Greedy Algorithms

Mohammad Javad Dousti

Changelog

- Rev. 1
  - Changed f_* to f_z (slide 25.)

Overview

- Introduction
- Fractional knapsack problem
- Activity-selection problem
- Huffman codes
- Sample problems

Introduction

- Greedy algorithms, like dynamic programming, used to solve optimization problems.
  - Dynamic programming is powerful but yet overkill for certain problems.
- Problems of interest must exhibit
  - Optimal substructure (like dynamic programming).
  - The **greedy-choice** property.
    - When we have a choice to make, make the one that looks best *right now*.
    - Make a **locally optimal choice** in hope of getting a **globally optimal solution**.

- Clearly, greedy algorithms don't always yield optimal solutions.
  - …but for many problems they do ☺

Greedy Algorithms

Fractional Knapsack Problem

Fractional Knapsack Problem

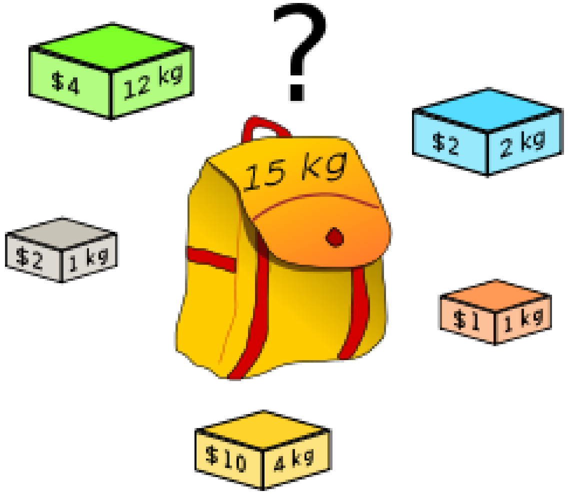
- Given n objects, with weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$.

- **0-1 knapsack problem:** Objects can be taken as a whole.

$$\text{Maximize } \sum_{i=1}^{n} x_i v_i$$

$$\text{subject to } \sum_{i=1}^{n} x_i w_i \leq W \text{ and } x_i \in \{0, 1\}$$

  - Solution: Dynamic programming. We have seen it before.



- **Fractional knapsack problem:** Objects can be taken partially.

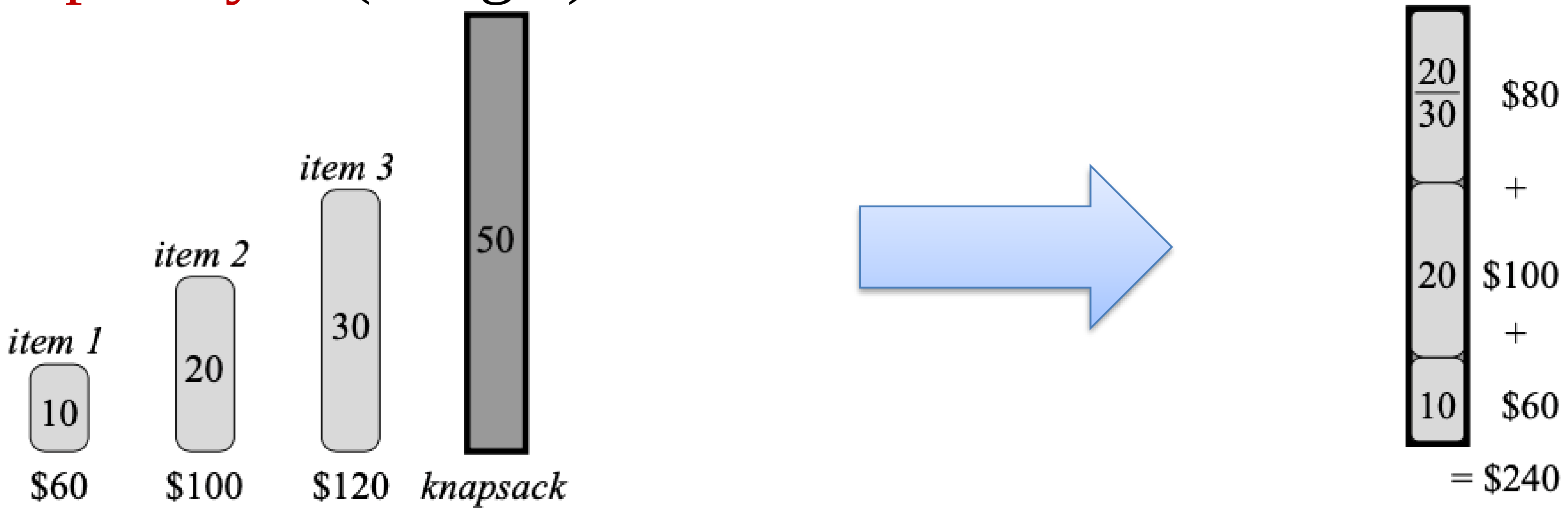
$$\text{Maximize } \sum_{i=1}^{n} x_i v_i$$

$$\text{subject to } \sum_{i=1}^{n} x_i w_i \leq W \text{ and } 0 \leq x_i \leq 1$$

  - Solution: Greedy algorithm. Let's see how it works.

