Lecture 3: More recursion and backtracking examples

BE5B33ALG — Algorithms

Ing. Robert Pěnička, Ph.D.
doc. RNDr. Daniel Průša, Ph.D.

Faculty of Electrical Engineering
Czech Technical University in Prague

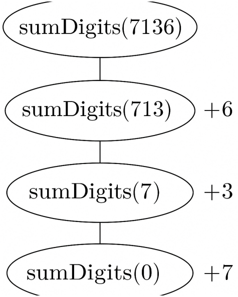
Recursion reminder

- Recursion is when a function calls itself in order to solve a problem.
- Instead of solving the whole (big) problem at once, we can break it down into smaller subproblems that are solved recursively.
- It always has to have two key parts:
  - Base case — a simple stopping condition so it does not call itself forever.
  - Recursive step — the function calling itself with a smaller or simpler input.

Example 1: Sum of digits in an integer

Task: Compute recursively the sum of all digits in an integer.

Method by Example: For integer 7136 we want to get result $7 + 1 + 3 + 6 = 17$.



- Any idea how to do it recursively?
- What could be the recursive step and when to stop the recursion?

Example 1: Sum of digits in an integer

Task: Compute recursively the sum of all digits in an integer.

Method by Example: For integer 7136 we want to get result $7+1+3+6=17$.

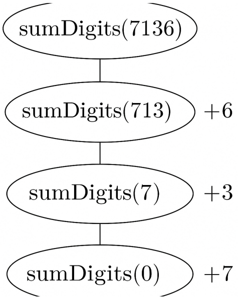
How do we access the last (0-th order) digit of an integer X?

$$\text{last digit} = X \bmod 10$$

How do we remove the last (0-th order) digit of an integer X?

$$X = \left\lfloor \frac{X}{10} \right\rfloor$$

(or in code: `X = X // 10`)



Example 1a: Sum of digits in an integer — global variable

Example of going from the end to the beginning with global variable (simple but not frequent recursion form):

- For $n = 7136$: start with `total_sum = 0`
- **Level 0:** add 6 to total_sum, recurse on 713
- **Level 1:** add 3 to total_sum, recurse on 71
- **Level 2:** add 1 to total_sum, recurse on 7
- **Level 3:** add 7 to total_sum, recurse on 0
- **Level 4:** number is 0 $\rightarrow$ stop recursion

Observation

Each recursive call processes the **last digit** of the number until zero is reached.

Example 1a: Sum of digits in an integer — global variable

Example of going from the end to the beginning with global variable (simple but not frequent recursion form):

- For $n = 7136$: start with `total_sum = 0`
- **Level 0:** add 6 to total_sum, recurse on 713
- **Level 1:** add 3 to total_sum, recurse on 71
- **Level 2:** add 1 to total_sum, recurse on 7
- **Level 3:** add 7 to total_sum, recurse on 0
- **Level 4:** number is 0 $\rightarrow$ stop recursion

The code can look like:

```
total_sum = 0
def sumDigits( n ):
    global total_sum
    if n == 0:
        return
    total_sum += n % 10
    sumDigits( n // 10 )
```

Observation

Each recursive call processes the **last digit** of the number until zero is reached.

Example 1b: Sum of digits — recursive return

A more common recursion form: instead of using a global variable, each call **returns a value** that is summed up.

- Base case: if $n = 0$, return 0.
- Recursive case: return last digit + recursive result of the rest.
- Each call contributes its digit to the final sum.

Recursive equation

$$\text{sumDigits}(n) = \begin{cases} 0, & n = 0 \\ (n \bmod 10) + \text{sumDigits}\left(\left\lfloor \frac{n}{10} \right\rfloor\right), & n > 0 \end{cases}$$

Example 1b: Sum of digits — recursive return

A more common recursion form: instead of using a global variable, each call **returns a value** that is summed up.

- Base case: if $n = 0$, return 0.
- Recursive case: return last digit + recursive result of the rest.
- Each call contributes its digit to the final sum.

