho výpočet.

Řešení: Rekurzivní volání **`rekur(x-1)+rekur(x-1)`** můžeme zapsat jako **`2*rekur(x-1)`** a potom máme:

```
rekur(4) = 2*rekur(3) = 2*2*rekur(2) = 2*2*2*rekur(1) =
= 2*2*2*2*rekur(0) = 2*2*2*2*2 = 32.
```

17.
Funkce

```
int ff(int x, int y) {
 if (x > 0) return ff(x-1,y)+y;
 return 0;
}
```

a) sčítá dvě libovolná celá čísla
b) násobí dvě libovolná celá čísla
c) násobí dvě celá čísla, pokud je první nezáporné
d) vrací nulu za všech okolností
e) vrací nulu nebo y podle toho, zda x je kladné nebo ne

Řešení: Při každém návratu z funkce (kromě prvního s nulou) je vrácena hodnota předchozího volání zvětšená o hodnotu y, která se během jednotlivých volání nemění. Celkem je tedy vrácen vícenásobný součet hodnoty y, takže se jedná o násobení. Jediná podmínka ve funkci zároveň také určuje, že násobení proběhne pouze při nezáporném prvním parametru, takže správná je varianta c).

18.
Funkce

```
int ff(int x, int y) {
 if (x > 0) return ff(x-1, y)-1;
 return y;
}
```

a) pro kladná x vrací 0, jinak vrací y
b) odečte x od y, pokud x je nezáporné
c) odečte y od x, pokud x je nezáporné
d) vrací –y pro kladné x, jinak vrací y
e) spočte zbytek po celočíselném dělení y%x

19.
Funkce
```
int ff(int x, int y) {
 if (x < y) return ff(x+1,y);
 return x;
}
```

a) buď hned vrátí první parametr nebo jen „do nekonečna“ volá sama sebe
b) vrátí x+1
c) vrátí součet svých parametrů
d) vrátí maximální hodnotu z obou parametrů
e) neprovede ani jednu z předchozích možností

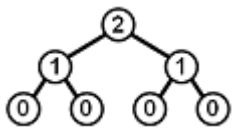
20.
Funkce
```
int ff(int x, int y) {
 if (y>0) return ff(x, y-1)+1;
 return x;
}
```

a) sečte x a y, je-li y nezáporné
b) pro kladná y vrátí y, jinak vrátí x
c) spočte rozdíl x–y, je-li y nezáporné
d) spočte rozdíl y–x, je-li y nezáporné
e) vrátí hodnotu svého většího parametru

Řešení: Pro y záporné nebo nulové vrátí fukce hodnotu x, Pro kladné y volá sama sebe. Počet volání je roven hodnotě y (neboť tento parametr se při každém volání o jedničku zmenší) a při návratu z rekurze se návratová hodnota z většuje pkaždé o 1. Počet volání je y, nejvnitřnější volání vrátí hodnotu x, takže návrat z rekurze přičte k x ještě hodnotu y. Správná odpověď je a).

21.
```
void ff(int x) {
  if (x > 0) ff(x–1) ;
  abc(x);
  if (x > 0) ff(x–1) ;
}
```
Daná funkce ff je volána s parametrem 2: **`ff(2);`**. Funkce **`abc(x)`** je tedy celkem volána
a) 1 krát
b) 3 krát
c) 5 krát
d) 7 krát
e) 8 krát

Řešení: Jedno volání funkce reprezentujeme uzlem ve stromu rekurzivního volání, hodnotou uzlu bude hodnota parametru funkce v daném volání.
Protože při každém volání funkce **`ff`** se zavolá funkce **`abc()`** právě jednou, je počet volání funkce **`abc()`** právě 7. Platí varianta d).



22.
Funkce

```
int ff(int x, int y) {
 if (x < y) return ff(x+1,y);
 return x;
}
```

a) buď hned vrátí první parametr nebo jen „do nekonečna“ volá sama sebe
b) vrátí maximální hodnotu z obou parametrů
c) vrátí součet svých parametrů
d) vrátí x+1
e) neprovede ani jednu z předchozích možností

Řešení: Když je x větším (nebo stejným) z obou parametrů, jeho hodnota je vrácena ihned. V opačném případě se v rekurzivním volání jeho hodnota zvětšuje tak dlouho, dokud nedosáhne hodnoty y tedy původně většího z obou parametrů. Při návratu z rekurze již k žádným změnám nedochází, opět je tedy vrácena hodnota většího z obou parametrů. Platí možnost b).

