1 Greedy algorithms and dynamic programming

This chapter covers two malgorithm design principles more: *greedy algorithms* and *dynamic programming*.

A greedy algorithm is often the most natural starting point for people when searching a solution to a given problem.

An example: change making problem

- For euro or US dollar coins the problem is solvable by the greedy approach

A greedy algorithm doesn't however always produce correct results as the problem against initial expectations does not fullfill all the requirements for the greedy approach.

An example: "weird" coin values such as 7,5 and 1.

$\Rightarrow$Solution: dynamic programming

1.1 Storing results to subproblems

Techniques that store the results of subproblems are:

- memoization
- dynamic programming

Let's take the algorithm that calculates Fibonacci numbers as an example.

- Fibonacci numbers are defined:

  $F_0 = 0$
  $F_1 = 1$
  $F_n = F_{n-1} + F_{n-2}$, kun $n > 1$

- A divide and conquer recursive solution can be implemented based on the definition:

```
VERY-SLOW-FIBONACCI(n)
1   if n = 0 or n = 1 then          (the solution is known with 0 and 1)
2       return n
3   else                            (the result is calculated from F_{n-1} ja F_{n-2})
4       return VERY-SLOW-FIBONACCI(n − 1)
                + VERY-SLOW-FIBONACCI(n − 2)
```

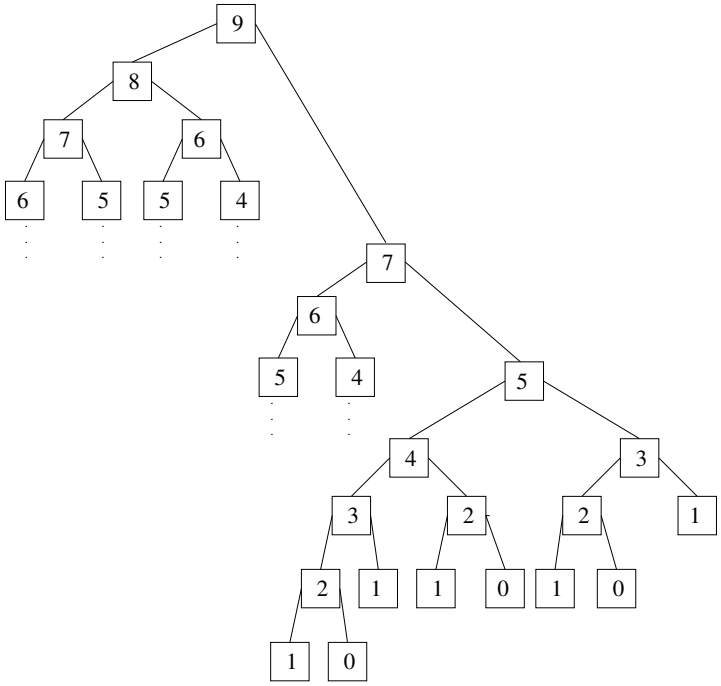
The algorithm is not really that efficient:

- Each VERY-SLOW-FIBONACCI runs in $\Theta(1)$ time
- The picture shows that the solution computes solutions to the subproblems more than once for no reason:

Number:	8	7	6	5	4	3	2	1	0
Times:	1	2	3	5	8	13	21	34	21

$\Rightarrow$ The amount of function calls is $O(2^n)$.
$\Rightarrow$ The running-time of VERY-SLOW-FIBONACCI is exponential

- when this happens it is possible that there is no efficient soluton available
- There **is** one for Fibonacci numbers.



The algorithm calculates the solutions to the subproblems several times. Let's use this information to our advantage and store the results as they get counted.

Thus when the result is needed again, it doesn't need to calculated but the stored value can be used instead.

$\Rightarrow$ *memoization*

Let's implement an algorithm based on memoization to Fibonacci numbers.

- when determining F_n the possible solutions to subproblems are $F_0, F_1, \ldots, F_{n-1}$.
- The algorithm can be made simpler by storing F_n also.