A Greedy Solution

- In every step, take as much as possible of any remaining object with the highest $\frac{v_i}{w_i}$.
- Greedy choice: Taking an object with the highest $\frac{v_i}{w_i}$.
- Runtime complexity: $O(n \log n)$
- Example:



But does it yield an optimum solution?

Proof of Optimality

- Assume object indices are sorted with decreasing order by $\frac{v_i}{w_i}$.
- Suppose by contradiction, $ALG=\{p_1, p_2, \ldots, p_n\}$ is a greedy algorithm which isn't optimal, whereas $OPT=\{q_1, q_2, \ldots, q_n\}$ is. In other words, $\sum_{i=1}^{n} p_i v_i < \sum_{i=1}^{n} q_i v_i$.
- Let i be the smallest index where $p_i \neq q_i$.
  - Greedy choice tell us than $p_i > q_i$.
  - By the optimality of OPT, there exists a j where $p_j < q_j$.
- Now consider another solution $OPT'=\{q'_1, q'_2, \ldots, q'_n\}$, where $q'_k = q_k$ for all $k \neq i,j$.
  - *OPT'* takes a little more of item i and a little less of item j compared to OPT:
    - $q'_i = q_i + \varepsilon$ ($q'_i \leq 1$) and $q'_j = q_j - \varepsilon\, w_i/w_j$. The total weight remains the same: $\sum_{i=1}^{n} q'_i w_i = \sum_{i=1}^{n} q_i w_i$.
    - Total value increases though: $\sum_{i=1}^{n} q_i' v_i = \sum_{i=1}^{n} q_i v_i + \varepsilon v_i - \varepsilon \frac{w_i}{w_j} v_j > \sum_{i=1}^{n} q_i v_i$.
- *OPT'* is a better solution than *OPT*, which is a contradiction. Hence, *ALG* is optimum.

A Way to Prove Optimality of Greedy Algorithms

- Use contradiction.
- Find the most similar solution to the greedy solution.
  - If it’s identical to the greedy solution, the problem is solved.
  - If it is different than the greedy solution, make another solution, which is not worse than the optimal solution but is more similar to the greedy solution.
    - Use this new solution arrive reach a contradiction.

Wrap Up

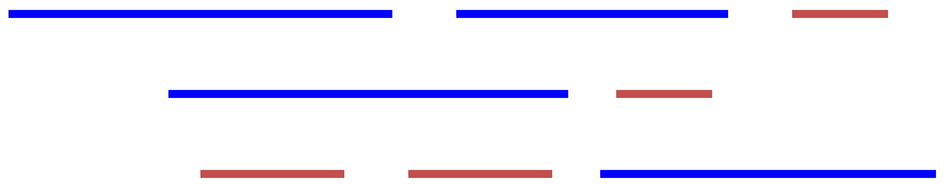
- Define the greedy choice.
- Prove that the greedy choice is optimal.
- Due to the optimal subproblem property, the remaining smaller problem is still similar to the original problem. Hence, using recursing and the greedy choice, one can solve the entire problem.
  - Usually, it’s easy (and recommended) to convert a recursive greedy algorithm to iterative.

Greedy Algorithms

Activity-Selection Problem

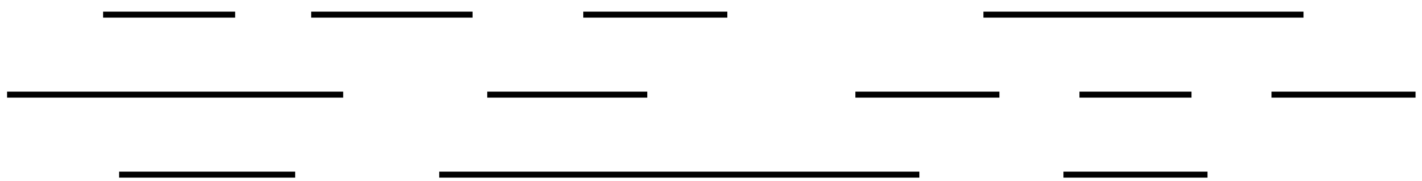
Activity-Selection Problem

- Given: n activities with start and finish times denoted by $[s_i, f_i]$.
- Problem: Find a maximum set of compatible activities. Activity i and j are compatible if they don't overlap, i.e., either $f_i < s_j$ or $f_j < s_i$.
- Example:

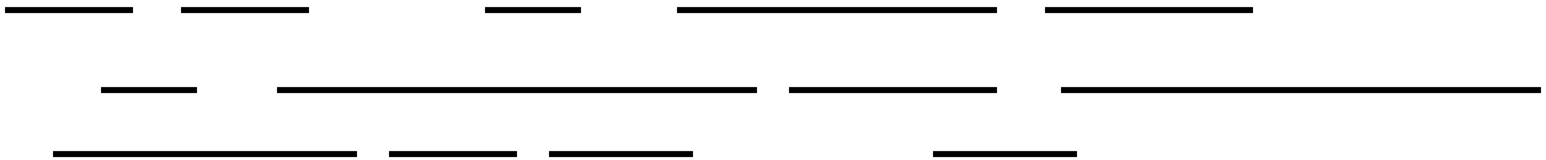


Greedy Choices: A Few Examples

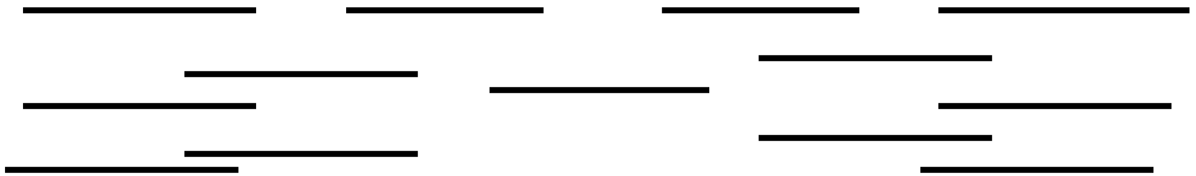
- Schedule earliest starting task



- Schedule shortest available task

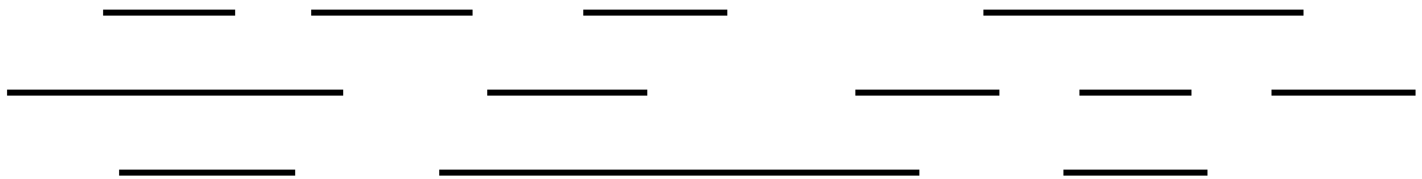


- Schedule task with fewest conflicting tasks

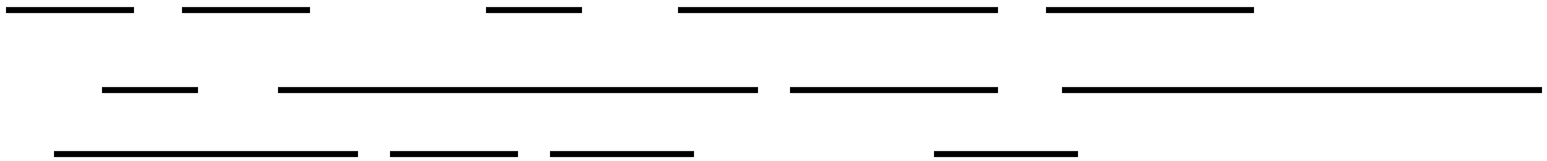


Greedy Choices: Earliest Finish Time

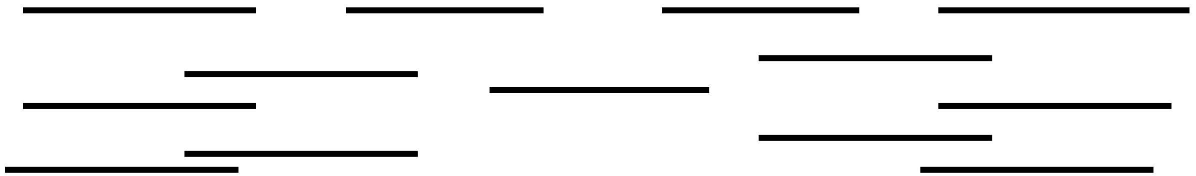
- Example 1



- Example 2



- Example 3



Greedy Algorithm

- Runtime complexity: $O(n)$
  - Considering the sort runtime complexity: $O(n \log n)$

```
ActivitySelector(s, f) {
    n = s.length
    A = {a_1}
    k = 1
    for m = 2 to n {
        if s[m] ≥ f[k] {
            A = A ∪ {a_m}
            k = m
        }
    }
    return A
}
```

Proof of Optimality

- Suppose $A=\{a_1, a_2, \ldots, a_p\}$ be a greedy solution, $B=\{b_1, b_2, \ldots, b_q\}$ be the maximum-size subset of mutually compatible activities which , and $|A| < |B|$.
  - Among all optimum solutions, B is chosen such that it has the most common intervals with A.
  - Both A and B activities are sorted in increasing order of finish time.
- a_i and b_i are the first different activities in A and B.
- One can create B' as follows: $B'=B-\{b_i\}\cup\{a_i\}$.
  - B' is still a set of compatible activities because all activities would start after b_i finishes which happens after a_i finishes.
  - $|B'|=|B|$
- This process can continue until B becomes identical to A, i.e., $|A| = |B|$, which is a contradiction. Hence, the greedy solution is optimum.

