

# Algorithms and Data Structures

## Designing algorithms

dr Szymon Murawski

Comarch SA

March 8, 2019

# Table of contents I

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Course plan

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# What is recursion

## Definition

Given a problem, recursive algorithm calls itself one or more times to solve closely related sub-problem. Usually the sub-problems are smaller in size, but that is not a rule.

## Informal definitions

- To understand recursion one must understand recursion
- Before we can define recursion, we must define recursion
- We define unknown by unknown, that will become known
- When a thing is defined in terms of itself
- Possibility of defining an infinite set of objects by a finite statement.

# Recursive function

- Recursive function must contain at least one base case and one or more recursive calls
- Base case should return simple type, without calls to any recursive functions
- Function can recursively call itself, or a connected

## Types of recursion

- Direct single recursion - function calls itself once
- Direct multiple recursion - function has multiple calls to itself
- Indirect recursion - function calls another function, that will recursively call original function

# Do use recursion!

- Recursion is very powerful programming method!
- Do use it!
- Some functional languages do not define loops - they rely solely on recursive calls

# Factorial - recursive

## Definition

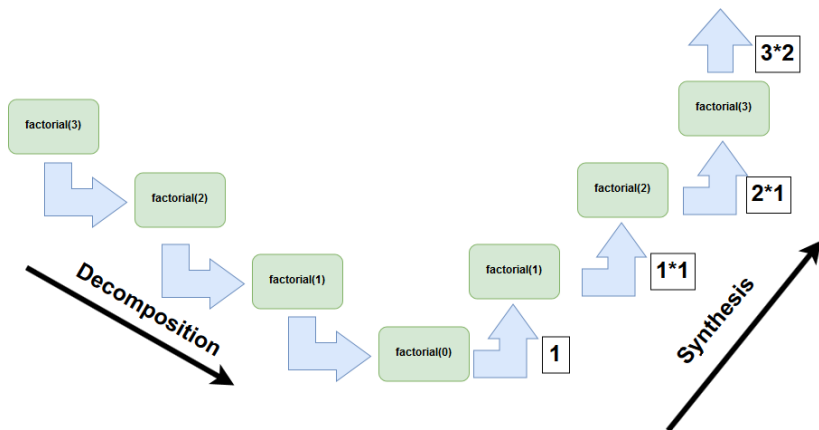
$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n(n-1)!, & \text{if } n > 1 \end{cases}$$

## Pseudocode

- Input: Positive integer  $n$
- Output: Positive integer factorial of  $n$

```
1  function factorial(n)
2      if n = 0 then return 1
3      else return n*factorial(n-1)
4
```

# Recursive factorial call stack



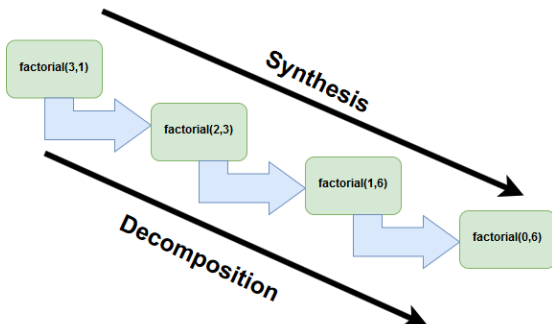


# Alternative implementation

## Pseudocode

- Input: Positive integer  $n$
- Output: Positive integer factorial of  $n$

```
1 function factorial(n, tmp = 1)
2   if n = 0 then return tmp
3   else return factorial(n-1, n*tmp)
```



# Tail recurrence

- When the last instruction of a recursive function is a simple call to itself we call it **tail recurrence**
- `return factorial(n-1, n*tmp)` - this is tail recurrence
- `return n*factorial(n-1)` - this is not
- Modern compilers can optimize functions with tail recurrence
- If it is possible, we should create recursive functions with tail recurrence, although in some cases it is impossible or very hard.

