



Theory of algorithms (9th lecture)

Pál Pusztai
pusztai@sze.hu

Outline

- Greedy algorithms
 - Greedy algorithms versus dynamic programming
 - Knapsack problems
 - An activity-selection problem
 - Designing a binary character code
 - Huffman code
 - Approximation algorithms
 - The set-covering problem
- Exercises



Greedy algorithms

■ Greedy algorithms versus dynamic programming

A **dynamic programming algorithm** makes a choice at each step and the choice depends on the solutions to subproblems. It works in a bottom-up manner, progressing from smaller subproblems to larger subproblems.

A **greedy algorithm** makes choice that seems best at the moment and then solve the subproblem that remains. It works in a top-down fashion, making one greedy choice after another, reducing each given problem instance to a smaller one.

- The choice may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems.
- This heuristic strategy does not always produce an optimal solution, but sometimes it does.

The **greedy-choice property**: a globally optimal solution can be reached by making locally optimal (greedy) choices.

The **optimal-substructure property**: an optimal solution to the problem contains within it optimal solutions to subproblems.



Greedy algorithms

■ Knapsack problems

The **0-1 knapsack problem**: A thief robbing a store finds n items. The i th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W . Which items should he take?

The **fractional knapsack problem**: the setup is the same, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.

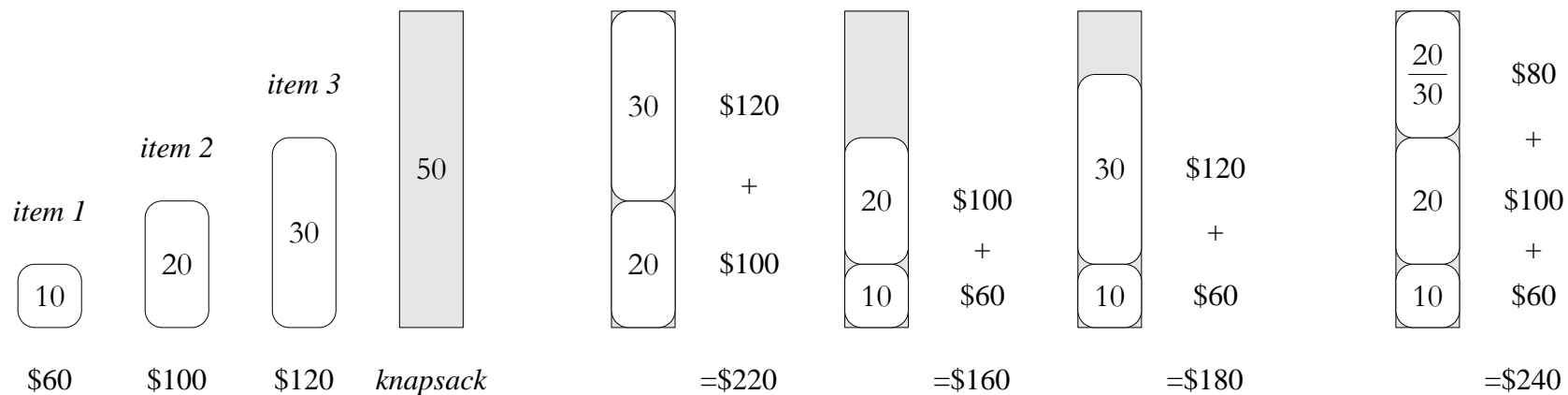
Both knapsack problems exhibit the optimal-substructure property.

The 0-1 knapsack problem does not exhibit the greedy-choice property, thus it can not be solved with greedy strategy, but it can be solved with dynamic programming.

The fractional knapsack problem exhibits the greedy-choice property, thus it can be solved with greedy strategy (with greedy choice of v_i / w_i).



Greedy algorithms



A knapsack problem

Greedy algorithms

■ An activity-selection problem

There is given a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed **activities** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time.

Each activity a_i has a **start time** s_i and a **finish time** f_i , where $0 \leq s_i < f_i < \infty$.

If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$.

Activities a_i and a_j are **compatible** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, if $s_i \geq f_j$ or $s_j \geq f_i$.

The **activity-selection problem**: to select a maximum-size subset of mutually compatible activities.