The code can look like:

```
def sumDigits2( n ):
    if n == 0:
        return 0
    return (  n % 10 + sumDigits2( n//10 ) )
```

Recursive equation

$$\text{sumDigits}(n) = \begin{cases} 0, & n = 0 \\ (n \bmod 10) + \text{sumDigits}\left(\left\lfloor \frac{n}{10} \right\rfloor\right), & n > 0 \end{cases}$$

Example 1b: Sum of digits — recursive return

Tracing execution:

For $n = 7136$:

- $7136 \rightarrow 6+$ sumDigits2b(713)
- $713 \rightarrow 3+$ sumDigits2b(71)
- $71 \rightarrow 1+$ sumDigits2b(7)
- $7 \rightarrow 7+$ sumDigits2b(0)
- $0 \rightarrow 0$ (base case)
- returning to the top:
  - Level 3 returns $7 + 0 = 7$
  - Level 2 returns $1 + 7 = 8$
  - Level 1 returns $3 + 8 = 11$
  - Level 0 returns $6 + 11 = 17$

```
n = 7136
print("Example 2b, n =", n)
s = sumDigits2b(n)
print(s)
```

Observation

The final result exists only after returning to the root of recursive tree.

Example 1c: Sum of digits — sum top $\rightarrow$ bottom

Another recursion pattern: the **partial result is computed in upper levels** of recursion and passed deeper.

- At each level, we **accumulate the sum so far**.
- When $n = 0$, we return the accumulated sum.
- The result is already complete at the bottom of recursion.

Idea

Unlike before, the sum is not built on the way *back up*, but is carried *downwards*.

Example 1c: Sum of digits — sum top → bottom

Another recursion pattern: the **partial result is computed in upper levels** of recursion and passed deeper.

- At each level, we **accumulate the sum so far**.
- When $n = 0$, we return the accumulated sum.
- The result is already complete at the bottom of recursion.

Idea

Unlike before, the sum is not built on the way *back up*, but is carried *downwards*.

The code can look like:

```
def sumDigits3(n, sumSoFar):
    if n == 0:
        return sumSoFar
    sumSoFar += n % 10
    return sumDigits3(n // 10,
                      sumSoFar)
```

Example 1c: Sum of digits — sum top → bottom

Tracing execution: For $n = 45645622$:

- Level 0: sumSoFar $= 0 + 2 = 2$
- Level 1: sumSoFar $= 2 + 2 = 4$
- Level 2: sumSoFar $= 4 + 6 = 10$
- Level 3: sumSoFar $= 10 + 5 = 15$
- ...
- Base case: $n = 0$, return sumSoFar $= 34$

Observation

The **final result exists already at the bottom**, no need to add values while returning.

```python
n = 45645622
print("Example 1c, n =", n)
s = sumDigits3(n, 0)
print(s)

# condensed form
def sumDigits3b(n, sumSoFar):
    if n == 0:
        return sumSoFar
    return sumDigits3b(n // 10,
                       sumSoFar + n % 10)
```

Example 2: Minimum of an array (classical iterative)

Task: Find the minimum value in an array.

- Classic approach: iterate through array elements.
- Keep track of the smallest value seen so far.
- Return it at the end.

Idea

Compare each new element with the current minimum.

Example 2: Minimum of an array (classical iterative)

Task: Find the minimum value in an array.

- Classic approach: iterate through array elements.
- Keep track of the smallest value seen so far.
- Return it at the end.

Idea

Compare each new element with the current minimum.

```
def myMin0(arr):
    min = arr[0]
    for i in range(1, len(arr)):
        if arr[i] < min:
            min = arr[i]
    return min
```

Example 2: Minimum of an array (recursive call)

Recursive idea:

- Base case: If only one element is left, that element is the minimum.
- Recursive case: Compare the current element with the minimum of the rest of the array.

Tracing execution:

- `myMin1(arr, 0)` $\rightarrow$ compares arr[0] with min of arr[1..end].
- Each call reduces the problem by 1 element.
- Stops when one element is left.

Observation

This is a **linear recursion**: one recursive call per step, shrinking the problem.

Example 2: Minimum of an array (recursive call)

Recursive idea:

- Base case: If only one element is left, that element is the minimum.
- Recursive case: Compare the current element with the minimum of the rest of the array.

Tracing execution:

- `myMin1(arr, 0)` → compares arr[0] with min of arr[1..end].
- Each call reduces the problem by 1 element.
- Stops when one element is left.

