

Computer Science & Engineering 423/823  
Design and Analysis of Algorithms  
Lecture 04 — Greedy Algorithms (Chapter 16)

Stephen Scott and Vinodchandran N. Variyam

sscott@cse.unl.edu

## Introduction

- ▶ Greedy methods: A technique for solving **optimization problems**
  - ▶ Choose a solution to a problem that is best per an objective function
- ▶ Similar to dynamic programming in that we examine subproblems, exploiting optimal substructure property
- ▶ Key difference: In dynamic programming we considered **all** possible subproblems
- ▶ In contrast, a greedy algorithm at each step commits to just one subproblem, which results in its **greedy choice** (locally optimal choice)
- ▶ Examples: Minimum spanning tree, single-source shortest paths

## Activity Selection (1)

- ▶ Consider the problem of scheduling classes in a classroom
- ▶ Many courses are candidates to be scheduled in that room, but not all can have it (can't hold two courses at once)
- ▶ Want to maximize utilization of the room in terms of number of classes scheduled
- ▶ This is an example of the **activity selection problem**:
  - ▶ Given: Set  $S = \{a_1, a_2, \dots, a_n\}$  of  $n$  proposed activities that wish to use a resource that can serve only one activity at a time
  - ▶  $a_i$  has a **start time**  $s_i$  and a **finish time**  $f_i$ ,  $0 \leq s_i < f_i < \infty$
  - ▶ If  $a_i$  is scheduled to use the resource, it occupies it during the interval  $[s_i, f_i) \Rightarrow$  can schedule both  $a_i$  and  $a_j$  iff  $s_j \geq f_i$  or  $s_j \leq f_i$  (if this happens, then we say that  $a_i$  and  $a_j$  are **compatible**)
  - ▶ Goal is to find a largest subset  $S' \subseteq S$  such that all activities in  $S'$  are pairwise compatible
  - ▶ Assume that activities are sorted by finish time:

$$f_1 \leq f_2 \leq \dots \leq f_n$$

## Activity Selection (2)

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	9	9	10	11	12	14	16

Sets of mutually compatible activities:  $\{a_3, a_9, a_{11}\}$ ,  $\{a_1, a_4, a_8, a_{11}\}$ ,  $\{a_2, a_4, a_9, a_{11}\}$

## Optimal Substructure of Activity Selection

- ▶ Let  $S_{ij}$  be set of activities that start after  $a_i$  finishes and that finish before  $a_j$  starts
- ▶ Let  $A_{ij} \subseteq S_{ij}$  be a largest set of activities that are mutually compatible
- ▶ If activity  $a_k \in A_{ij}$ , then we get two subproblems:  $S_{ik}$  (subset starting after  $a_i$  finishes and finishing before  $a_k$  starts) and  $S_{kj}$
- ▶ If we extract from  $A_{ij}$  its set of activities from  $S_{ik}$ , we get  $A_{ik} = A_{ij} \cap S_{ik}$ , which is an optimal solution to  $S_{ik}$ 
  - ▶ If it weren't, then we could take the better solution to  $S_{ik}$  (call it  $A'_{ik}$ ) and plug its tasks into  $A_{ij}$  and get a better solution
  - ▶ Works because subproblem  $S_{ik}$  independent from  $S_{kj}$
- ▶ Thus if we pick an activity  $a_k$  to be in an optimal solution and then solve the subproblems, our optimal solution is  $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$ , which is of size  $|A_{ik}| + |A_{kj}| + 1$

## Optimal Substructure Example

$i$	1	2	3	4	5	6	7	8	9	10	11
$s_i$	1	3	0	5	3	5	6	8	8	2	12
$f_i$	4	5	6	7	9	9	10	11	12	14	16