Greedy algorithms

GREEDY-ACTIVITY-SELECTOR(s, f)

```
1   $n = s.length$ 
2   $A = \{a_1\}$ 
3   $k = 1$ 
4  for  $m = 2$  to  $n$ 
5      if  $s[m] \geq f[k]$ 
6           $A = A \cup \{a_m\}$ 
7           $k = m$ 
8  return  $A$ 
```

Criteria: The data of activities are stored in s and f arrays and they are sorted in monotonically increasing order of finish time: $f_1 \leq f_2 \leq \dots \leq f_n$.

Remark: Since the activities are examined in order of monotonically increasing finish time, f_k is always the maximum finish time of any activity in A , that is, $f_k = \max\{f_i: a_i \in A\}$.

Efficiency: A set of n activities is scheduled in $\Theta(n)$ time.

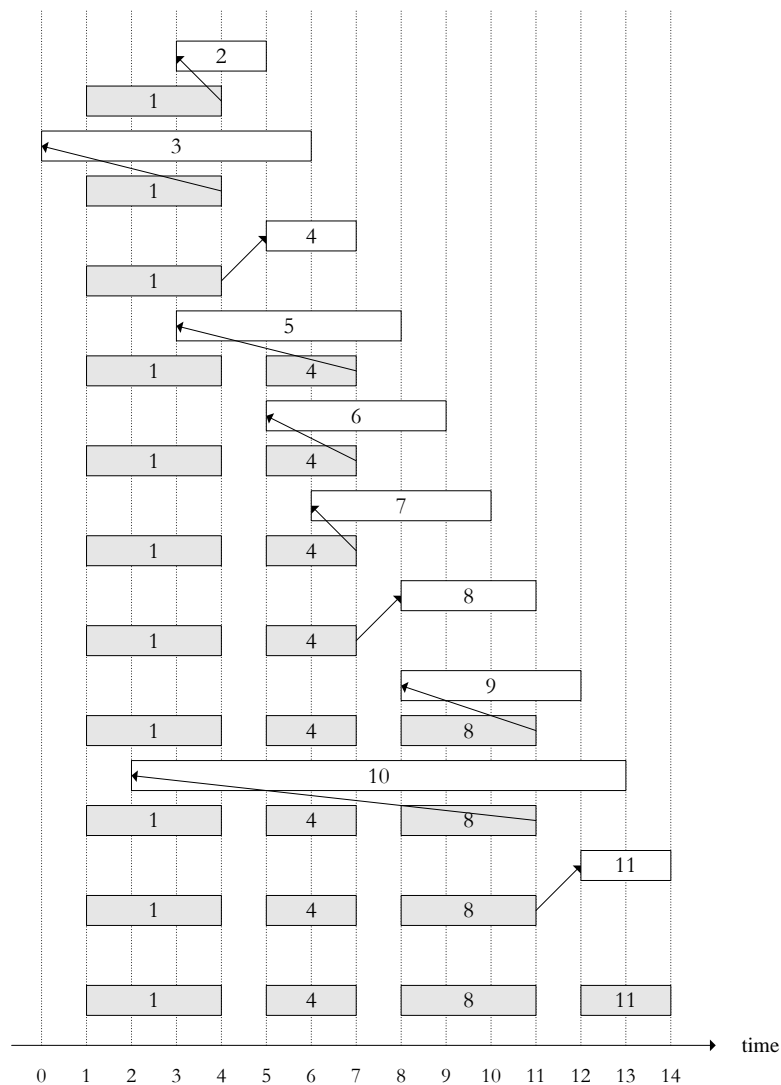
Theorem: Consider any nonempty subproblem S_k , and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k .

Corollary: The GREEDY-ACTIVITY-SELECTOR produces an optimal solution of the activity selection problem.



Greedy algorithms

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14



The operation of GREEDY-ACTIVITY-SELECTOR



Exercises

- What is the result of the GREEDY-ACTIVITY-SELECTOR if we have activities with below given s and f ?
 - $s = \langle 8, 3, 5, 12, 3, 6, 10, 6, 17 \rangle$
 - $f = \langle 12, 6, 8, 14, 7, 9, 15, 7, 20 \rangle$



Greedy algorithms

■ Designing a binary character code

How can a file of characters be stored compactly in which each character is represented by a unique binary string, which we call a **codeword**?