  $\Rightarrow$ Let's use an array $F[0 \ldots n]$ to store the solutions of the subproblems

- The algorithm needs to be able to regonize whether the solution has already been calculated or not
  - Since Fibonacci numbers cannot be negative, let's mark the uncalculated numbers in the array with -1

```
MEMOIZED-FIBONACCI(n)
1  F[0] := 0
2  F[1] := 1
3  ▷ Initialize rest of the elements -1
4  return MEM-FIB-RECURS(n)
```

```
MEM-FIB-RECURS(i)
1  if F[i] < 0 then                    (if the result hasn't been calculated)
2       F[i] := MEM-FIB-RECURS(i − 1) + MEM-FIB-RECURS(i − 2)
3  return F[i]
```

- The running-time of MEMOIZED-FIBONACCI is the running-time of initializing the array $\Theta(n)+$ the running-time of MEM-FIB-RECURS
- The running-time of lines 1 and 3 in MEM-FIB-RECURS is $\Theta(1)$
- Line 2 gets executed atmost once per each element in the array

  $\Rightarrow$ MEM-FIB-RECURS calls itself atmost $2n$ times.

  $\Rightarrow$ the running-time of MEM-FIB-RECURS and the maximum depth of the recursion is $\Theta(n)$.

  $\Rightarrow$ The running-time of the entire MEMOIZED-FIBONACCI and the extra memory requirement is $\Theta(n)$.

General details about algorithms that use memoization:

- Basic idea:
  - form an upper-bound to the set of subproblem results needed
  - design a data structure than can efficiently store said set
  - the parameters of the subproblem solutions act as the key to the data structure (i in the Fibonacci example)
- Most of the time the data structure can be a simple array
- A dynamic set with operations INSERT and SEARCH is needed in a general case
  - i.e. a dictionary without delete
- The programmer doesn't need to worry about the calculation order of the subproblems
  - recursion and the "not counted" mark make sure all results to the subproblems get calculated exactly once

$\Rightarrow$ These algorithms are

- easy to make efficient
- difficult to analyze

- The calculation order of the results is unknown
  $\Rightarrow$ all subproblem solutions need to be stored until the work is completed
  $\Rightarrow$ a lot of memory is consumed
- naturally subproblem results can be deleted if it is sufficiently clear that the amount of work needed to recount them if necesary is kept reasonable

1.2 Dynamic programming

Dynamic programming is also based on memoization of subresults, but the order of calculation is fixed

The subresults are usually stored in an ordinary array $\Rightarrow$ "dynamic tabulation", or something, would be a better name

- typically used in optimization problems:
  - there are several alternate solutions to the problem
  - each solution has a price (cost function)
  - the cheapest solution is needed (or one of the cheapest)

The basic principles of an algorithms that uses dynamic programming:

1. Figure out the optimal structure of the solution
2. Define the optimal solution with the solutions to its subproblems

3. Calculate the result for the optimal solution by calculating the results of the subproblems from the smallest onwards
   - it may be usefull to store information about the optimal solution simultaneously
   - the bottom-up approach is used
4. It the result of the optimal solution is not enough but the actual solution is also needed, it is calculated based the information gathered in stage 3

A dynamic programming algorithm for Fibonacci numbers:

```
DYNAMIC-FIBONACCI(n)
1  F[0] := 0                          (store the base cases in an array)
2  F[1] := 1
3  for i := 2 to n do
4      F[i] := F[i − 1] + F[i − 2]    (F_i determined based on the previous results)
5  return F[n]
```

- The running-time of lines 1 and 2 in DYNAMIC-FIBONACCI is $\Theta(1)$
- Line 4 gets executed once per each element in the array $\Rightarrow \Theta(n)$

  $\Rightarrow$ the running-time of DYNAMIC-FIBONACCI is $\Theta(n)$.
- as the results to all previous subproblems are stored into the array the memory consumption of the algorithm is $\Theta(n)$.

When should dynamic programming be used?

