Algorithms

Kiho Choi

Fall, 2022

Department of AI·Software
Gachon University

4. Greedy Algorithms I

Contents

- Intro. to Greedy Algorithms
- Activity selection problem
- Knapsack problem
- Huffman codes

Greedy Algorithms

- A greedy algorithm always makes the choice that looks best at the moment.
- That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

Optimization problems

- In optimization problems, there are many possible solutions.
- Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value.
- We call such a solution *an optimal solution to* the problem, as opposed to *the optimal* solution, since there may be several solutions that achieve the optimal value.

Greedy algorithms

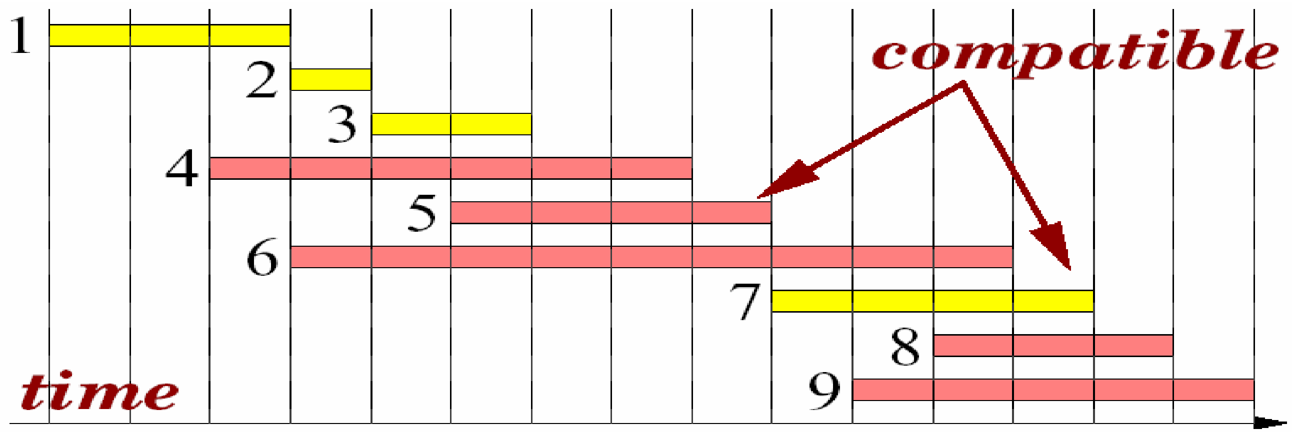
- When solving an optimization problem, we typically make a choice at each step.
- A ***greedy algorithm*** always makes the choice that looks best at the moment, without depending on any future choices.
- That is, it makes a *locally optimal choice* in the hope that this choice will lead to *a globally optimal solution*.

Activity-selection problem

- $S = \{1, 2, \ldots, n\}$

  is a set of *activities.*
- *i* takes place during time period $[s_i, f_i)$, $s_i \le f_i$.
- activities are *compatible if they have disjoint* time periods.

  Activity-selection problem : select a largest set of mutually compatible activities.



Greedy strategy

GREEDY-ACTIVITY-SELECTOR (s, f)