23.
Napište rekurzivní funkci, která pro zadané číslo N vypíše řetězec skládající se z N jedniček následovaných 2N dvojkami. Např. pro N = 3 vypíše 111222222.

```
void uloha9 (int n) {
  if ( n <= 0) return;
  printf("1");
  uloha9(n-1);
  printf("22");
}
```

24.
Pomocí rekurzivní funkce vypište pro zadané N posloupnost čísel 1 2 ... N-2 N-1 N N N-1 N-2 ... 2 1

```
void uloha10 (int n, int i) {
 if ( i > n)    return;
 printf("%d ", i);
 uloha10(n, i+1);
 printf("%d ", i);
}
```

25.
Sečtěte (odečtěte, vynásobte, vydělte, umocněte) dvě nezáporná celá čísla pomocí rekurzivní (samozřejmě neefektivní) funkce.

```
int uloha11soucet (int a, int b) {
 if ( b == 0 )  return a;
 return uloha11soucet(a, b-1) + 1;  // aritmetika jen +1 a -1
}

int uloha11rozdil (int a, int b) {
 if ( b == 0 )  return a;
 return uloha11rozdil(a, b-1) - 1;
}

// dalsi varianty samostatne :-)
```

26.
Napište rekurzivní funkci, která vypíše pouze hodnoty uložené v listech daného stromu (nebo jen ve vnitřních uzlech).

```
void jenlisty(  NODE *ptr) {
   if (ptr == NULL) return;
   jenlisty(ptr->left);
   if ((ptr->left == NULL) && (ptr->right == NULL))
      printf("%d ", ptr->key);
   jenlisty(ptr->right);
}
// vnitrni uzly se vypisi podobne, vymysli sam ...
```

27.
Posloupnost 1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 lze generovat rekurzivní funkcí zavolanou s parametrem 4.

```
  void ruler(int val) {
    if (val < 1) return;
    ruler(val-1);
    printf("%d%s",val," ");
    ruler(val-1);
  }
```

(Funkce se jmenuje ruler, neboť číselná posloupnost na jejím výstupu charakterizuje délky rysek na pravítku se stupnicí v binární soustavě.)

Zjistěte, co vypíší podobné funkce:
a)
¨

```
  void ruler2(int val) {
    if (val < 1) return;
    printf("%d%s",val," ");
    ruler2(val-1);
    ruler2(val-1);
  }
```

b)

```
  void ruler3(int val) {
    if (val < 1) return;
    ruler3(val-1);
    ruler3(val-1);
    printf("%d%s",val," ");
  }
```

Řešení: a) Např. pro val = 4 dostaneme posloupnost 4 3 2 1 1 2 1 1 3 2 1 1 2 1 1.
Jde o výpis uzlů vyváženého binárního stromu v pořadí preorder, přičemž v každém uzlu se vypisuje číslo rovné hloubce stromu plus 1 zmenšené o hloubku tohoto uzlu.

b) Jako a), jen jde tentokrát o pořadí postorder.

28.
Kolik znaků vypíše každá z funkcí v předchozí úloze, spustíme-li ji s prametrem 20?

Dvě na dvacátou mínus jedna, tedy 1048575.

29.
Sestavte rekurzivní proceduru, která vypíše všechny možnosti rozměnění stokoruny na 1, 2,5,10,20,50 korunová platidla.
Úloha vyžaduje pro většinu zájemců více než 5-10 minut času i s odladěním, nejlépe tedy ji provést jako samostanou domácí úlohu.

30.
Ackermanova A(*n*, *m*) funkce je definována níže. Vypočtěte ručně hodnotu A(2, 2). Zjistěte, pro které dvojice *n*, *m* se dá hodnota A(*n*, *m*) vypočítat na běžném počítači (není jich příliš mnoho).

$$A(n,m) = \begin{cases} m+1 & \text{pro } n=0 \\ A(n-1,1) & \text{pro } n>0,\ m=0 \\ A(n-1,A(n,m-1)) & \text{pro } n>0,\ m>0 \end{cases}$$

31.
Schodová posloupnost
Posloupnost celých čísel nazveme schodovou, pokud absloutní hodnota rozdílu každých dvou sousedních prvků je právě 1. Prázdnou posloupnost a posloupnost s jediným prvkem považujeme také za schodové.
Ukázka 1.
a) 1 2 3 4 3 2 1
b) 1 2 1 2 1 2 3 2 3 4 3 4 5 4 5
c) 0 −1 −2 −1 −2 −1 0 1 0
d) 1 2 3 3 4 5 5
e) 8 7 5 4 3 4
Posloupnosti a), b), c) jsou schodové, posloupnosti d), e) nejsou schodové.
Ukázka 2.
Uvažujme nyní jako prvky posloupnosti pouze čísla 0 1 2 a délku posloupnosti rovnou 3. Všech schodových posloupností s těmito parametry je právě 6 a jsou to: 0 1 0, 0 1 2, 1 0 1, 1 2 1, 2 1 0, 2 1 2.