- the problem needs to have two properties:
  - the optimal solution can be given easily enough with the function of the optimal solutions to the subproblems:*optimal substructure*
  - the results to the subproblems can be used several times: *overlapping subproblems*
- for comparison: "memoization" requires only "overlapping subproblem"
  $\Rightarrow$ it can be used more generally than dynamic programming

- "memoization" is better also when the tabulation of dynamic programming is needed only for a small amount of the subproblems
  $\Rightarrow$ dynamic programming calculates subproblems needlessly
  - if all results to the subproblems are needed, dynamic programming is better than memoization by a constant coefficient
- dynamic programming makes certain improvements possible that cannot be done with memoization
  - reason: the order of the calculation of subresults is known and maybe it can even be effected
- if even "overlapping subproblems" doesn't apply, "divide and conquer" may be the best solution

Let's go through some of the properties of the optimal substructure:

- the substructure can be proven optimal in the following way:
  - let's assume that the optimal solution to the problem contains a non-optimal subresult
  - prove that replacing it with an optimal one would improve the soltution to the entire problem
    $\Rightarrow$ the solution wasn't optimal afterall $\Rightarrow$ contradiction
    $\Rightarrow$ the initial assumption was incorrect
    $\Rightarrow$ the substructure is optimal
- the problem can be split into subproblems in several ways
- all of them are usually not as suitable for dynamic programming
- the set of subproblems should be kept as small as possible
  - all subproblems must be calculated and stored into the array

$\Rightarrow$ it's worth testing the division plan by breaking the problem into subproblems and into their subproblems on 2...4 levels and see if the same subproblems start repeating

- this often clarify which subproblem needs to be taken into account when designing the array
- for example the nth Fibonacci number:

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &= F_{n-2} + F_{n-3} + F_{n-3} + F_{n-4} \\ &= F_{n-3} + F_{n-4} + F_{n-4} + F_{n-5} + \\ &\quad F_{n-4} + F_{n-5} + F_{n-5} + F_{n-6} \end{aligned}$$

$\Rightarrow$ same subproblems are clearly repeated and the result can be represented based on the subproblems solved earlier

Improving the solution:

- It is often easy to improve an algorithm implemented based on one principle once it has been developed
- dynamic programming algorithms can sometimes be improved by
  - changing the calculation order
  - destroying the subresults that are not going to be used again
- destroying futile subresults saves memory
- for example, when calculating Fibnacci numbers only the two latest results are needed at each point

$\Rightarrow$ instead of $F[0 \ldots n]$ it is enough to maintain a "window" of two subresults which is then slid over the array

```
MEMORY-SAVING-FIBONACCI(n)
1  if n = 0 then
2      return 0
3  previous := 0          (store the base cases)
4  newest := 1
5  for i := 2 to n do
6      x := newest        (calculate F_i and update the values )
7      newest := newest + previous
8      previous := x
9  return newest
```

- this is how most of you would have solved this in the first place!
- the running-time of the algorithm is still $\Theta(n)$, but the memory consumption is only $\Theta(1)$

1.3 Greedy algorithms

The most natural way to approach a problem is often the greedy algorithm.

- the idea of a *greedy algorithm* is to choose the best appearing choise at that moment and ignore the others completely

Example: making a timetable for a concert hall

- the owner of the concert hall has received n reservations
  - for simplicity we'll mark the reservations with $1, 2, \ldots, n$
  - the set of reservations are marked with S
- only one event can be held at a time in the hall
- each reservation i has a starting time s_i and an fininshing time f_i such that $f_i > s_i$
  - the finishing time indicates the time when the next event can start (the hall is empty and clean etc.)
- the rent of the hall is almost independent of the length of the event $\Rightarrow$ the goal is to book **as many events as possible** into the hall

- the algorithm:

```
GREEDY-ACTIVITY-SELECTOR(S, f)      (uses a queue Q)
1  ▷ sort the reservations so that f_1 ≤ f_2 ≤ ... ≤ f_n
2  n := S.length
3  Q.push(1)                        (push the first event into the queue)
4  j := 1
5  for i := 2 to n do               (on each step add the event which,)
6      if s_i ≥ f_j then            (starts as early as possible)
7          Q.push(i)
8          j := i
9  return Q
```

- the running-time
  - sorting $O(n \lg n)$
  - GREEDY-ACTIVITY-SELECTOR $\Theta(n)$, sillä $Q.Push()$ on $\Theta(1)$.

$\Rightarrow$ quite efficient

When does a greedy algorithm produce correct results?