$n \leftarrow length[s]$

$A \leftarrow \{1\}$

$j \leftarrow 1$

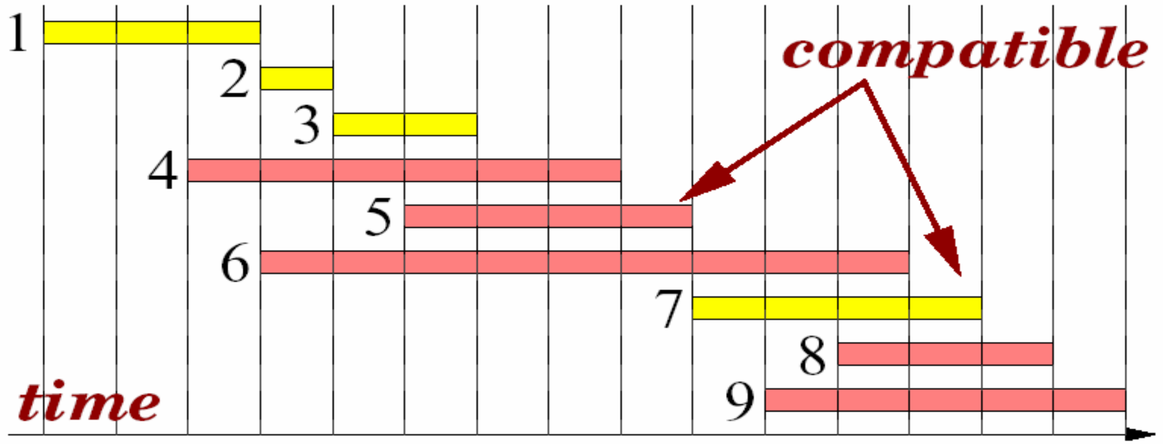
for $i \leftarrow 2$ **to** n **do**

  if $s_i \geq f_j$ **then**

    $A \leftarrow A \cup \{i\}$

    $j \leftarrow I$

  return A

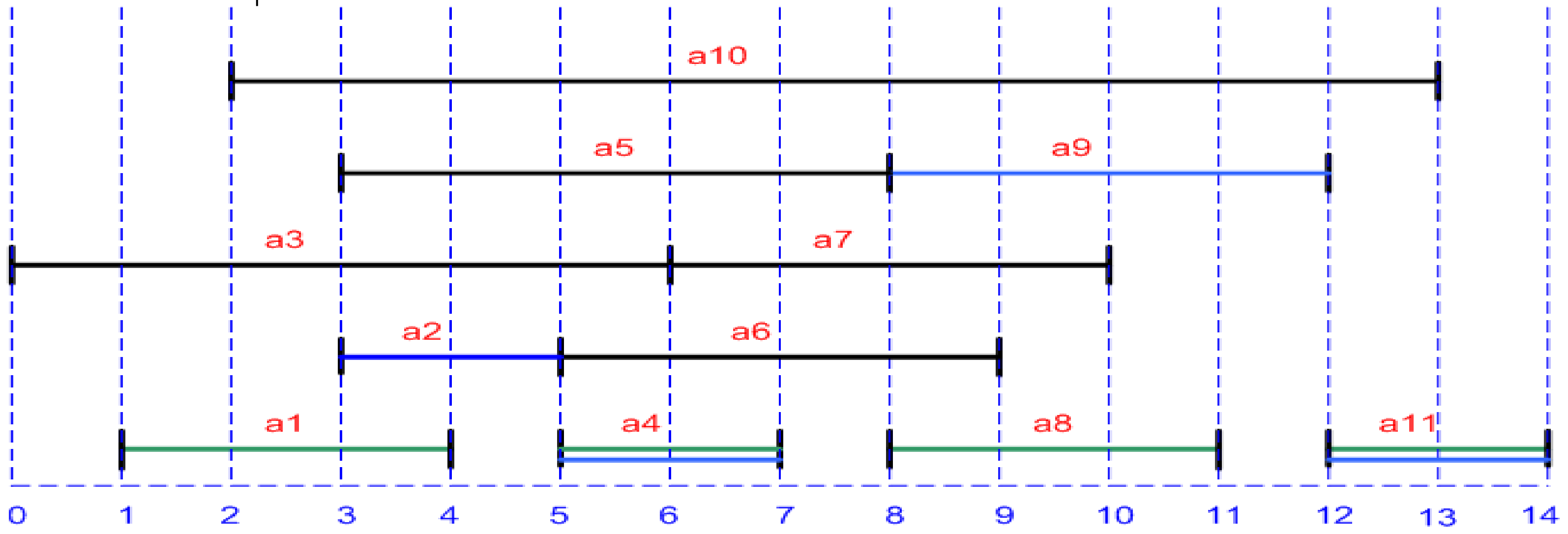


- sort the activities in non-decreasing order of finishing times.
- scan the sorted list and select current activity if it is compatible with the current selection.
- Running time: time to sort + $\Theta(n)$.