Huffman Codes

Some slides are courtesy of Dr. Mahini.

Lossless Data Compression (1)

File

ABAA
CDAA
DDCC

12 characters

A = 01010000
B = 01010001
C = 01110011
D = 01011111

Encoding

Stored File

01010000 01010001 01010000 01010000
01110011 01011111 01010000 01010000
01011111 01011111 01110011 01110011

96 bits

Lossless Data Compression (2)

File

ABAA
CDAA
DDCC

12 characters

A new compression with 23 bits

A = 0
B = 110
C = 10
D = 111

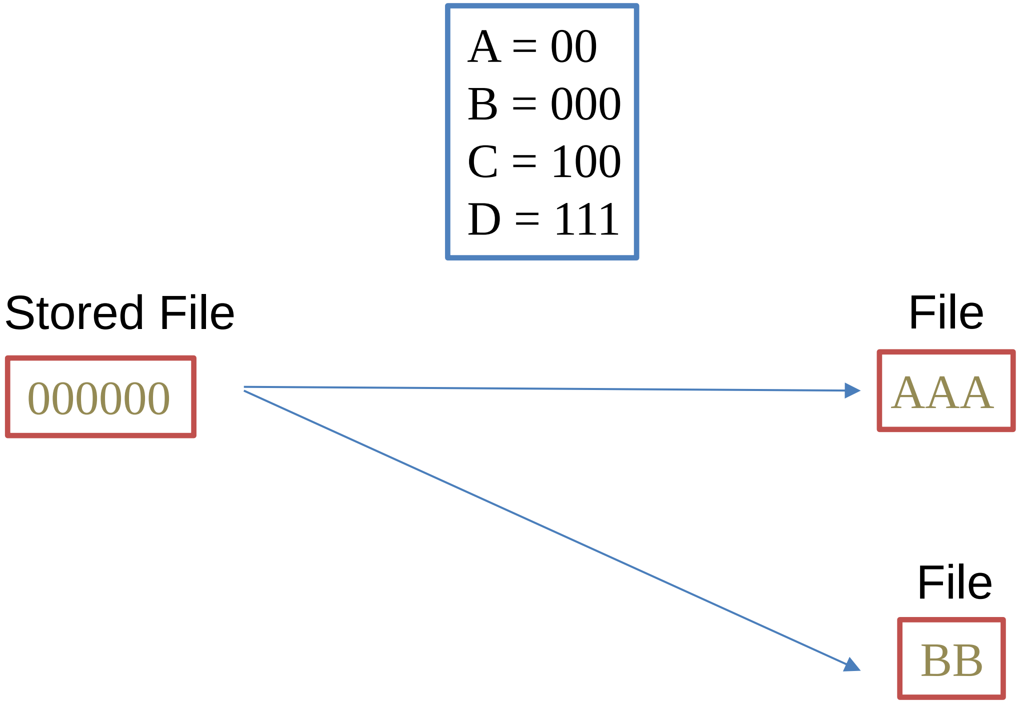
Encoding

Stored File

0 110 0 0
10 111 0 0
111 111 10 10

23 bits

What is lossless data compression?



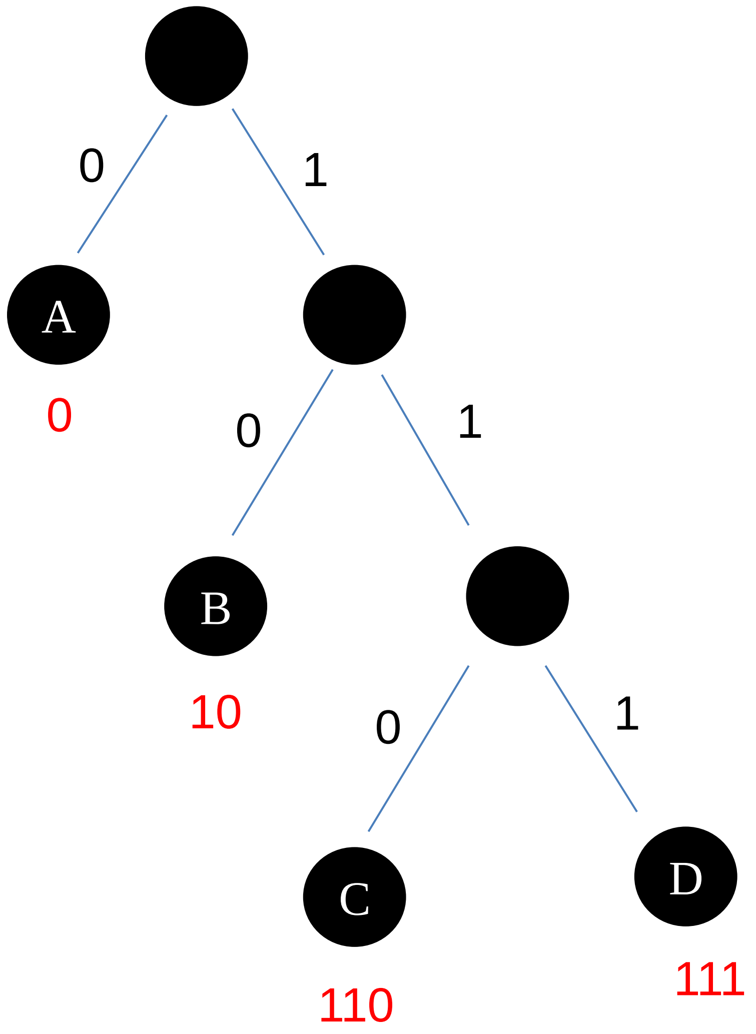
No code should be the prefix of the other one