# Memory complexity

- In recursive algorithms thing that must be taken into account is memory efficiency
- Each new call to recursive function places a lot of function overhead onto stack
- Depending on the algorithm it could be very easy or very hard to predict how many recursive calls a program will make.
- This is why we should try to obtain tail recurrence, so the compiler can optimize the function

# Using recursion

- Describing algorithms using recursion is often easier and more concise than using iteration
- However efficiency of recursive algorithms is often worse than iterative approach (not true for functional programming).
- Efficiency is worse due to all the function calls the algorithm needs to make, as well as passing all the arguments and retrieving all results.
- Every recursive algorithm however can be transformed into iterative one, although for some algorithms it can be very hard

For most of the software You will write, drop in efficiency will not be enough to justify NOT using recursion. You should use recursion, as clean code is above all!

# Course plan

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Course plan

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Calculating Fibonacci sequence

## Definition

$$f(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ f(n-1) + f(n-2), & \text{if } n > 1 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, ...

## Pseudocode

- Input: Positive integer  $n$
- Output: Value of  $n$ -th number in Fibonacci sequence

```
1 1.  function fibonacci(n)
2 2.      if n = 0 || n = 1 then return n
3 3.      else return fibonacci(n-1) + fibonacci(n-2)
4
```

## Call stack

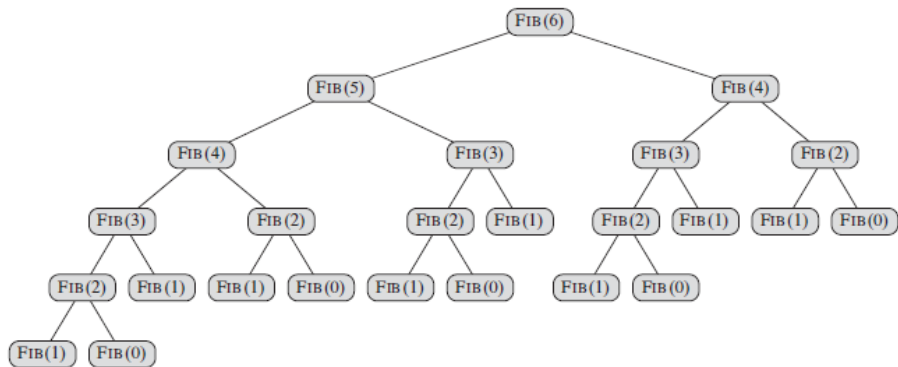
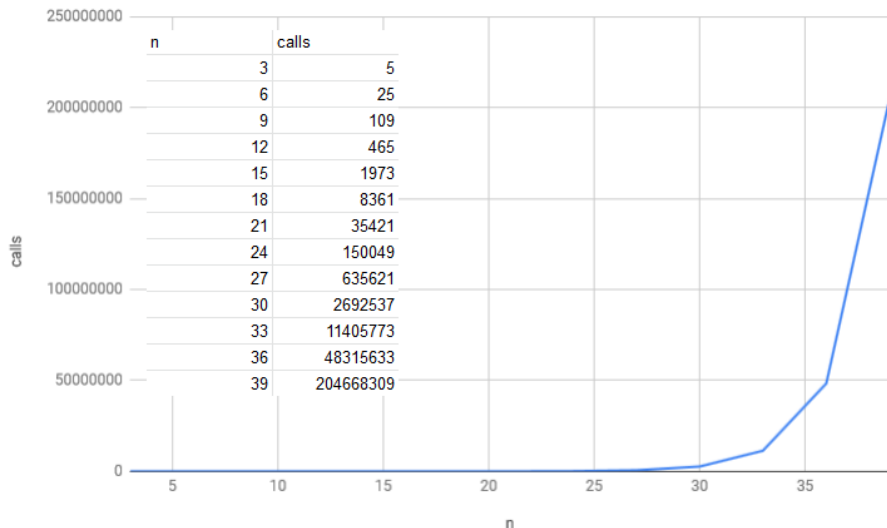