Observation

This is a **linear recursion**: one recursive call per step, shrinking the problem.

```
def myMin1( arr, iFrom ):
    # stop recursion? just a single element?
    if iFrom == len(arr) - 1: return arr[iFrom]

    # more elements
    rightMin = myMin1( arr, iFrom+1 )
    if arr[iFrom] < rightMin: return arr[iFrom]
    else:                     return rightMin
```

Example 2: Minimum of an array (recursive binary)

Recursive idea with two calls:

- Base case: if the interval has a single element, that element is the minimum.
- Recursive case: halving the array and computing the minimum of each half.

Observation:

- This is a **binary recursion**.
- The recursion tree resembles a **binary tree**.
- Each level halves the problem size.
- Depth of recursion $= \log_2(n)$.
- Divide and conquer strategy.

Example 2: Minimum of an array (recursive binary)

Recursive idea with two calls:

- Base case: if the interval has a single element, that element is the minimum.
- Recursive case: halving the array and computing the minimum of each half.

Observation:

- This is a **binary recursion**.
- The recursion tree resembles a **binary tree**.
- Each level halves the problem size.
- Depth of recursion $= \log_2(n)$.
- Divide and conquer strategy.

```
def myMin2( arr, iFrom, iTo ):
    # stop recursion? just a single element?
    if iFrom == iTo: return arr[iFrom]

    # more elements
    iMid = (iFrom + iTo) // 2  # index of middle eleme
    # recursive calls
    leftMin  = myMin2( arr, iFrom, iMid )
    rightMin = myMin2( arr, iMid+1, iTo )
    # compute return value
    if   leftMin < rightMin: return leftMin
    else:                    return rightMin
```

Backtracking

- **Backtracking** is a general algorithmic technique for solving problems incrementally.
- It builds candidates for solutions step by step, and **abandons (backtracks)** as soon as it determines that the partial candidate cannot lead to a valid solution.
- Often used in:
  - Combinatorial search problems (e.g., subset sum, knapsack, N-Queens).
  - Constraint satisfaction (e.g., Sudoku, puzzles).

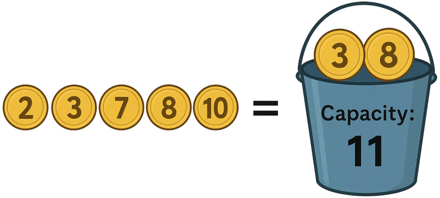
Key Idea

- Explore all possibilities, but prune paths early when they cannot succeed.
- Traverses this search tree recursively, from the root down, in depth-first order

Example: Container Filling Problem

Task: We are given a set of items, each with a certain weight. We also have a container with a fixed weight capacity.

- We must select a subset of items so that the sum of their weights is **exactly equal** to the container's capacity.
- If no such combination exists, the container cannot be filled completely.



Example

Items: $\{2, 3, 7, 8, 10\}$

Capacity: 11

Possible solution: $3 + 8 = 11$

Example: Container Filling Problem - recursive backtracking strategy

The recursive function `pack(...)` explores two choices for each item:

- **Try including the current item.**
- **Skip the current item.**

Stopping conditions:

Example: Container Filling Problem - recursive backtracking strategy

The recursive function `pack(...)` explores two choices for each item:

- **Try including the current item.**
- **Skip the current item.**

Stopping conditions:

- If `sumSoFar == capacity` → valid solution found.
- If `iItem == len(items)` → no more items left.
- If even all remaining items cannot fill capacity → prune branch.

Key Point

At each recursive call, the algorithm branches into two possibilities, building a decision tree of subsets.

Recursive Packing Function

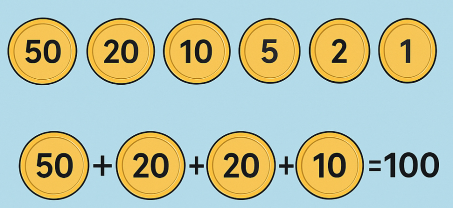
The pack function:

```python
def pack( items, chosenItems,  iItem, capacity, sumSoFar ):
    if sumSoFar == capacity: # solution found?
        print("solution", chosenItems )
        return
    if iItem == len( items ): # no more items available?
        return
    if  sumSoFar + sum(items[iItem:]) < capacity : # not enough left
        return

    currItem = items[iItem] # a) try adding current item
    if currItem + sumSoFar <= capacity:
        chosenItems.append( currItem )
        pack( items, chosenItems, iItem+1, capacity, sumSoFar+currItem )
        chosenItems.pop()    # remove the item after recursion!
    pack( items, chosenItems, iItem+1, capacity, sumSoFar ) # b) adding next item
```

Example: Coin Change Problem

Task: Express a given total as a sum of coins of given value. Each coin value may be used repeatedly.