- ▶ Let<sup>1</sup>  $S_{ij} = S_{1,11} = \{a_1, \dots, a_{11}\}$  and  $A_{ij} = A_{1,11} = \{a_1, a_4, a_8, a_{11}\}$
- ▶ For  $a_k = a_8$ , get  $S_{1k} = S_{1,8} = \{a_1, a_2, a_3, a_4\}$  and  $S_{8,11} = \{a_{11}\}$
- ▶  $A_{1,8} = A_{1,11} \cap S_{1,8} = \{a_1, a_4\}$ , which is optimal for  $S_{1,8}$
- ▶  $A_{8,11} = A_{1,11} \cap S_{8,11} = \{a_{11}\}$ , which is optimal for  $S_{8,11}$

<sup>1</sup>Left-hand boundary condition addressed by adding to  $S$  activity  $a_0$  with  $f_0 = 0$  and setting  $i = 0$

## Recursive Definition

- Let  $c[i, j]$  be the size of an optimal solution to  $S_{ij}$

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

- In dynamic programming, we need to try all  $a_k$  since we don't know which one is the best choice...
- ...or do we?

## Greedy Choice

- What if, instead of trying all activities  $a_k$ , we simply chose the one with the earliest finish time of all those still compatible with the scheduled ones?
- This is a **greedy choice** in that it maximizes the amount of time left over to schedule other activities
- Let  $S_k = \{a_j \in S : s_j \geq f_k\}$  be set of activities that start after  $a_k$  finishes
- If we greedily choose  $a_1$  first (with earliest finish time), then  $S_1$  is the only subproblem to solve

## Greedy Choice (2)

- Theorem:** Consider any nonempty subproblem  $S_k$  and let  $a_m$  be an activity in  $S_k$  with earliest finish time. Then  $a_m$  is in some maximum-size subset of mutually compatible activities of  $S_k$
- Proof (by construction):**
  - Let  $A_k$  be an optimal solution to  $S_k$  and let  $a_j$  have earliest finish time of all in  $A_k$
  - If  $a_j = a_m$ , we're done
  - If  $a_j \neq a_m$ , then define  $A'_k = A_k \setminus \{a_j\} \cup \{a_m\}$
  - Activities in  $A'_k$  are mutually compatible since those in  $A$  are mutually compatible and  $f_m \leq f_j$
  - Since  $|A'_k| = |A_k|$ , we get that  $A'_k$  is a maximum-size subset of mutually compatible activities of  $S_k$  that includes  $a_m$   $\square$
- What this means is that **there exists** an optimal solution that uses the greedy choice

## Greedy-Activity-Selector( $s, f, n$ )

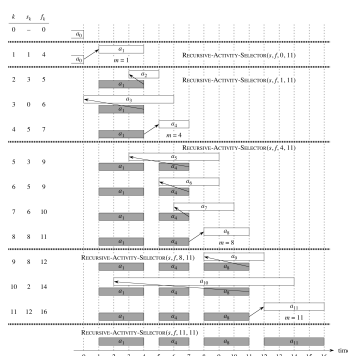
```

1  A = {a1}
2  k = 1
3  for m = 2 to n do
4      if s[m] ≥ f[k] then
5          A = A ∪ {am}
6          k = m
7
8  end
9  return A

```

What is the time complexity?

## Example

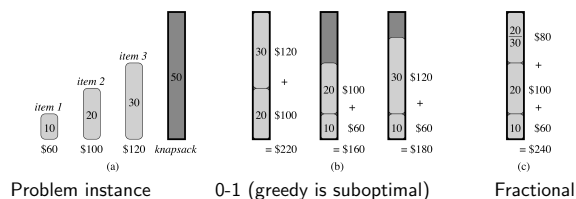


## Greedy vs Dynamic Programming (1)

- Like with dynamic programming, greedy leverages a problem's **optimal substructure property**
- When can we get away with a greedy algorithm instead of DP?
- When we can argue that the **greedy choice** is part of an optimal solution, implying that we need not explore all subproblems
- Example: The **knapsack problem**
  - There are  $n$  items that a thief can steal, item  $i$  weighing  $w_i$  pounds and worth  $v_i$  dollars
  - The thief's goal is to steal a set of items weighing at most  $W$  pounds and maximizes total value
  - In the **0-1 knapsack problem**, each item must be taken in its entirety (e.g., gold bars)
  - In the **fractional knapsack problem**, the thief can take part of an item and get a proportional amount of its value (e.g., gold dust)