Úloha
Jsou dána celá čísla 1, 2, 3, ..., N a nezáporné celé číslo L. Napište program, jehož vstupem budou hodnoty N a L a výstupem bude seznam všech schodových posloupností délky L, které obsahují pouze hodnoty 1, 2, 3, ..., N. Použijte rekurzivní funkci.

```
// Zakladni reseni, zkuste najit elegentnejsi

  int pole[NEJAKA_VELIKOST];

  void vsechnyschody (int maxN, int delka) {
    int i;
    for( i = 1; i <= maxN; i++)
      schody(i, maxN, delka, 0);
  }

  int schody(int prvni, int maxN, int delka, int pozice) {
    int i;
    pole[pozice] = prvni;
    if (delka == 1) {
      for (i=0; i <= pozice; i++)
        printf("%d ", pole[i]);
      printf("\n");
      return;
    }
    if (prvni > 1)    schody(prvni-1, maxN, delka-1, pozice+1);
    if (prvni < maxN) schody(prvni+1, maxN, delka-1, pozice+1);
  }
```

32.

Je dána funkce fff vypsaná níže. Ve svém těle kromě sama sebe volá funkci abcd, která rekurzivní není a která proběhne v konstantním čase pro každou hodnotu svých parametrů. Vytvořte funkci ggg, která bude provádět tutéž činnost jako funkce fff, nebude však rekurzivní. Nápověda: Můžete se inspirovat nerekurzivním průchodem binárním stromem.

```
void fff(int x, int y) {
  if (x+y <= 0) return;
  fff(x-2, y-2);
  abcd(x,y);
  fff(x-2, y-2);
}
```

Jediná činnost nad daty, kterou funkce fff ve skutečnosti provádí, je zahrnuta ve funkci abcd. Když vytvoříme strom rekurzivního volání funkce fff, vidíme, že postupná jednotlivá volání funkce abcd odpovídají zpracování tohoto stromu v pořadí inorder. Naše úloha se tak redukuje na průchod binárním stromem v pořadí inorder bez použití rekurze, tedy s použitím vlastního zásobníku.

Použijeme-li zásobník celkem přirozeným způsobem k tomu, abychom si v něm pamatovali celou cestu z kořene do aktuálního uzlu, zapíšeme obecně symbolicky průchod stromem takto:

```
if (root != null) { root.visits++; stack.push(root) };
while (!stack.empty())
  switch (stack.top.visits) {
    case 1: stack.top.visits++;
            if (stack.top.left != null)
              stack.push(stack.top.left);
            break;
    case 2: process(stack.top);
            stack.top.visits++;
            if (stack.top.right != null)
              stack.push(stack.top.right);
            break;
    case 3: stack.pop();
            break;
   }
}
```

Tvar a velikost stromu v naší úloze je dána počátečními hodnotami parametrů x a y. Jeden prvek zásobníku bude obsahovat dvojici hodnot aktuálních parametrů x a y funkce fff a navíc informaci, kolikráte již byl daný uzel navštíven. Existence jak pravého tak levého potomka uzlu u ve stromu rekurzivního volání je rovnomocná splnění podmínky **`((x-2)+(y-2) <= 0)`**, za předpokladu, že uzel u je charakterizován dvojicí hodnot (**`x,y`**). Toto jsou hlavní odlišnosti od obecného schématu uvedeného výše. Zbývá je jen do něj dosadit.

```
void ggg (int x, int y) {
(x+y <= 0) return;
stack.push(x,y,1);
while (!stack.empty() {
  switch (stack.top.visits) {
    case 1: stack.top.visits++;
            if (stack.top.x+stack.top.y-4 > 0)
              stack.push(stack.top.x-2, stack.top.y-2, 1);
            break;
    case 2: abcd(stack.top.x, stack.top.y);          // process!!
            stack.top.visits++;
            if (stack.top.x+stack.top.y-4 > 0)
              stack.push(stack.top.x-2, stack.top.y-2, 1);
            break;
```

```
    case 3: stack.pop();
            break;
   }
}
```