Lossless Data Compression: Problem

Input: a file with n characters such that the frequency of character i is f_i

Goal: Assign code c_i with length of h_i to character i to minimize $\sum h_i \times f_i$ (length of the stored file) such that no code is the prefix of the other one.

You can model "prefix codes" with a binary tree

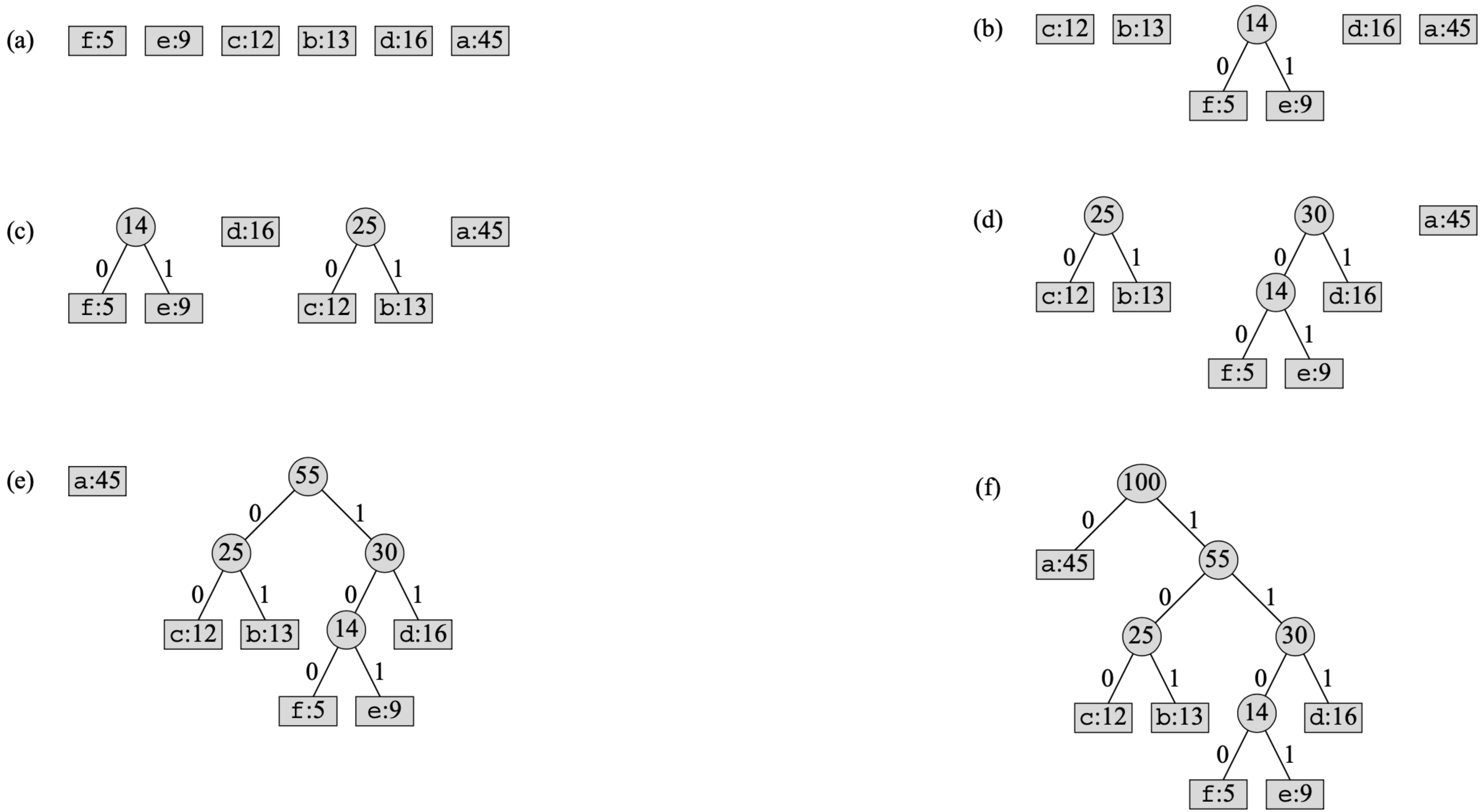


Huffman Codes: A Greedy Algorithm

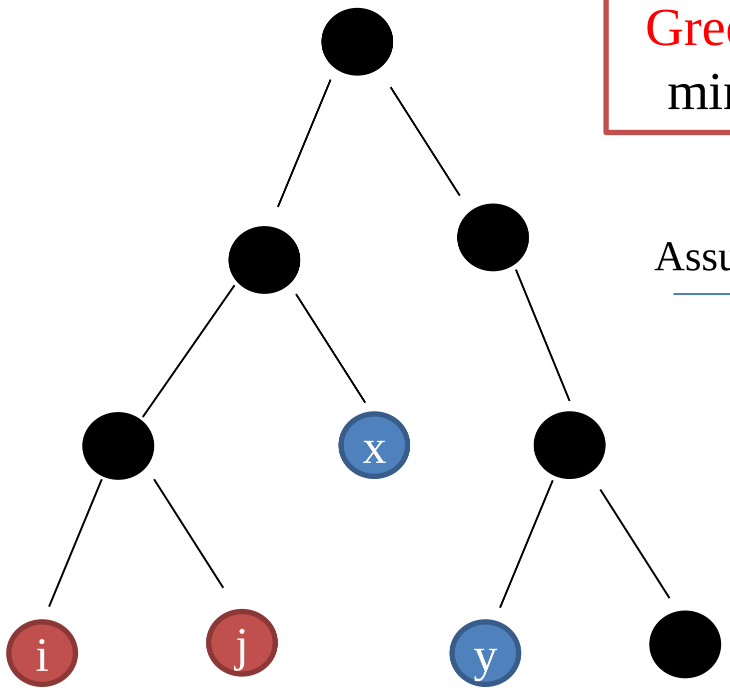
- Runtime complexity: $O(n \log n)$