## Greedy vs Dynamic Programming (2)

- There's a greedy algorithm for the fractional knapsack problem
  - Sort the items by  $v_i/w_i$  and choose the items in descending order
  - Has greedy choice property, since any optimal solution lacking the greedy choice can have the greedy choice swapped in
    - Works because one can always completely fill the knapsack at the last step
- Greedy strategy does not work for 0-1 knapsack, but do have  $O(nW)$ -time dynamic programming algorithm
  - Note that time complexity is *pseudopolynomial*
  - Decision problem is NP-complete



## Huffman Coding

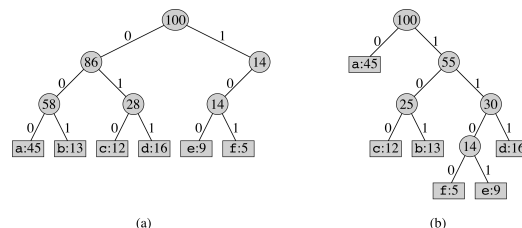
- Interested in encoding a file of symbols from some alphabet
- Want to minimize the size of the file, based on the frequencies of the symbols
- A **fixed-length code** uses  $\lceil \log_2 n \rceil$  bits per symbol, where  $n$  is the size of the alphabet  $C$
- A **variable-length code** uses fewer bits for more frequent symbols

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

Fixed-length code uses 300k bits, variable-length uses 224k bits

## Huffman Coding (2)

Can represent any encoding as a binary tree



If  $c.freq$  = frequency of codeword and  $d_T(c)$  = depth, cost of tree  $T$  is

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

## Algorithm for Optimal Codes

- Can get an optimal code by finding an appropriate **prefix code**, where no codeword is a prefix of another
- Optimal code also corresponds to a full binary tree
- Huffman's algorithm builds an optimal code by greedily building its tree
- Given alphabet  $C$  (which corresponds to leaves), find the two least frequent ones, merge them into a subtree
- Frequency of new subtree is the sum of the frequencies of its children
- Then add the subtree back into the set for future consideration

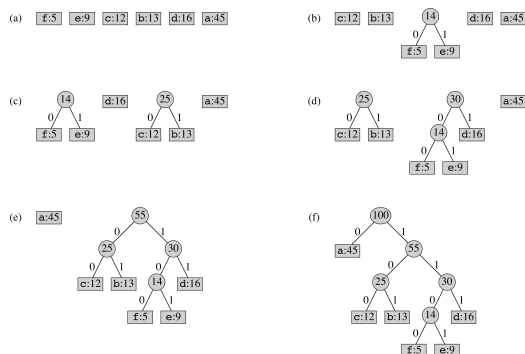
## Huffman(C)

```

1  n = |C|
2  Q = C           // min-priority queue
3  for i = 1 to n - 1 do
4      allocate node z
5      z.left = x = EXTRACT-MIN(Q)
6      z.right = y = EXTRACT-MIN(Q)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)
9  end
10 return EXTRACT-MIN(Q) // return root
    
```

Time complexity:  $n - 1$  iterations,  $O(\log n)$  time per iteration, total  $O(n \log n)$