Activity selection

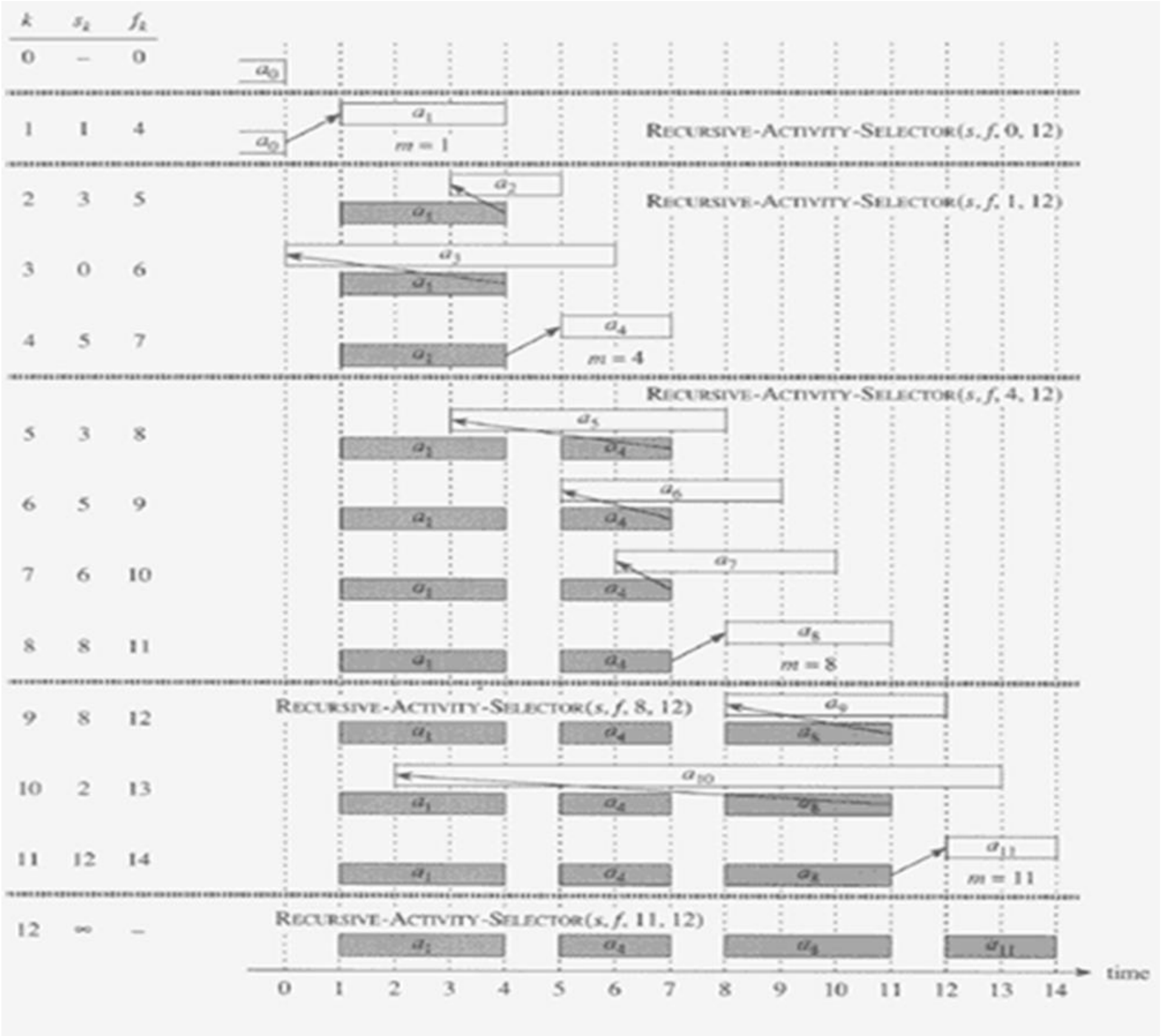
Example : *S* sorted by finish time:

a_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	8	9	10	11	12	13	14



Maximum-size mutually compatible set: $\{a1, a4, a8, a11\}$.
Not unique: also $\{a2, a4, a9, a11\}$.