- Note that the heap doesn't correspond to the Huffman tree.

```
HuffmanCode(C) {
    n = |C|
    Q = C   // Q is a binary min-heap
    for i = 1 to n - 1 {
        allocate a new node z
        z.left = x = Extract-Min(Q)
        z.right = y = Extract-Min(Q)
        z.freq = x.freq + y.freq
        Insert(Q, z)
    }
    return Extract-Min(Q)
}
```

Huffman Codes: Example

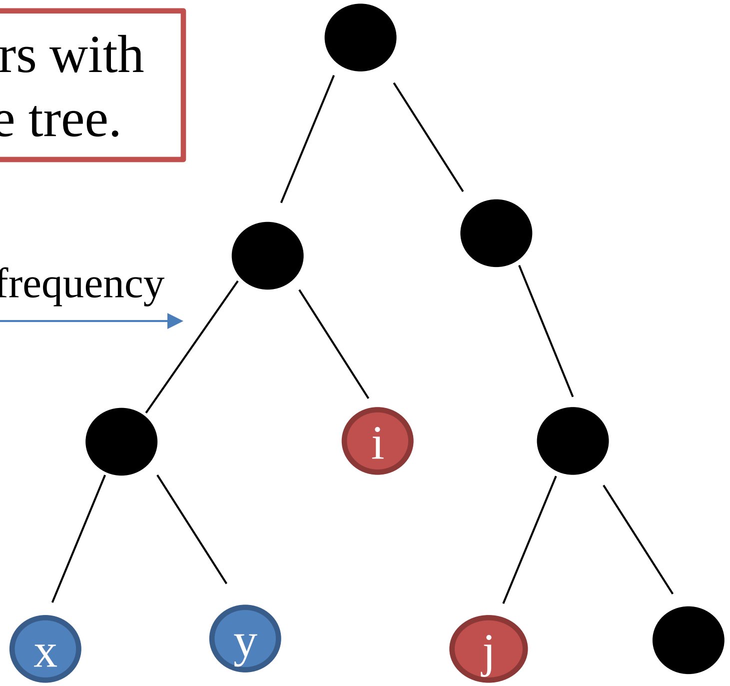


Greedy Choice



Greedy choice: The two characters with min frequency are siblings in the tree.