## Huffman Example



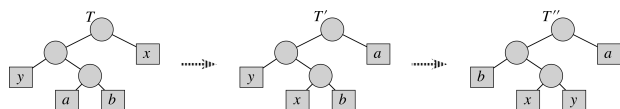
19/24

## Optimal Coding Has Greedy Choice Property (1)

- **Lemma:** Let  $C$  be an alphabet in which symbol  $c \in C$  has frequency  $c.freq$  and let  $x, y \in C$  have lowest frequencies. Then there exists an optimal prefix code for  $C$  in which codewords for  $x$  and  $y$  have the same length and differ only in the last bit.
  - I.e., an optimal solution exists that merges lowest frequencies first
- **Proof:** Let  $T$  be a tree representing an arbitrary optimal prefix code, and let  $a$  and  $b$  be siblings of maximum depth in  $T$ 
  - Assume, w.l.o.g., that  $x.freq \leq y.freq$  and  $a.freq \leq b.freq$
  - Since  $x$  and  $y$  are the two least frequent nodes, we get  $x.freq \leq a.freq$  and  $y.freq \leq b.freq$
  - Convert  $T$  to  $T'$  by exchanging  $a$  and  $x$ , then convert to  $T''$  by exchanging  $b$  and  $y$
  - In  $T''$ ,  $x$  and  $y$  are siblings of maximum depth

20/24

## Optimal Coding Has Greedy Choice Property (2)



Is  $T''$  optimal?

21/24

## Optimal Coding Has Greedy Choice Property (3)

Cost difference between  $T$  and  $T'$  is  $B(T) - B(T')$ :

$$\begin{aligned}
 &= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
 &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
 &= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
 &= (a.freq - x.freq)(d_T(a) - d_T(x)) \geq 0
 \end{aligned}$$

since  $a.freq \geq x.freq$  and  $d_T(a) \geq d_T(x)$

Similarly,  $B(T') - B(T'') \geq 0$ , so  $B(T'') \leq B(T)$ , so  $T''$  is optimal  $\square$

22/24

## Optimal Coding Has Optimal Substructure Property (1)

- **Lemma:** Let  $C$  be an alphabet in which symbol  $c \in C$  has frequency  $c.freq$  and let  $x, y \in C$  have lowest frequencies. Let  $C' = C \setminus \{x, y\} \cup \{z\}$  and  $z.freq = x.freq + y.freq$ . Let  $T'$  be any tree representing an optimal prefix code for  $C'$ . Then  $T$ , which is  $T'$  with leaf  $z$  replaced by internal node with children  $x$  and  $y$ , represents an optimal prefix code for  $C$

- **Proof:** Since  $d_T(x) = d_T(y) = d_{T'}(z) + 1$ ,

$$\begin{aligned}
 &x.freq \cdot d_T(x) + y.freq \cdot d_T(y) \\
 &= (x.freq + y.freq)(d_{T'}(z) + 1) \\
 &= z.freq \cdot d_{T'}(z) + (x.freq + y.freq)
 \end{aligned}$$

Also, since  $d_T(c) = d_{T'}(c)$  for all  $c \in C \setminus \{x, y\}$ ,

$$B(T) = B(T') + x.freq + y.freq \text{ and}$$

$$B(T') = B(T) - x.freq - y.freq$$

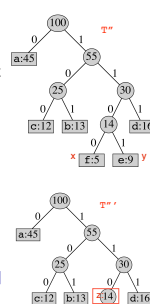
23/24

## Optimal Coding Has Optimal Substructure Property (2)

- Assume that  $T$  is not optimal, i.e.,  $B(T'') < B(T)$  for some  $T''$
- Assume w.l.o.g. (based on greedy choice lemma) that  $x$  and  $y$  are siblings in  $T''$
- In  $T''$ , replace  $x$ ,  $y$ , and their parent with  $z$  such that  $z.freq = x.freq + y.freq$ , to get  $T'''$ :

$$\begin{aligned}
 B(T''') &= B(T'') - x.freq - y.freq && \text{(prev. slide)} \\
 &< B(T) - x.freq - y.freq && \text{(subopt assumption)} \\
 &= B(T') && \text{(prev. slide)}
 \end{aligned}$$

- Contradicts assumption that  $T'$  is optimal for  $C'$   $\square$



24/24