Example: The operation of RECURSIVE-ACTIVITY-SELECTOR



An Iterative Greedy Algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
n ← length[s]
A ← {a_1}
i ← 1
for m ← 2 to n
        do if s_m ≥ f_i
              then A ← A ∪ {a_m}
                   i ← m
return A
```

$F_i = \max\{f_x : a_k \in A\}$

Go through example given earlier. $\Rightarrow$ $\{a_1, a_4, a_8, a_{11}\}$.

Time: *$\Theta(n)$*

Proof of correctness

Theorem. GREEDY-ACTIVITY-SELECTOR produces optimum solutions for the activity selection problem.

Proof:

- Let $A \subseteq S$ be an optimal solution whose first activity to finish is k.

If $k = 1$, then A begins with a greedy choice. Otherwise,

since $f_1 \leq f_k$, we can replace k by 1 to get an optimal solution

$B = A - \{k\} \cup \{1\}$ which starts with activity 1.

Proof of correctness

- Next, once the greedy choice of activity 1 has been made, the problem reduces to the an activity-selection problem on the set $S' = \{i \in S : s_i \geq f_1\}$ of activities 호환되는 compatible with 1, whose optimal solution A' is such that $A' \cup \{1\}$ is an optimal solution to the original problem.
- By induction on the number of choices made, we conclude that a greedy choice at each step produces an optimal solution.

Elements of the greedy strategy

- *greedy-choice property*: a global optimum can be arrived at by choosing a local optimum.
- *optimal substructure*: an optimal solution to a problem contains an optimal solution to its sub-problems.

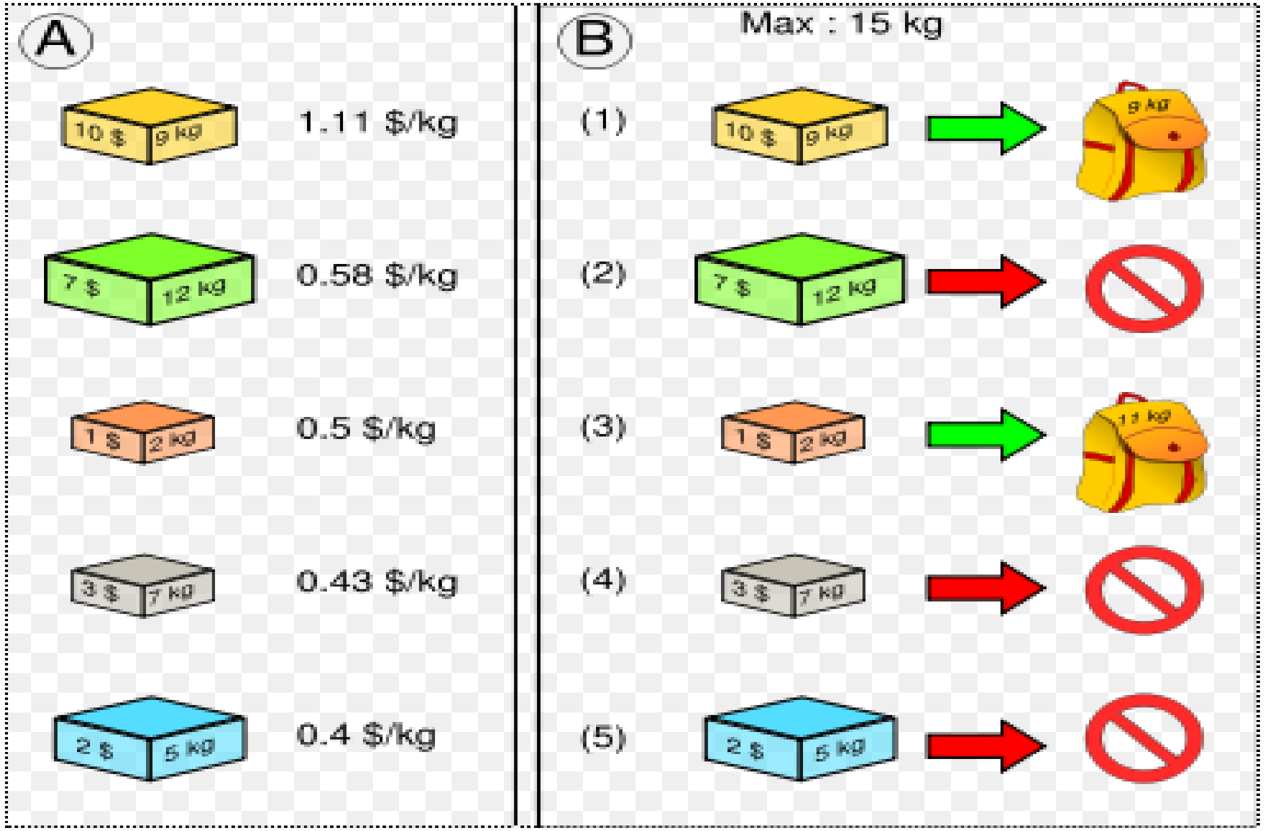
