



# *Advanced Algorithm Design and Analysis (Lecture 2)*

SW8 spring 2011

*Simonas Šaltenis*

3.2.12

*simas@cs.aau.dk*

Center for Data-intensive Systems

# Greedy Algorithms

---

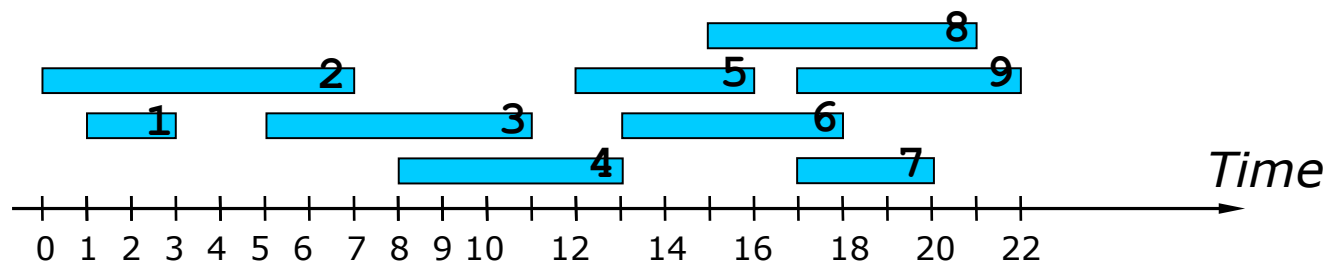


- Goals of the lecture:
  - *to understand the **principles** of the greedy algorithm design technique;*
  - *to understand the **example greedy algorithms** for activity selection and Huffman coding, to be able to **prove** that these algorithms find optimal solutions;*
  - *to be able to **apply** the greedy algorithm design technique.*

# Activity-Selection Problem



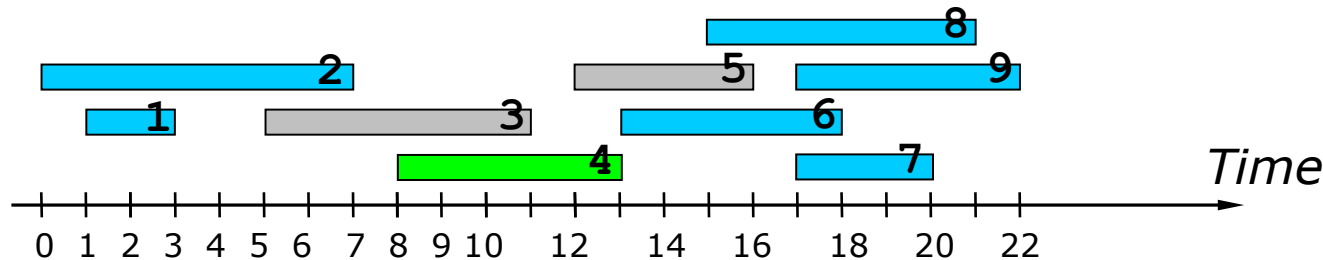
- Input:
  - A set of  $n$  activities, each with start and end times:  $A[i].s$  and  $A[i].f$ . The activity lasts during the period  $[A[i].s, A[i].f)$
- Output:
  - The **largest** subset of mutually *compatible* activities
    - ♦ Activities are compatible if their intervals do not intersect



# “Straight-forward” solution



- Let's just pick (schedule) one activity  $A[k]$ 
  - This generates two set's of activities compatible with it:  
 $Before(k)$ ,  $After(k)$ 
    - ♦ E.g.,  $Before(4) = \{1, 2\}$ ;  $After(4) = \{6, 7, 8, 9\}$



- Solution:

$$MaxN(A) = \begin{cases} 0 & \text{if } A = \emptyset, \\ \max_{a \in A} \{ MaxN(Before(a)) + MaxN(After(a)) + 1 \} & \text{if } A \neq \emptyset. \end{cases}$$