33.
Je dána funkce fff vypsaná níže. Ve svém těle kromě sama sebe volá funkci abcd, která rekurzivní není a která proběhne v konstantním čase pro každou hodnotu svých parametrů. Vytvořte funkci ggg, která bude provádět tutéž činnost jako funkce fff, nebude však rekurzivní. Nápověda: Můžete se inspirovat nerekurzivním průchodem binárním stromem.

```
void fff(int x, int y) {
  if (x+y <= 0) return;
  abcd(x,y);
  fff(x-2, y-2);
  fff(x-2, y-2);
}
```

Řešitele této úlohy odkazujeme na řešení předchozí úlohy. Jediný rozdíl mezi úlohami je v pořadí zpracování uzlů ve stromu rekurzivního volání. Tam je to inorder, zde je to preorder. Průchod stromem sám o sobě se děje vždy ve stejném pořadí uzlů, liší se jen okamžiky, kdy dochází ke zpracování uzlu. V prořadí preorder je to při první návštěvě, v pořadí inorder je to při druhé návštěvě. Schématický kód se tedy bude lišit pouze místem, v němž je volána funkce abcd, jinak jsou všechny další úvahy identické.

```
void ggg (int x, int y) {
 (x+y <= 0) return;
 stack.push(x,y,1);
 while (!stack.empty())
   switch (stack.top.visits) {
     case 1: abcd(stack.top.x, stack.top.y);       // process!!
             stack.top.visits++;
             if (stack.top.x+stack.top.y-4 > 0)
               stack.push(stack.top.x-2, stack.top.y-2, 1);
             break;
     case 2: stack.top.visits++;
             if (stack.top.x+stack.top.y-4 > 0)
               stack.push(stack.top.x-2, stack.top.y-2, 1);
             break;
     case 3: stack.pop();
             break;
    }
}
```

---------- RECURSION MASTER THEOREM ----------

1.
Rekurzivní algoritmus A dělí úlohu o velikosti n na 2 stejné části, pro zisk výsledku musí každou tuto část zpracovat dvakrát. Čas potřebný na rozdělení úlohy na části a na spojení dílčích řešení je úměrný hodnotě n. Asymptotická složitost algoritmu A je popsána rekurentním vztahem

a) $T(n) = 4T(n/2) + n$
b) $T(n) = n \cdot T(n \cdot 4/2)$
c) $T(n) = T(n/2) + 4n/2$
d) $T(n) = 2T(n/4) + n$
e) $T(n) = n \cdot T(n/2) + n \cdot \log(n)$

2.
Rekurzivní algoritmus A dělí úlohu o velikosti n na 3 stejné části a pro zisk výsledku stačí, když zpracuje pouze dvě z nich. Čas potřebný na rozdělení úlohy na části a na spojení dílčích řešení je úměrný hodnotě n^2. Asymptotická složitost algoritmu A je popsána rekurentním vztahem

$T(n) = n \cdot T(n \cdot 3/2)$
$T(n) = T(n/3) + 2n/3$
$T(n) = 3T(n/2) + n^2$
$T(n) = n \cdot T(n/2) + n^2$
$T(n) = 2T(n/3) + n^2$

3.
Rekurzivní algoritmus A dělí úlohu o velikosti n na 4 stejné části, zpracuje však jen tři z nich. Čas potřebný na rozdělení úlohy na části a na spojení dílčích řešení je úměrný hodnotě n. Asymptotická složitost algoritmu A je popsána rekurentním vztahem

$T(n) = 4T(n/3) + n$
$T(n) = n \cdot T(n \cdot 4/3)$
$T(n) = 4T(3n) - n$
$T(n) = 3T(n/4) + n$
$T(n) = n \cdot T(n/3) + n \cdot \log(n)$

4.
Rekurzivní algoritmus A dělí úlohu o velikosti n na 3 stejné části, každou z nich zpracuje dvakrát. Čas potřebný na rozdělení úlohy na části a na spojení dílčích řešení je úměrný hodnotě n^2. Asymptotická složitost algoritmu A je popsána rekurentním vztahem

$T(n) = 3T(n/6) + n^2$
$T(n) = n^2 \cdot T(n/3)$
$T(n) = 6T(n/3) + n^2$
$T(n) = 6T(3n) - n^2$
$T(n) = n^2 \cdot T(n/3) - n^2$

5.
Daný rekurzivní algoritmus pracuje tak, že pro $n > 1$ data rozdělí na 4 části stejné velikosti, zpracuje 5 těchto částí (tj. jednu z nich dvakrát) a pak jejich řešení spojí. Na samotné rozdělení problému a spojení řešení menších částí potřebuje dobu úměrnou hodnotě $n^2 - n$.