Greedy algorithms are easy to understand and implement. For some problems without the greedy-choice property, a greedy algorithm may provide a *heuristic* that works well in practice.

스스로 발견하게 하는

Greedy-choice property

1. Show that a global optimum can be modified so that a greedy choice is made as the first step.
2. Demonstrate that the greedy choice reduces the problem to a similar but smaller problem whose optimal solution can be combined with the greedy choice to obtain an optimal solution to the original problem.
3. Apply induction to show that a greedy choice can be made at each step.

Knapsack problem



Knapsack problem

A thief robbing a store finds n items: the i-th item has value v_i pesos and weighs w_i kilos. He wants to take as valuable a load as possible, but he can carry at most W kilos in his knapsack.

Assuming that v_i , w_i and W are positive integers, which items should he take?

0-1: Take an item (1) or leave it (0). Cannot take a fractional amount nor take more than one.

Fractional: Can take a fractional amount.

Each problem has an *optimal substructure.*

Knapsack problem

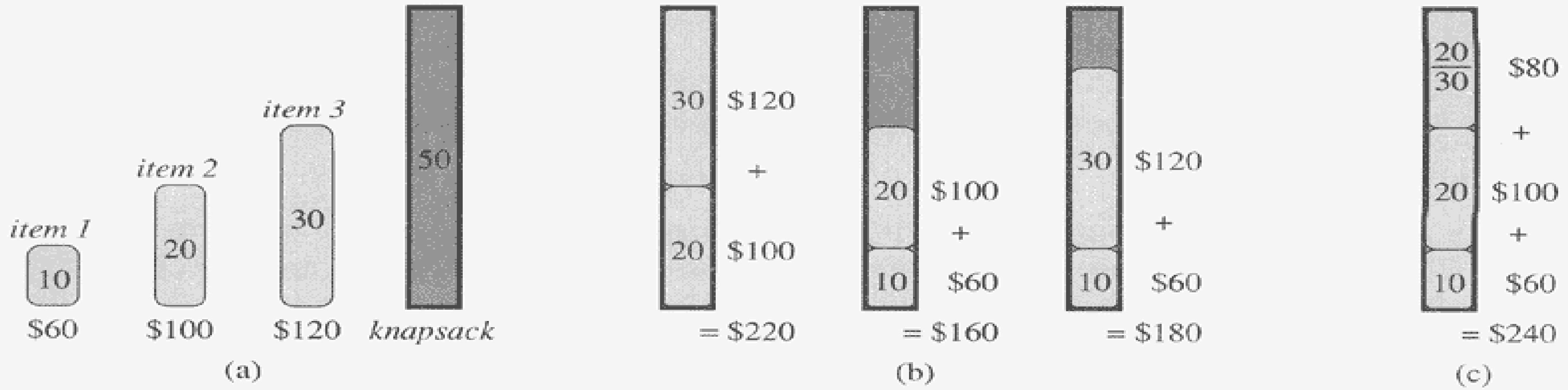


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

Greedy strategy

Sort the items in non-increasing *value density*, take whole items in that order until some item *j* does not fit, then pack a fraction of *j* to fill.

works for fractional but not for 0-1 problem:

item	1	2	3
value (pesos)	60	100	120
weight (kilos)	10	20	30