Prefix codes: a codes in which no codeword is also a prefix of some other codeword. The prefix codes are desirable because they **simplify** decoding. A prefix code can always achieve the **optimal** data compression among any character code.

Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file.

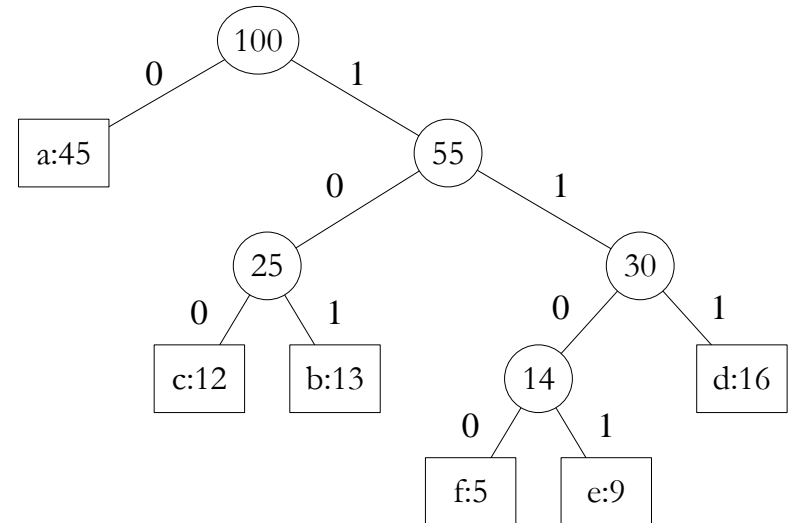
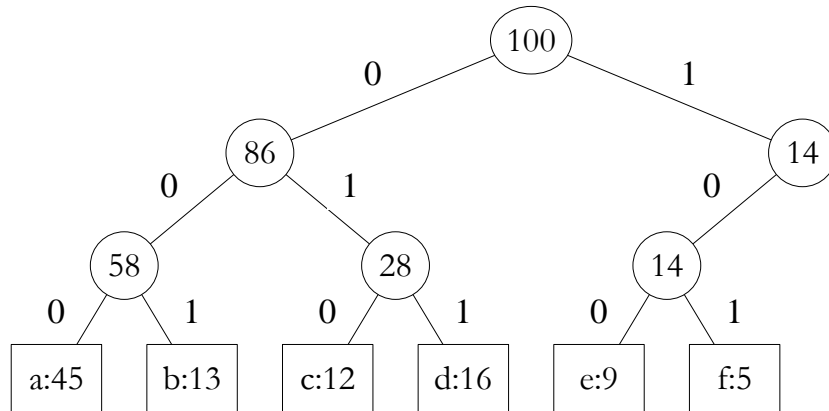
Decoding needs a convenient representation for the prefix code so that we can easily pick off the initial codeword. A binary tree whose leaves are the given characters provides one such representation. The codeword for a character is given by the simple path from the root to that character.

An **optimal code** for a file is always represented by a **full binary tree**, in which every nonleaf node has two children.

If C is the alphabet from which the characters are drawn and all character frequencies are positive, then the tree for an **optimal prefix code** has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C|-1$ internal nodes.

Greedy algorithms

	a	b	c	d	e	f	
Frequency (in thousands)	45	13	12	16	9	5	
Fixed-length codeword	000	001	010	011	100	101	(300,000 bits)
Variable-length codeword	0	101	100	111	1101	1100	(224,000 bits)



A character-coding problem

Greedy algorithms

HUFFMAN(C)