# Dynamic Programming Alg.



- The recurrence results in a dynamic programming algorithm
  - Sort activities on the start or end time (for simplicity assume also “sentinel” activities  $A[0]$  and  $A[n+1]$ )
  - Let  $S_{ij}$  – a set of activities after  $A[i]$  and before  $A[j]$  and compatible with  $A[i]$  and  $A[j]$ .
  - Let’s have a two-dimensional array, s.t.,  $c[i, j] = \text{MaxN}(S_{ij})$ :

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max_{i < k < j} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

- $\text{MaxN}(A) = \text{MaxN}(S_{0, n+1}) = c[0, n+1]$

# Dynamic Programming Alg. II

---



- Does it really work correctly?
  - We have to prove the optimal sub-structure:
    - ♦ *If an optimal solution  $A$  to  $S_{ij}$  includes  $A[k]$ , then solutions to  $S_{ik}$  and  $S_{kj}$  (as parts of  $A$ ) must be optimal as well*
    - ♦ To prove use “cut-and-paste” argument
- What is the running time of this algorithm?

# Greedy choice



- What if we could choose “the best” activity (as of now) and be sure that it belongs to an optimal solution
  - We wouldn’t have to check out all these sub-problems and consider all currently possible choices!
- Idea: Choose the activity that **finishes first!**
  - Then, solve *only one* sub-problem for the remaining compatible activities

```
MaxN(A[0..n], i) //returns a set of activities
01 m ← i + 1
02 while m ≤ n and A[m].s < A[i].f do
03     m ← m + 1
04 if m ≤ n then return {A[m]} ∪ MaxN(A, m)
05     else return ∅
```

# Greedy-choice property

---



- What is the running time of this algorithm?
- Does it find an optimal solution?:
  - We have to prove the *optimal sub-structure* property (we did that already)
  - We have to prove the *greedy-choice property*, i.e., that our locally optimal choice belongs to some globally optimal solution.
- The challenge is to choose the right interpretation of “the best choice”:
  - How about the activity that starts first
    - ♦ Show a *counter-example*

# Data Compression



- *Data compression* problem – strings  $S$  and  $S'$ :
  - $S \rightarrow S' \rightarrow S$ , such that  $|S'| < |S|$
- Text compression by coding with *variable-length* code:
  - Obvious idea – *assign short codes to frequent characters*:  
“**abracadabra**”

Frequency table:

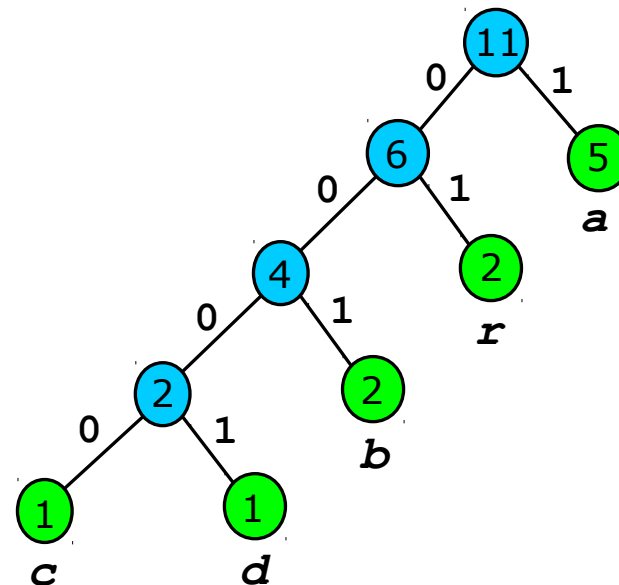
	<b>a</b>	<b>b</b>	<b>c</b>	<b>d</b>	<b>r</b>
Frequency	5	2	1	1	2
Fixed-length code	000	001	010	011	100
Variable-length code	1	001	0000	0001	01

- *How much do we save in this case?*

# Prefix code



- Optimal code for given frequencies:
  - Achieves the minimal length of the coded text
- *Prefix code*: no codeword is a prefix of another
  - It can be shown that optimal coding can be done with prefix code



- We can store all codewords in a *binary trie* – very easy to decode
  - Coded characters in leaves
  - Each node contains the sum of the frequencies of all descendants