Figure: Taken from: Cormen, Leiserson, Rivest, Stein - *Introduction to algorithms*



# Fibonacci recursive calls



# Problems with naive recursive approach

- In naive implementation of Fibonacci function we are repeating a lot of calculations
- Every level of our tree consists of the same values, that we need to calculate once again
- This approach grows exponentially with the size of our problem

# Course plan

- 1 Recursion
- 2 **Dynamic programming**
  - Fibonacci sequence
  - **Dynamic programming**
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Dynamic programming

- Developed by Richard Bellman in 1957
- Dynamic Programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions
- Basically a fancy way of saying "remembering stuff to save time later"
- Dynamic programming trades space for speed, by storing solutions to sub-problems rather than calculating them time and time again

# Elements of dynamic programming

- Constructing solution to a problem by building it up dynamically from solutions to smaller (or simpler) sub-problems
  - sub-instances are combined to obtain sub-instances of increasing size, until finally arriving at the solution of the original instance.
  - make a choice at each step, but the choice may depend on the solutions to sub-problems
- Principle of optimality - the optimal solution to any nontrivial instance of a problem is a combination of optimal solutions to some of its sub-instances.
- Memoization (for overlapping sub-problems) - avoid calculating the same thing twice, usually by keeping a table of known results that fills up as sub-instances are solved.

# Development of a dynamic programming algorithm

- Characterize the structure of an optimal solution, breaking a problem into sub-problem whether principle of optimality apply
- Recursively define the value of an optimal solution - define the value of an optimal solution based on value of solutions to sub-problems
- Compute the value of an optimal solution in a bottom-up fashion
  - save the values along the way
  - later steps use the save values of previous steps
- Construct an optimal solution from computed information

# Fibonacci sequence - dynamic programming

## Pseudocode

- Input: Positive integer  $n$
- Output: Value of  $n$ -th number in Fibonacci sequence

```
1 1.  function fibonacci(n)
2 2.      if  $n = 0 \ || \ n = 1$  then return  $n$ 
3 3.      else
4 4.          int[] fibArr = new int[n+1] {0, 1}
5 5.          for(int i = 2; i<=n; i++)
6 6.              fibArr[i]=fibArr[i-1]+fibArr[i-2]
7 7.          return fibArr[n];
8
```

# Fibonacci sequence - dynamic programming v2

## Pseudocode

- Input: Positive integer  $n$
- Output: Value of  $n$ -th number in Fibonacci sequence

```
1 1.  function fibonacci(n)
2 2.      int f=0, f0=1, f1=1
3 3.      if n = 0 || n = 1 then return n
4 4.      else
5 5.          while (n >=2)
6 6.              f = f0 + f1
7 7.              f0=f1
8 8.              f1=f
9 9.              n—
10 10.     return f
11
```



# Comparison of Fibonacci algorithms

## Simple recursive algorithm

- Easy to construct straight from mathematical definition
- High memory and time complexity
- Depth of recurrence linearly dependent on problem size
- Repetition of calculations for subproblems

## Simple dynamic programming algorithm

- Quite easy to construct straight from mathematical definition
- Definitely better both time and memory scaling
- No recursive calls
- Requires an array to store subresults, with size linearly dependent on problem size

# Comparison of Fibonacci algorithms

## Dynamic programming algorithm v2

- Requires significant modification of the algorithm
- Constant memory requirements (only three integers)
- No recursive calls
- Slightly faster than previous method

# Course plan

- 1 Recursion
- 2 **Dynamic programming**
  - Fibonacci sequence
  - Dynamic programming
  - **0-1 Knapsack problem**
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# 0-1 Knapsack problem

## Definition

Given the knapsack of size  $W$  and list of items, each of size  $w_i$  and value  $v_i$ , which items should be put into the knapsack to maximize the value of items carried.