Inputs:

- A list of available coin values (e.g., `[50, 20, 10, 5, 2, 1]`)
- The target amount (e.g., `100`)

- **Goal:** Find all combinations of coins whose sum equals the target.
- **Example solution:** `[50, 20, 20, 10]`, `[20, 20, 20, 20, 20]`, etc.
- **Note:** for the above coin set, there are 4562 ways to make change for 100.

The recursive function keeps track of:

- `coins` — the coin values (read-only)
- `changeList` — current partial solution
- `iCoin` — index of the coin currently considered
- `givenTotal` — target sum
- `sumSoFar` — sum of chosen coins so far

Select your coin now! (source: dig.watch)

Example: Recursive Algorithm for Coin Change

The recursive function keeps track of:

- `coins` — the coin values (read-only)
- `changeList` — current partial solution
- `iCoin` — index of the coin currently considered
- `givenTotal` — target sum
- `sumSoFar` — sum of chosen coins so far

Select your coin now! (source: dig.watch)

Base cases

- If `sumSoFar == givenTotal` ⇒ we found a solution.
- If no more coins are available ⇒ stop recursion.

Example: Coin Change - choices at each step

At each level of recursion we have two options:

- **Option A:** Add another coin of the same value (if it does not exceed the total).
  `coinChange(coins, changeList+[currCoin], iCoin, givenTotal, sumSoFar+currCoin)`

Example: Coin Change - choices at each step

At each level of recursion we have two options:

- **Option A:** Add another coin of the same value (if it does not exceed the total). `coinChange(coins, changeList+[currCoin], iCoin, givenTotal, sumSoFar+currCoin)`
- **Option B:** Skip to the next coin value. `coinChange(coins, changeList, iCoin+1, givenTotal, sumSoFar)`

Observation

This is a classic **backtracking** pattern: explore a choice, recurse, and then undo the choice.

Example: Coin Change function

The coin change function:

```python
def coinChange(coins, changeList, iCoin, givenTotal, sumSoFar):
    if sumSoFar == givenTotal: # solution found?
        numSol += 1
        print( numSol, changeList )
        return
    if iCoin == len( coins ): # no more coin type available?
        return

    currCoin = coins[iCoin] #  try to add another coin of the same value...
    if currCoin + sumSoFar <= givenTotal:
        changeList.append( currCoin )
        coinChange(coins, changeList, iCoin, givenTotal, sumSoFar + currCoin)  # (*)
        changeList.pop()   # remove the item after recursion!

    coinChange(coins, changeList, iCoin + 1, givenTotal, sumSoFar) # try next coin
```

Coin Change vs. Item Packing

Remark: The two problems are structurally very similar with only one key difference:

- **Item Packing:** `pack(items, chosenItems, iItem+1, capacity, sumSoFar+currItem)`
  - Index is **increased** ⇒ each item can be used at most once.
- **Coin Change:** `coinChange(coins, changeList, iCoin, givenTotal, sumSoFar+currCoin)`
  - Index is **not increased** ⇒ current coin can be used repeatedly.

Conclusion: By controlling whether the index is advanced, we decide if elements are used once or many times. Combination without repetition or with repetition.