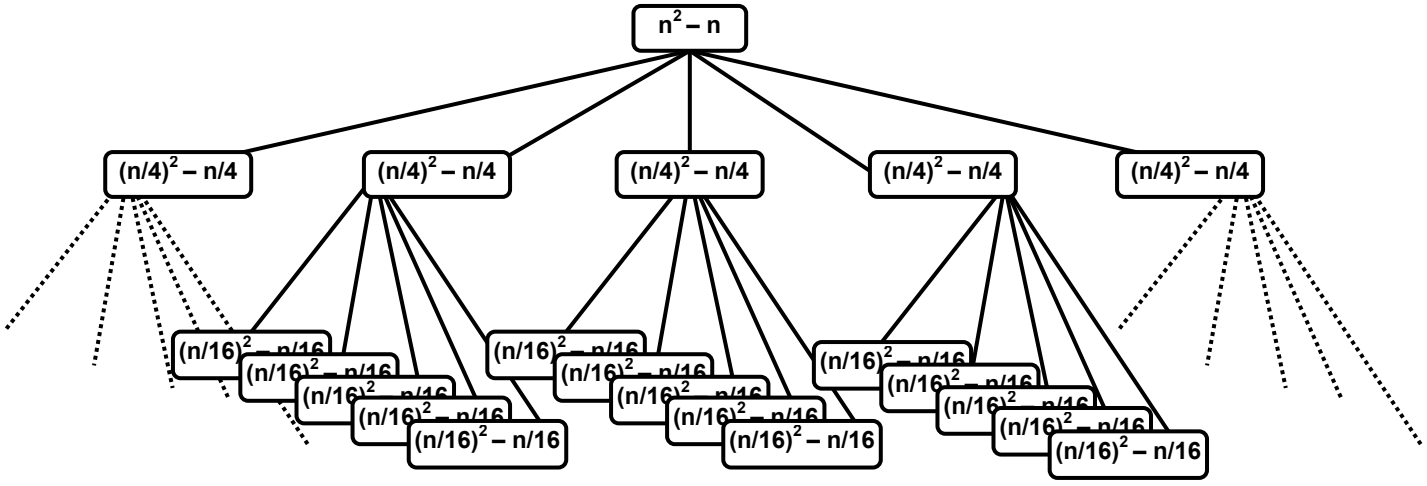
a. Nakreslete první tři úrovně (kořen a dvě další) stromu rekurze.

b. Předpokládejte, že kořen stromu odpovídá činnosti algoritmu nad daty velikosti n.
Vypočtěte cenu uzlu v hloubce 2 (=ve 3. úrovni) stromu. Cena uzlu je doba, kterou algoritmus potřebuje na rozdělení dat a sloučení vyřešených podproblémů při velikosti dat, která odpovídá hloubce uzlu.

c. Vypočtěte hloubku stromu rekurze.

d. Zjistěte asymptotickou složitost daného algoritmu použitím Mistrovské věty.

Řešení:
a. Viz obrázek. (na okrajích to vypadá stejně v hloubce 2 uprostřed...)



b. Cena je vepsána v uzlech v hloubce 2, tj. je to $n^2/256 - n/16$.

c. Hloubka je zřejmě $\log_5(n)$.

d. Nutno určit, která podmínka 1.-3. Mistrovské věty nastává.
Zkoumáme proto vztah veličin
$n^{\log_4(5)}$ a $n^2 - n$.
Zřejmě je $\log_4(5) \approx 1.161$, takže např pro $\varepsilon = 0.5$ bude platit

(*) $n^{\log_4(5)+\varepsilon} < n^2 - n$, pro dostatečně veliká n, tedy také automaticky
$n^2 - n \in \Omega\left(n^{\log_4(5)+\varepsilon}\right)$,

čímž máme splněnu první část podmínky 3. Mistrovské věty. Ověřme její druhou část. Ta praví, že musí být

(**) $5\left(\left(\frac{n}{4}\right)^2 - \frac{n}{4}\right) \le c(n^2 - n)$ pro nějaké $c < 1$ a všechna dostatečně velká n.