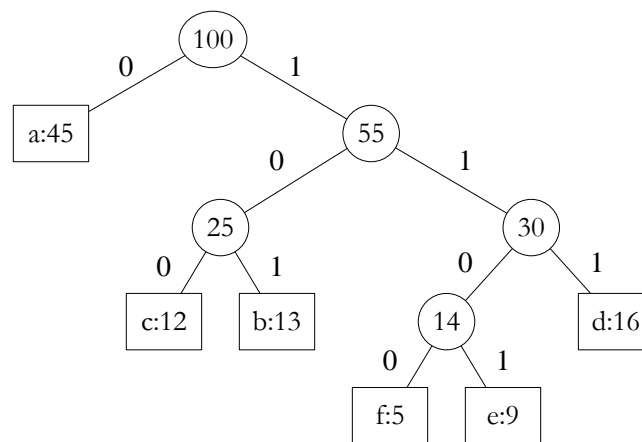
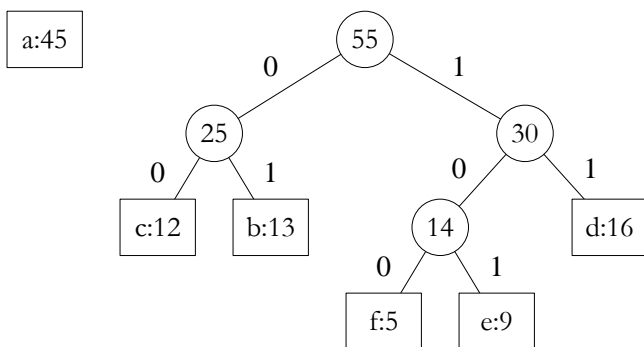
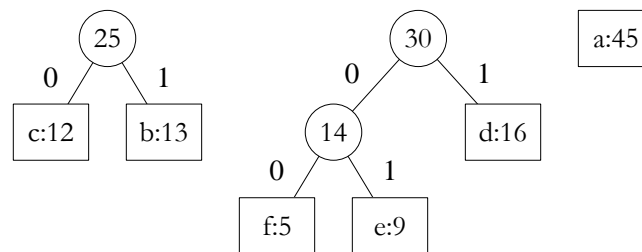
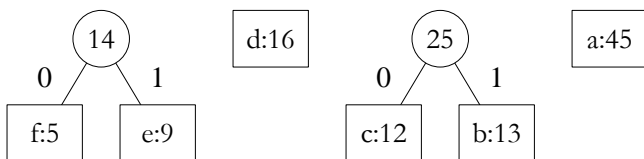
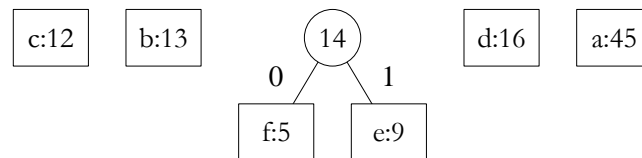
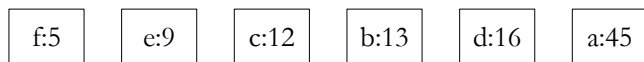
```
1   $n = |C|$ 
2   $Q = C$ 
3  for  $i = 1$  to  $n-1$ 
4      allocate a new node  $z$ 
5       $z.left = x = \text{EXTRACT-MIN}(Q)$ 
6       $z.right = y = \text{EXTRACT-MIN}(Q)$ 
7       $z.freq = x.freq + y.freq$ 
8       $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$            // return the root of the tree
```

Remark: Each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency.

Efficiency: If the min-priority queue Q is implemented as a binary min-heap, then each heap operation (EXTRACT-MIN, INSERT) requires time $O(\lg n)$. Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$.



Greedy algorithms



The operation of HUFFMAN



Exercises

- What tree is the result of HUFFMAN with the below given input data? Give the total number of bits that is necessary to code the entire file with the fixed-length code and Huffman code.
 - a: 23, b: 15, c: 12, d: 13, e: 6, f: 10, g: 4, h: 17
- What is an optimal Huffman code for the following set of frequencies, based on the first Fibonacci numbers:
 - a: 1, b: 1, c: 2, d: 3, e: 5, f: 8, g: 13, h: 21, ...



Greedy algorithms

■ Approximation algorithms

We call an algorithm that returns near-optimal solutions an **approximation algorithm**.

Suppose that each potential solution of an optimization problem has a positive cost, and we wish to find a near-optimal solution.

Depending on the problem, an optimal solution is one with **maximum** possible cost or one with **minimum** possible cost; that is, the problem may be either a maximization or a minimization problem.