Tiling a rectangular board with given tiles

- **Given**: a rectangular board and a multiset of rectangular tiles.
- **Goal**: list *all* tilings that exactly cover the board.
- **Constraints**: tiles cannot overlap, cannot go outside the board, and *all* tiles must be used.
- **Representation**:
  - A tile is `[height, width]` (no stored position).
  - Board cells store `freeSpace` or the tile index.
  - We try placements using the tile's **bottom-right** corner `(posY,posX)`.

```
      0   1   2   3   4   ...  X axis
    +---+---+---+---+---+---+---+
  0 |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+
  1 |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+     The board
  2 |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+
... |   |   |   |   |   |   |   |
    +---+---+---+---+---+---+---+
#  Y axis
```

Feasibility test and place/unplace operations

- We backtrack: for each tile, try every valid position.
- `canPutTile` checks bounds and emptiness of the rectangle.
- `putTile`/`liftTile` write/erase the tile id on the board.

```python
freeSpace = -1

def canPutTile(tileNo, posY, posX):
    currTile = tileList[tileNo]                       # [h, w]
    y0 = posY - currTile[0] + 1                       # UL corner
    x0 = posX - currTile[1] + 1
    if y0 < 0 or x0 < 0:
        return False
    for y in range(y0, posY + 1):
        for x in range(x0, posX + 1):
            if board[y][x] != freeSpace:
                return False
    return True
```

```python
def putTile(tileNo, posY, posX):
    h, w = tileList[tileNo]
    for y in range(posY - h + 1, posY + 1):
        for x in range(posX - w + 1, posX + 1):
            board[y][x] = tileNo

def liftTile(tileNo, posY, posX):
    h, w = tileList[tileNo]
    for y in range(posY - h + 1, posY + 1):
        for x in range(posX - w + 1, posX + 1):
            board[y][x] = freeSpace
```

Recursive enumeration and practical improvements

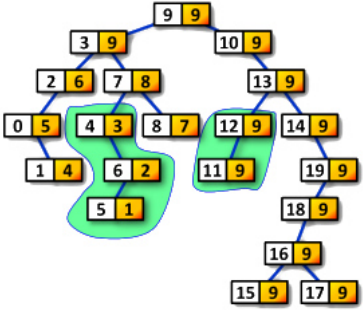
```
def tileBoard(currTileNo):
    # If all tiles are placed, we found a tiling.
    if currTileNo == len(tileList):
        printBoard()
        return
    # Try every board cell for current tile.
    for posY in range(len(board)):
        for posX in range(len(board[0])):
            if canPutTile(currTileNo, posY, posX):
                putTile(currTileNo, posY, posX)
                tileBoard(currTileNo + 1)
                liftTile(currTileNo, posY, posX)

# Example instance:
tileList = [[3,3], [2,2], [2,2], [2,1], [3,2]]
boardHeight, boardWidth = 5, 5
board = [[freeSpace]*boardWidth for _ in range(boardHeight)]
tileBoard(0)
```

Idea:

- Test all placements of bottom right corner of all tiles.
- Backtrack when tile can not be placed
- When tile is placed test the next tile.

Homework: Dangling paths in a binary tree



- Two nodes are **neighbours** if one is the parent of the other.
- The **degree** of a node $=$ number of its neighbours.
- A **path of length** N is a sequence $(x_1, x_2, \ldots, x_N)$ such that each consecutive pair are neighbours, and all nodes are unique.
- A path is a **dangling path** if:
  - degree of each node ≤ 2;
  - x_1 is a leaf;
  - the parent of x_N either does not exist or has degree 3.
- The **cost** of a path $=$ sum of keys of all nodes in it.

Goal: Given a binary rooted tree, find the minimum and maximum cost of a dangling path.

Examples and expected output

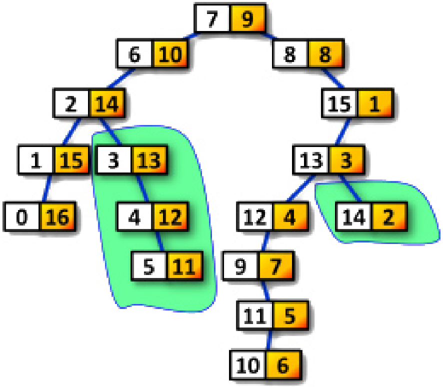
Example 1

- `Input:` $N = 16, R = 7$
- `Output:` **2 36**

Dangling paths with minimum and maximum cost are highlighted.

Hints:

- Representing the tree as a matrix is convenient given tree node indices are 0..N-1 and input data are unordered.
- Ideally first traverse the tree to find which nodes are in a dangling path.
- Then traverse the tree to find min/max cost of dangling paths.



Another example illustrating different tree structure.