Vidíme, že nalevo díky druhé mocnině získáme ve jmenovateli 16, zkusme tedy položit c zdola blízko 1, např.

$c = \frac{15}{16}$. Pak dostaneme

(***) $5\left(\left(\frac{n}{4}\right)^2 - \frac{n}{4}\right) \le \frac{15}{16}(n^2 - n)$. Upravíme levou stranu, dostaneme

$\frac{5}{16}\left(n^2 - 4n\right) \le \frac{15}{16}(n^2 - n)$ a zkrátíme, dostaneme

$n^2 - 4n \le 3n^2 - 3n$. To upravíme na konečný tvar

$-n \le 2n^2$.

Poslední nerovnost platí pro každé přirozené n.
Protože jsme prováděli pouze ekvivalentní úpravy nerovnice (***), uzavíráme, že jsme našli $c = \frac{15}{16} < 1$, pro které platí (**), tudíž je splněna i druhá část podmínky 3. Mistrovské věty, a můžeme tvrdit, že asymptotická složitost daného algoritmu je $\Theta\left(n^2 - n\right) = \Theta\left(n^2\right)$.

Poznámka
Někdo může namítnout, že vztah (*) „spadl z nebe“ a že není jasné, proč by měl platit. Ověřme to.
Máme ukázat, že platí

(*) $n^{\log_4(5)+\varepsilon} < n^2 - n$, pro ε = 0.5 a pro dostatečně veliká *n*.
Zřejmě platí
(i) $\log_4(5) + \varepsilon \approx 1.161 + 0.5 < 1.75$.
Tedy také platí
(ii) $n^{\log_4(5)+0.5} < n^{1.75}$.
Pokud tedy ukážeme, že pro všechna dostatečně velká *n* platí
$n^{1.75} < n^2 - n$,
budeme hotovi.
To je však jednoduché, vydělme obě strany poslední nerovnice členem $n^{1.75}$. Dostaneme

(****) $1 < \frac{n^2 - n}{n^{1.75}} = n^{0.25} - n^{-0.75} = \sqrt[4]{n} - \frac{1}{n^{\frac{3}{4}}}$.

Na pravé straně roste výraz $\sqrt[4]{n}$ do nekonečna a naopak výraz $\frac{1}{n^{\frac{3}{4}}}$ konverguje k nule, takže celkově výraz $\sqrt[4]{n} - \frac{1}{n^{\frac{3}{4}}}$ roste do nekonečna, tudíž je pro dostatečně velké *n* větší než 1, čímž je ověřena nerovnost (****) a jsme hotovi.
Připomínám, že (ii) plyne z (i) díky tomu, že exponenciála je rostoucí funkce pro základ větší než 1. Pokud by někdo selhával v této partii, musí si zopakovat kurs matematiky předtím, než bude moci dokončit ALG :-).

6.
Předchozí Úlohu řešte dále pro případy a, b, c, d níže, postup výpočtu bude analogický:
a. Daný rekurzivní algoritmus pracuje tak, že pro *n* > 1 data rozdělí na 3 části stejné velikosti, zpracuje každou tuto část dvakrát a pak jejich řešení spojí. Na samotné rozdělení problému a spojení řešení menších částí potřebuje dobu úměrnou hodnotě $\sqrt{n} \cdot \log_2(n)$.

b. Daný rekurzivní algoritmus pracuje tak, že pro *n* > 1 data rozdělí na 6 částí stejné velikosti, zpracuje každou tuto část a pak jejich řešení spojí. Na samotné rozdělení problému a spojení řešení menších částí potřebuje dobu úměrnou hodnotě $(n + 1)^2$.

c. Daný rekurzivní algoritmus pracuje tak, že pro *n* > 1 data rozdělí na 6 částí stejné velikosti, zpracuje 3 tyto části a pak jejich řešení spojí. Na samotné rozdělení problému a spojení řešení menších částí potřebuje dobu úměrnou hodnotě $\sqrt{n} + \log_2(n)$.

d. Daný rekurzivní algoritmus pracuje tak, že pro *n* > 1 data rozdělí na 3 části stejné velikosti, zpracuje každou tuto část a pak jejich řešení spojí. Na samotné rozdělení problému a spojení řešení menších částí potřebuje dobu úměrnou hodnotě $(n - 1)^2$.