- Also known as thief/rucksack/student problem.
- Items cannot be split in half, we can either take it or leave it
- Brute-force solutions is not feasible - it computes in exponential time  $2^n$
- It is optimization problem

## 0-1 Knapsack problem



Figure: Micah George @Pinterest

# 0-1 Knapsack problem solution

- If we calculate the total weight and value of all the subsets of total capacity  $W$ , we can then pick the best one
- This problem can be solved by dynamic programming!
- We can divide the problem into smaller one:  $n$ th item can either be or not be a part of optimal solution
  - If we include the item, then we need to solve problem for  $n - 1$  items and weight  $W - w_i$ , where  $w_i$  is the weight of  $n$ th item.
  - If we exclude the item, then we need to solve problem for  $n - 1$  items and weight  $W$

# 0-1 Knapsack problem solution

- 1 Create 2D array  $V[0..n, 0..W]$ . For  $1 \leq i \leq n$  and  $0 \leq w \leq W$ ,  $V[i, w]$  will store the maximum value of subset of items  $1, 2, \dots, i$  of combined weight at most  $w$ . Array element  $V[n, W]$  will contain the maximum value!
- 2 Initialize the array, set  $V[0, w] = 0$  for  $0 \leq w \leq W$  (no item) and  $V[i, 0]$  for  $0 \leq i \leq n$  (no space left).
- 3 Recursively define the value of an optimal solution in terms of solutions to smaller problems:

$$V[i, w] = \max(V[i - 1, w], v_i + V[i - 1, w - w_i])$$

for  $1 \leq i \leq n, 0 \leq w \leq W$ .

# Course plan

- 1 Recursion
- 2 **Dynamic programming**
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - **0-1 Knapsack example**
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem



# Example

Given  $W = 10$  and

$i$	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

$$V[i, w] = \max(V[i - 1, w], v_i + V[i - 1, w - w_i])$$

$V[i, w]$	$w = 0$	1	2	3	4	5	6	7	8	9	10
$i = 0$											
1											
2											
3											
4											

# Example - Initialize

Given  $W = 10$  and

i	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

$$V[i, w] = \max(V[i - 1, w], v_i + V[i - 1, w - w_i])$$

$V[i, w]$	$w = 0$	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0										
2	0										
3	0										
4	0										

# Example - $i=1$

Given  $W = 10$  and

$i$	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w - w_i])$$

$V[i, w]$	$w = 0$	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0										
3	0										
4	0										

# Example - $i=2$

Given  $W = 10$  and

$i$	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w - w_i])$$

$V[i, w]$	$w = 0$	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0										
4	0										

# Example - $i=3$

Given  $W = 10$  and

$i$	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

$$V[i, w] = \max(V[i-1, w], v_i + V[i-1, w - w_i])$$

$V[i, w]$	$w = 0$	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0										

# Example - $i=4$

Given  $W = 10$  and

$i$	1	2	3	4
$v_i$	10	40	30	50
$w_i$	5	4	6	3

$$V[i, w] = \max(V[i - 1, w], v_i + V[i - 1, w - w_i])$$

$V[i, w]$	$w = 0$	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

# Course plan

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms**
  - Greedy Problems**
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Greedy algorithms

- Finding an optimal solution to a problem can be very time consuming
- Sometimes we are not interested in optimal solution, but in good enough solution computed in feasible time
- A greedy algorithms finds such a solution, by making a sequence of choices based upon some heuristics
- At each decision point, the algorithms makes choice that seems best at the moment
- As a result we obtain locally optimal solution, but it could be not a global optimal solution!

## Greedy heuristics for 0-1 knapsack problem

- We start packing items starting from the most valuable
- We start packing items starting from the lightest
- We start packing items starting from the ones having best value/weight ratio



# Course plan

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Fractional knapsack problem

- Similar problem to the one discussed before, but this time we can break the items (gold/silver dust, types of grains, meters of fabric etc.)
- Greedy approach:
  - Calculate value/weight ratio for all the items
  - Take as much of the item with the best value/weight ratio as possible
  - Repeat previous step if there is still space left in knapsack