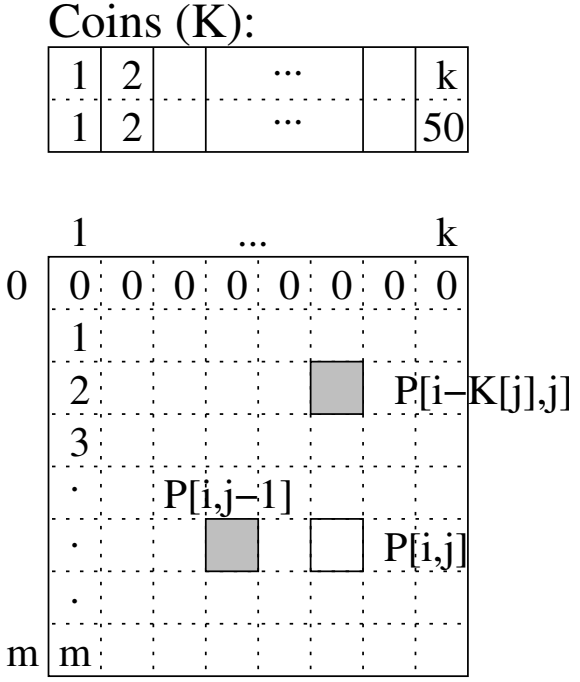
- there is no general test that could be used to test whether the greedy approach produces correct results or not
- a greedy algorithm is most likely good when the problem has two properties
  - *greedy choice property*
  - optimal substructure
- the optimal substructure was discussed with dynamic programming
  - it was a requirement also there
  - the event management problem has an optimal substructure: after choosing the first event the optimal solution to the rest of the problem combined with the solution to the first part is the optimal solution to the entire problem
- the greedy choise propertly is that the greedy choise may not divert the algorithm away from the global optimal solution

- a lot of effort and skill is sometimes needed to prove that the problem fullfills the greedy choise property
- sometimes at first glance the problem seems like is fullfills the greedy choise property
- Example: minimizing the amount of coins
  - Always choose the largest possible coin that still fits the sum
- The coin values in the kingdom of Poldavia are due to the monarchs superstition his lucky numbers

$\Rightarrow$ the greedy algorithm doesn't produce the correct result in Poldavia: the coins to the money sum 60 cannot be minimized with a greedy selection if the coins are 1, 20 and 50

- Solution: dynamic programming

- A dynamic programming algorithm:
  - First let's form the optimal solution recursively:
    - Mark the needed amount of coins with $P(m, k)$, where m is the sum of money and k the amount of available coins
    - The first base case is when there are only coins of one $\Rightarrow m$ coins are needed into the sum m $P(m, 1) = m$.
    - If the sum of money is zero no coins are needed, $P(0, k) = 0$, regardless of the amount of the coins available
    - Otherwise we assume that the possible coins are listed in an array K from largest to the smallest, the largest available coin is $K[k]$.



$\Rightarrow$ the task is divided into two depending on whether the sum is larger or smaller than $K[k]$.

- If the sum $m < K[k]$ the coin $K[k]$ cannot be in it and the amount of coins needed is $P(m, k) = P(m, k-1)$
- If $m \geq K[k]$, $K[k]$ can either be chosen into the sum or it can be left out
- If $K[k]$ is chosen in, the search for the solution if continued for the sum $m - K[k]$
- If the coin is not taken in, the solution is $P(m, k-1)$
- Since the amount of coins needs to be minimized the alternative that produces that smaller $P(m, k)$ is chosen, i.e. $P(m, k) = min(1 + P(m - K[k]), P(m, k-1))$.

- The minimization algorithm for the coins: for simplicity the algorithm assumes that the first index in the result array P can be also zero

```
MIN-COIN(m, k)
1   for i := 0 to m do
2       P[i, 1] := i                          (first base case)
3   for i := 1 to k do
4       P[0, i] := 0                          (second base case)
5   for i := 1 to m do                        (fill the array based on the recursive equation)
6       for j := 2 to k do                    (top to bottom, left to right)
7           if i < K[j] then
8               P[i, j] := P[i, j − 1]        (coin j too large)
9           else                              (chose the alternative that requires less coins)
10              P[i, j] := min(1 + P[i − K[j], j], P[i, j − 1])
```

- Running-time:
  - The result array is filled from the simple base cases towards the larger ones until the desired solution is reached
  - An element in the array is determined based on previously calculated results

  $\Rightarrow$ The array needs to be filled entirely

  $\Rightarrow$ Running-time and memory consumption $\Theta(km)$