We say that an algorithm for a problem has an **approximation ratio** of $\rho(n)$ if, for any input of size n , the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max(C/C^*, C^*/C) \leq \rho(n).$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a **$\rho(n)$ -approximation algorithm**.

Remark: if the $\rho(n)$ function is independent from n , then approximation ratio is ρ and the algorithm is called a ρ -approximation algorithm.

Greedy algorithms

■ The set-covering problem

An instance (X, \mathcal{F}) of the **set-covering problem** consists of a finite set X and a family \mathcal{F} of subsets of X , such that every element of X belongs to at least one subset in \mathcal{F} :

$$X = \bigcup_{S \in \mathcal{F}} S.$$

We say that a subset $S \in \mathcal{F}$ **covers** its elements.

The problem is to find a minimum size subset $\mathcal{C} \subseteq \mathcal{F}$ whose members cover all of X :

$$X = \bigcup_{S \in \mathcal{C}} S.$$

Example: Suppose that X represents a set of skills that are needed to solve a problem and that we have a given set of people available to work on the problem. We wish to form a committee, containing as few people as possible, such that for every requisite skill in X , at least one member of the committee has that skill.

Greedy algorithms

GREEDY-SET-COVER(X, \mathcal{F})

```
1   $U = X$ 
2   $\mathcal{C} = \emptyset$ 
3  while  $U \neq \emptyset$ 
4      select an  $S \in \mathcal{F}$  that maximizes  $|S \cap U|$ 
5       $U = U - S$ 
6       $\mathcal{C} = \mathcal{C} \cup \{S\}$ 
7  return  $\mathcal{C}$ 
```

Efficiency: Since the iteration runs at most $\min(|X|, |\mathcal{F}|)$ times and the loop body can be implemented in time $O(|X||\mathcal{F}|)$, a simple implementation runs in time $O(|X||\mathcal{F}| \min(|X|, |\mathcal{F}|))$.

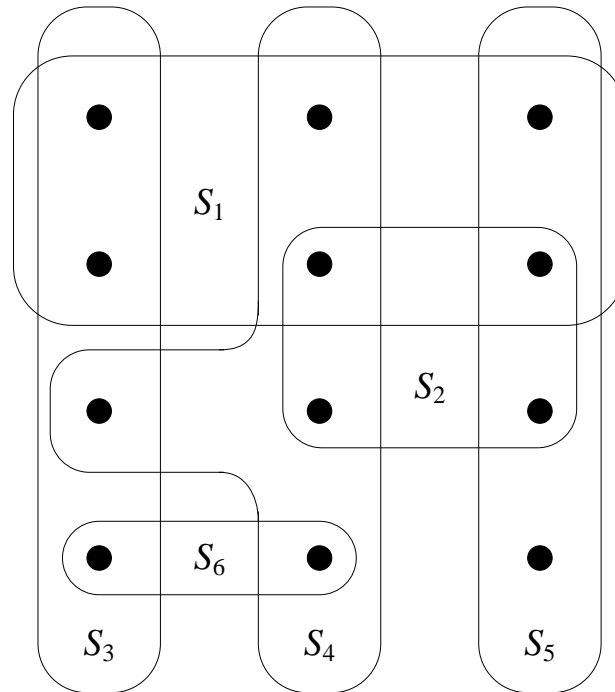
Let us denote the d th harmonic number $H(d) = \sum_{i=1, \dots, d} 1/i$.

Theorem: GREEDY-SET-COVER is a polynomial-time $\rho(n)$ -approximation algorithm, where

$$\rho(n) = H(\max\{|S| : S \in \mathcal{F}\}).$$

Corollary: Since $\sum_{i=1, \dots, d} 1/i \leq \ln d + 1$, GREEDY-SET-COVER is a polynomial-time $(\ln |X| + 1)$ -approximation algorithm.

Greedy algorithms



An instance of the set-covering problem

Exercises

- What is the result of GREEDY-SET-COVER with the sets of the previous slide?
- Consider each of the below given words as a set of letters. Give the result of GREEDY-SET-COVER if the ties are broken in favor of the word that appears first in the dictionary.
 - {arid, dash, drain, heard, lost, nose, shun, slate, snare, thread}