# Job sequencing

Given a list of tasks, each with its deadline and associated profit calculate how much profit can be earned by completing tasks within the specified deadlines. Assume that completing a task takes one unit of time and no task can be completed more than once.

task	deadline	profit
1	4	15
2	2	2
3	1	18
4	3	1
5	4	25

## Greedy approach

- Sort tasks in decreasing order of profits
- Schedule each task from the top of the list at the last possible deadline slot

# Egyptian fraction

Every positive fraction can be represented as sum of unique unit fractions. A fraction is unit fraction if numerator is 1 and denominator is a positive integer, for example  $1/3$  is a unit fraction. Such a representation is called Egyptian Fraction as it was used by ancient Egyptians. Examples:

$$\frac{2}{3} = \frac{1}{2} + \frac{1}{6}$$

$$\frac{6}{14} = \frac{1}{3} + \frac{1}{11} + \frac{1}{231}$$

## Greedy approach

- 1 Given the fraction  $a/b$  calculate the ceiling  $c = \lceil b/a \rceil$
- 2  $a/b = 1/c + (a/b - 1/c)$
- 3 Repeat the first step for the second term

# Other greedy problems

- Coloring nodes of a graph
- Huffman coding
- Shortest substring
- Dijkstra algorithm
- Kruskal's algorithm for minimum spanning tree

# Course plan

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Divide and conquer

In this approach we break the problem into several sub-problems that are similar to the original problem, but smaller in size, solve the sub-problems recursively, and then combine these solutions to create a solution to the original problem

## Divide and conquer steps

- **Divide** the problem into a number of sub-problems that are smaller instances of the same problem
- **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve them in a straightforward manner
- **Combine** the solutions to the sub-problems into the solution for the original problem.

# Divide and conquer

- Divide and conquer is the basis for many other approaches
- It is similar to dynamic programming as it also divides a problem into subproblems, when to choose which?
  - Choose dynamic programming approach, if the subproblems are recalculated many times
  - Choose divide and conquer if the subproblem is calculated only once
- It is one of the ways of how to transform exponential problem into polynomial one, although this approach does not always work

## Typical problems

- Binary search
- Merge sort
- Quicksort
- Matrix multiplication

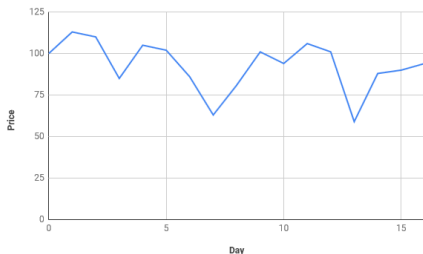


# Course plan

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Stock trader problem

Given a graph of stock values over time, when should you buy and sell for maximum profit?



## Brute force

- Compare every possible pair of buy and sell dates
- Two loops are required
- Complexity  $O(n^2)$

# Maximum subarray problem

Day	Price	Change
0	100	
1	113	13
2	110	-3
3	85	-25
4	105	20
5	102	-3
6	86	-16
7	63	-23
8	81	18
9	101	20
10	94	-7
11	106	12
12	101	-5
13	59	-42
14	88	29
15	90	2
16	94	4

- Start by transforming data, so you obtain daily change
- Reiterate the problem: for which continuous days the sum of change is maximized?
- This problem is also known as **maximum subarray problem**

# Divide and conquer approach

- We can apply divide and conquer approach to this problem
- Let's assume that we keep the change values in array  $A$ , where each index is a day
- According to divide and conquer, if we want to find maximum subarray of array  $A[first \dots last]$ , where  $first$ ,  $last$  are first and last days of a period of our interest, we need to split the array into two subarrays:  $A[first \dots mid]$  and  $A[mid + 1 \dots last]$ , where  $mid$  is a midpoint of our choosing
- Now the maximum subarray  $A[i \dots j]$  must lie in exactly one of these spots:
  - entirely in subarray  $A[first \dots mid]$ , so that  $first \leq i \leq j \leq mid$
  - entirely in subarray  $A[mid + 1 \dots last]$ , so that  $mid < i \leq j \leq last$
  - crossing the midpoint, so that  $first \leq i \leq mid < j \leq last$