Assume x and y are characters with min frequency



$$h_i = h_j = h$$
$$h_x = h'$$
$$h_y = h''$$

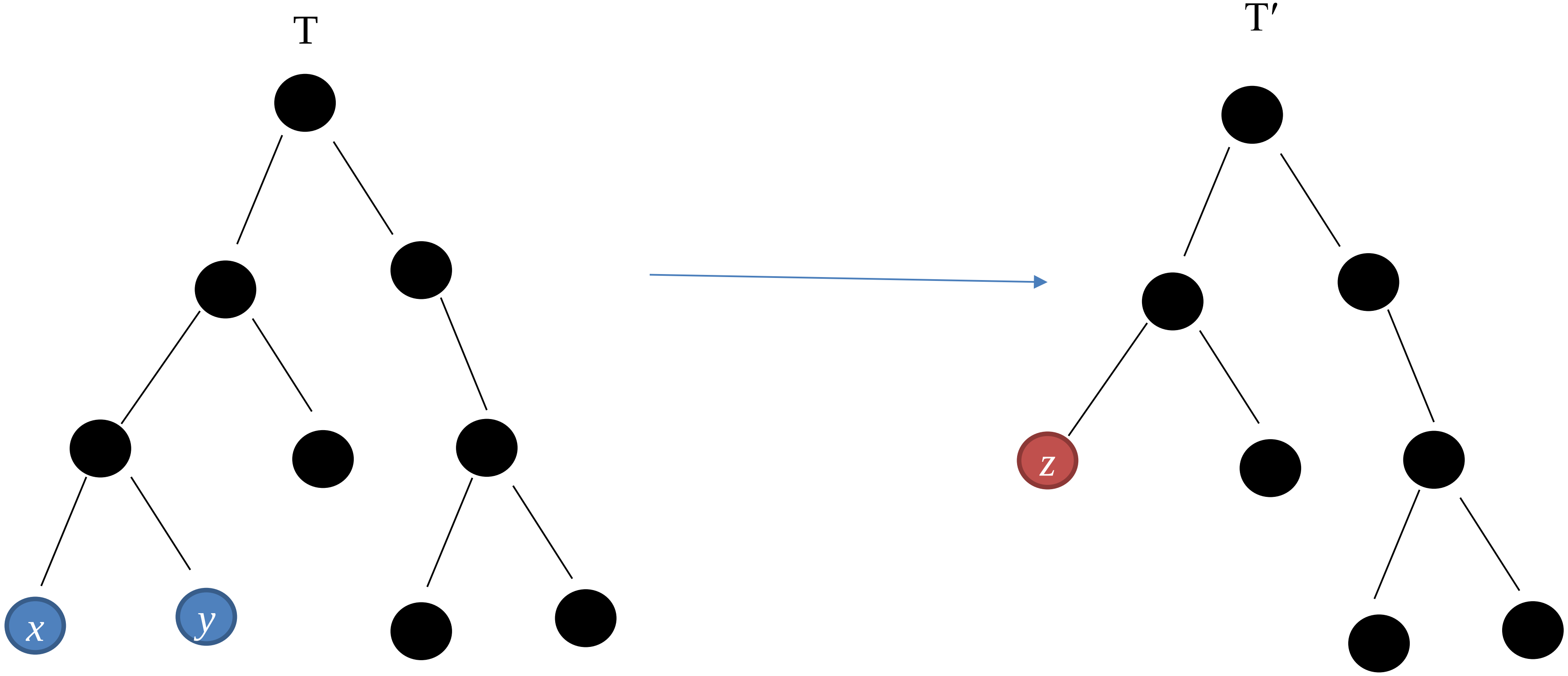
$$f_x \times h_x + f_y \times h_y + f_i \times h_i + f_j \times h_j$$

$$\boxed{f_x \times h' + f_y \times h'' + f_i \times h + f_j \times h} \geq \boxed{f_x \times h + f_y \times h + f_i \times h' + f_j \times h''}$$

Exchanging the lowest nodes (i.e., i and j) with x and y does not increase the cost.

Optimal Subproblem Property

- Do we have the same sub-problem?



$$Cost(T) - Cost(T') = f_x h + f_y h - f_z(h-1) = (f_x + f_y)h - (f_x + f_y)(h-1) = f_x + f_y$$