# Optimal Code/Trie

---



- The *cost* of the coding trie  $T$ :

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

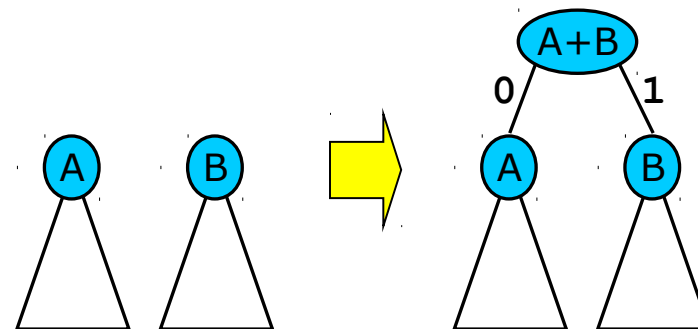
- $C$  – the alphabet,
  - $f(c)$  – frequency of character  $c$ ,
  - $d_T(c)$  – depth of  $c$  in the trie (length of code in bits)
- 
- Optimal trie – the one that minimizes  $B(T)$
  
  - Observation – optimal trie is always full:
    - Every non-leaf node has two children. Why?

# Huffman Algorithm - Idea



- Huffman algorithm, builds the code trie bottom up. Consider a forest of trees:

- Initially – one separate node for each character.
- In each step – join two trees into a larger tree



- Repeat this until one tree (trie) remains.
- Which trees to join? Greedy choice – the trees with the **smallest** frequencies!

# Huffman Algorithm

---



## **Huffman** (C)

```
01 Q.build(C) // Builds a min-priority queue on frequencies
02 for i ← 1 to n-1 do
03     Allocate new node z
04     x ← Q.extractMin()
05     y ← Q.extractMin()
06     z.setLeft(x) // corresponding to bit 0
07     z.setRight(y) // corresponding to bit 1
08     z.setF(x.f() + y.f())
09     Q.insert(z)
10 return Q.extractMin() // Return the root of the trie
```

- What is its running time?
- Run the algorithm on: “**oho ho, ole**”

# Correctness of Huffman

---



- Greedy choice property:
  - Let  $x, y$  – two characters with lowest frequencies. Then there exists an optimal prefix code where codewords for  $x$  and  $y$  have the same length and differ only in the last bit
  - Let's prove it:
    - ♦ Transform an optimal trie  $T$  into one ( $T''$ ), where  $x$  and  $y$  are max-depth siblings. Compare the costs.

# Correctness of Huffman

---



- Optimal sub-structure property:
  - Let  $x, y$  – characters with minimum frequency
  - $C' = C - \{x, y\} \cup \{z\}$ , such that  $f(z) = f(x) + f(y)$
  - Let  $T'$  be an optimal code trie for  $C'$
  - Replace leaf  $z$  in  $T'$  with internal node with two children  $x$  and  $y$
  - The result tree  $T$  is an optimal code trie for  $C$
- Proof a little bit more involved than a simple “cut-and-paste” argument

# Elements of Greedy Algorithms

---



- Greedy algorithms are used for optimization problems
  - A number of choices have to be made to arrive at an optimal solution
  - At each step, make the “locally best” choice, without considering all possible choices and solutions to sub-problems induced by these choices (compare to dynamic programming)
  - After the choice, only one sub-problem remains (smaller than the original)
- Greedy algorithms usually sort or use priority queues

# Elements of Greedy Algorithms

---



- First, one has to prove the *optimal sub-structure* property
  - the simple “cut-and-paste” argument may work
- The main challenge is to decide the interpretation of “the best” so that it leads to a global optimal solution, i.e., you can prove the *greedy choice property*
  - The proof is usually constructive: takes a hypothetical optimal solution without the specific greedy choice and transforms into one that has this greedy choice.
  - Or you find counter-examples demonstrating that your greedy choice does not lead to a global optimal solution.

# Other Greedy Algorithms

---



- Find a minimum spanning tree in a weighted graph
- Coin changing