# Location of maximum subarrays

Maximum subarray in the left part



Maximum subarray in the right part



Maximum subarray crossing midpoint



# Finding max subarray

- Finding max subarray would then be split into three steps: check left/right/middle part of the array
- Checking left and right part of the arrays are easy - we can make a recursive call to a procedure
- Checking middle part of the array is different, as it has one constrain: it must contain middle element
- We can write function `findMaxSubarray(A,first,last)` that returns sum of maximum subarray, and takes as parameters array `A`, and indexes of `first` and `last` element of the array. This function will make recursive calls to itself
- For finding maximum subarray in the middle part we can write function `findMaxCrossSubarray(A,first,mid,last)`, that returns sum of maximum subarray going through the middle element, that takes as parameters array `A`, and indexes of `first` middle and `last` element of the array.

# findMaxSubarray pseudocode

```
1 findMaxSubarray(A, first , last)
2   return A[first] if first = last
3   mid = floor((first+last)/2)
4   leftSum = findMaxSubarray(A, first , mid)
5   rightSum = findMaxSubarray(A, mid+1, last)
6   crossSum = findMaxCrossSubarray(A, first , mid , last)
7   if leftSum >= rightSum && leftSum >= crossSum
8     return leftSum
9   else if rightSum >= leftSum && rightSum >= crossSum
10    return rightSum
11  return crossSum
```

# findMaxCrossSubarray pseudocode

```
1 findMaxCrossSubarray(A, first, mid, last)
2   leftSum = -infinity
3   sum = 0
4   for i in (mid..first)
5     sum += A[i]
6     if sum > leftSum
7       leftSum = sum
8   rightSum = -infinity
9   sum = 0
10  for i in (mid+1..last)
11    sum += A[i]
12    if sum > rightSum
13      rightSum = sum
14  return leftSum + rightSum
```



# Calculating complexity

- To simplify calculations let's assume that size of problem  $n$  is always a power of 2, so that all subproblem sizes are integers
- Complexity of `findMaxCrossSubarray` is linear:  $O(n)$
- If by  $T(n)$  we denote time it takes for `findMaxSubarray` to finish, that it can be written:

$$T(n) = 2T(n/2) + O(n) = 4T(n/4) + 2O(n)$$

- It can be proven, that the solution to the above equation is  $T(n) = O(n \lg n)$
- As brute-force solution was  $O(n^2)$ , we developed a better algorithm using divide and conquer approach!

# Course plan

- 1 Recursion
- 2 Dynamic programming
  - Fibonacci sequence
  - Dynamic programming
  - 0-1 Knapsack problem
  - 0-1 Knapsack example
- 3 Greedy algorithms
  - Greedy Problems
- 4 Divide and conquer
  - Maximum subarray problem
- 5 Coin change problem

# Coin change problem

Given a set of coins  $C = \langle c_1, c_2, \dots, c_n \rangle$  and a change to be made  $x$ , how many of each coins should be returned to customer?

- Examples for  $C = \langle 1, 2, 5, 10, 20, 50, 100 \rangle$ :
  - For  $x = 3$ , return  $1 \times 2, 1 \times 1$
  - For  $x = 47$ , return  $2 \times 20, 1 \times 5, 1 \times 2$
- Greedy and dynamic approach can be used for this problem
- We will limit ourselves to case of integer values, to simplify the problem

# Greedy approach

- In this approach we always take the coin with highest value and check if it less or equal than change to be made. If so, it becomes part of the solution, we subtract coin value from change and we repeat the process
- We are using this exact approach when dealing with almost all monetary systems
- This algorithm scales linear with number of available coins
- Greedy approach to this problem gives optimal result only for some sets of coins, luckily our monetary system is one of them!
- For optimal result in general case we will need to use dynamic programming