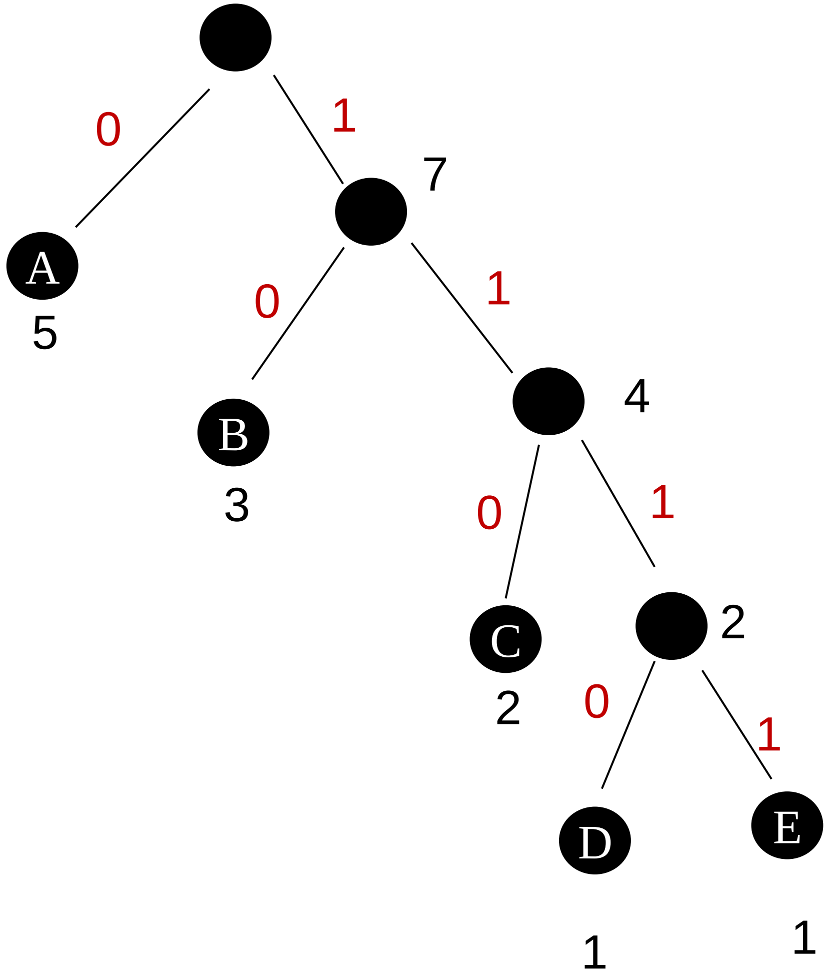
Constant

Optimal Subproblem Property (cont'd)

- *T* can be created from an optimal prefix tree *T'* by replacing a leaf node *z* with *x* and *y*.
- Suppose *T* does not represent an optimal prefix code, while *T''* does, i.e., $Cost(T'') < Cost(T)$.
  - We can assume that *x* and *y* are sibling leaf nodes in *T''*. Why?
    - Due to greedy choice property.
- Now suppose *T'''* is created from *T''* by merging *x* and *y* into *z* leaf node:
  - $Cost(T''') = Cost(T'') - f_x - f_y < Cost(T) - f_x - f_y = Cost(T')$
  - This is a contradiction as we assumed *T'* is an optimal tree. Hence, *T* represents an optimal prefix code.

Another Example

Character	Frequency	Code
A	5	0
B	3	10
C	2	110
D	1	1110
E	1	1111



Sample Problems

True or False?

- A greedy algorithm always makes the choice that looks best at the moment.
- Problems solved using dynamic programming cannot be solved thru greedy algorithms.
- If a problem can be solved using both the greedy method and dynamic programming, greedy will always give you a lower time complexity.

Fill in the Blanks

- A ______________ algorithm may produce an optimal solution when ______________ algorithm cannot, because ______________ algorithm will exclude some possible solutions while ______________ algorithm will examine all possible solutions.

  - Use either "greedy" or "dynamic programming".

Coin/Money Change Problem: Greedy Algorithm

- The objective of the *Coin Problem* is to come up with the minimum number of coins to pay X cents.
  - Consider the greedy approach to solving the coin problem for US coins (25¢, 10¢, 5¢, and 1¢):
    - We start with the largest coin (25¢ coin) and use it as many times as possible, then use as many 10¢ coins as possible on the remainder, then 5¢, then 1¢.
    - Prove or disprove that "this greedy algorithm always gives an optimal solution", i.e. gives the minimum number of coins to pay X cents.
  - Now suppose that 5¢ coins are not allowed, only 1¢, 10¢, and 25¢.
    - Prove or disprove that "the corresponding greedy approach (25¢ then 10¢ then 1¢) always gives an optimal solution".

Coin/Money Change Problem: Dynamic Programming

- فرض کنید در کشوری زندگی می‌کنید که سه نوع اسکناس دارد. اسکناس ۱ واحدی، ۳ واحدی و ۴ واحدی. از شما خواسته شده برای کمک به اقتصاد این کشور، الگوریتمی به کمک برنامه‌ریزی پویا طراحی کنید که مقادیر پول خواسته شده را با کمترین تعداد اسکناس پرداخت کند. به عنوان مثال اگر بخواهید ۶ واحد پول پرداخت کنید، اگر آن‌را با دو اسکناس ۳ واحدی پرداخت کنید به صرفه‌تر از پرداخت با یک اسکناس ۴ واحدی و دو اسکناس ۱ واحدی است (چون به جای سه اسکناس، فقط دو اسکناس استفاده کرده‌اید.) الگوریتم شما مقدار درخواستی، K را دریافت و بر اساس آن تعداد اسکناس از هر نوع را مشخص می‌کند.
- فرض کنید $b(i,k)$ کمترین تعداد اسکناسی است که با آن می‌توانید مبلغ k واحد را با i اسکناس پرداخت کنید و $u = \{1, 3, 4\}$.

$$b(i,k) = \begin{cases} 0 & \text{if } k = 0 \\ 1 + \min\limits_{0 \le j < i} b(i, k - u[j]) & \text{otherwise} \end{cases}$$

Gradient Descent

- An approach used to find the local minimum of a function.