# Greedy approach pseudocode

## Inputs

- coins - array of available coin values
- change - change to be made

## Pseudocode

```
1 MakeChange(coins, change){
2   foreach coin in coins{
3     howMany = change/coin;
4     change = change % coin;
5     print howMany + " times " + coin + "$";
6   }
7 }
```

## Dynamic programming approach

- In general case greedy approach does not provide optimal solution
- For example given coins  $C = \langle 1, 3, 4 \rangle$  and input  $x = 6$  greedy approach would produce  $1 \times 4, 2 \times 1$ , while the optimal answer is  $2 \times 3$
- We solve this problem using dynamic approach by solving problems of making change for  $p = 1, 2, 3, \dots, x$  - we start by simple problem and build the solution starting from the bottom!
- We will use two temporary arrays: *minCoins*[] and *firstCoinIndex*[], each of length equal to number of available coins + 1.
- The formula we will be using ( $c_i$  is the value of  $i$ -th coin):

$$\text{minCoins}[p] = \begin{cases} 0, & \text{if } p = 0 \\ \min_{i: c_i \leq p} \{1 + \text{minCoins}[p - c_i]\}, & \text{if } p > 0 \end{cases}$$

- Complexity of the algorithm is  $O(mn)$ , where  $m$  - number of available coins,  $n$  - change to be made

# Dynamic coin change example

- $Coins = \langle 1, 3, 4 \rangle$
- $x = 6$

$$minCoins[p] = \begin{cases} 0, & \text{if } p = 0 \\ \min_{i: c_i \leq p} \{1 + minCoins[p - c_i]\}, & \text{if } p > 0 \end{cases}$$

p	0	1	2	3	4	5	6
minCoins[p]	0	1	2	1	1	2	2
firstCoinIndex[p]		0	0	1	2	0	1

coinIndex	0	1	2
coinValue	1	3	4
howManyCoins	0	2	0

# Pseudocode


















```

1 MakeChange(coins , change)
2   for currChange in (0..change+1)
3     coinCount = currChange
4     newCoin = null
5     for c in coins
6       next if c >= currChange
7       if 1+ minCoins[currChange-c] < coinCount
8         coinCount = 1+ minCoins[currChange-c]
9         newCoin = c
10    minCoins[currChange] = coinCount
11    firstCoinIndex[currChange] = newCoin
12  return minCoins[change]
13
14 PrintCoins(firstCoinIndex , change):
15   currChange = change
16   while currChange > 0
17     coin = firstCoinIndex[currChange]
18     print(coin)
19     currChange -= coin

```



# British predecimal currency

	1 quarter farthing	= 1/16 d.	= 1/192 shilling	= £1/3840
	1 third farthing	= 1/12 d.	= 1/144 shilling	= £1/2880
	1 half farthing	= 1/8 d.	= 1/96 shilling	= £1/1920
	1 farthing	= 1/4 d.	= 1/48 shilling	= £1/960
	1 halfpenny	= 1/2 d.	= 1/24 shilling	= £1/480
	<b>1 penny</b>	<b>= 1 d.</b>	<b>= 1/12 shilling</b>	<b>= £1/240</b>
	1 twopence	= 2 d.	= 1/6 shilling	= £1/120
	1 threepence	= 3 d.	= 1/4 shilling	= £1/80
	1 groat	= 4 d.	= 1/3 shilling	= £1/60
	1 sixpence	= 6 d.	= 1/2 shilling	= £1/40
	<b>1 shilling</b>	<b>= 12 d.</b>	<b>= 1/-</b>	<b>= £1/20</b>
	1 florin	= 24 d.	= 2/-	= £1/10
	1 half crown	= 30 d.	= 2/6	= £1/8
	1 crown	= 60 d.	= 5/-	= £1/4
	1 half sovereign	= 120 d.	= 10/-	= £1/2
	<b>1 sovereign</b>	<b>= 240 d.</b>	<b>= 20 shillings</b>	<b>= £1</b>
	1 guinea	= 252 d.	= 21 shillings	= £1.05