value density	6	5	4

$W = 50$

Greedy: items 1 and 2, value 160, weight 30.
Optimal: items 2 and 3, value 220, weight 50.

0-1 knapsack is harder!

- 0-1 knapsack cannot be solved by the greedy strategy
  - Unable to fill the knapsack to capacity, and the empty space lowers the effective value per pound of the packing
  - We must compare the solution to the sub-problem in which the item is included with the solution to the sub-problem in which the item is excluded before we can make the choice
  - **Dynamic Programming**

Huffman codes

Storage space for text files can be saved by *compressing* them if we are given the frequency of each character.

Assume compression is *lossless.*

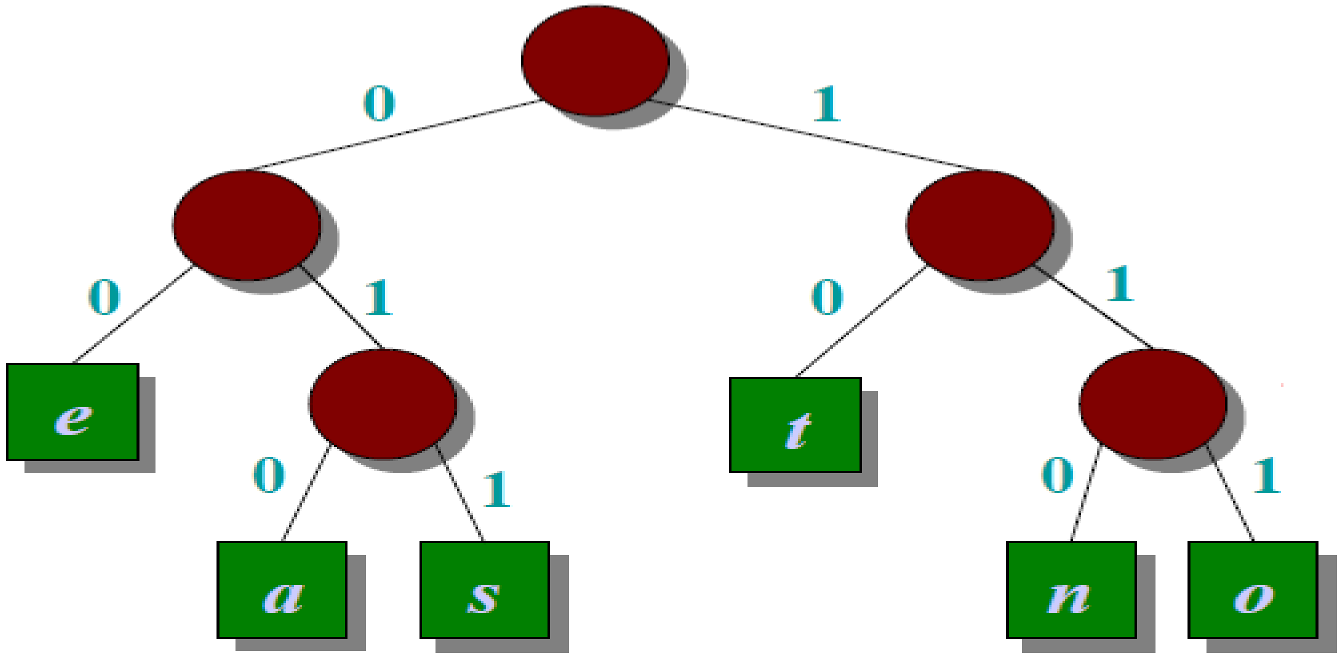
Idea: instead of using a *fixed-length binary* character code, use a *variable-length* code where

- frequent characters have shorter codes, and
- rarer characters have longer codes.

To simplify decoding, we use *prefix codes* — *no* codeword is a prefix of any other.

Prefix codes

e	00
a	010
s	011
t	10
n	110
o	111



- represent by a full binary tree.
- encode by following the root to a leaf.

Character coding problem

f:5	e:9	c:12	b:13	d:16	a:45

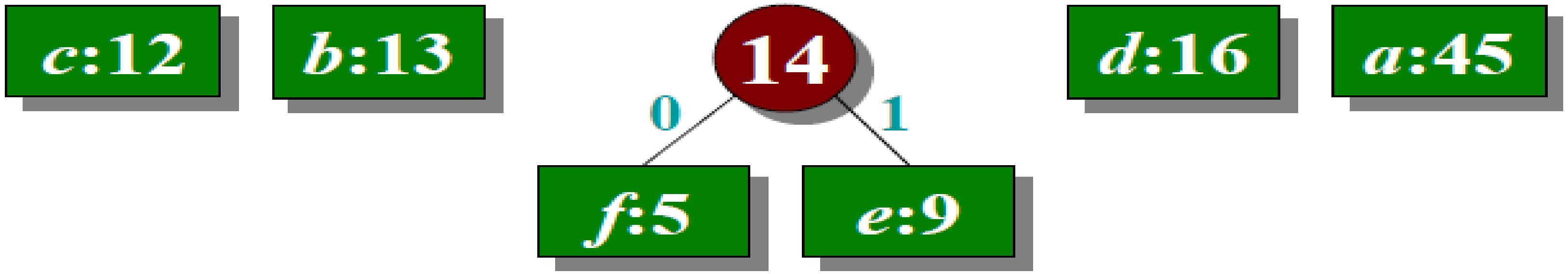
Given a text file represented by a frequency function *f* defined on an alphabet *C*, find a prefix code determined by a tree *T* which minimizes the number of bits

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

required to encode the file.

Greedy strategy: Huffman code.

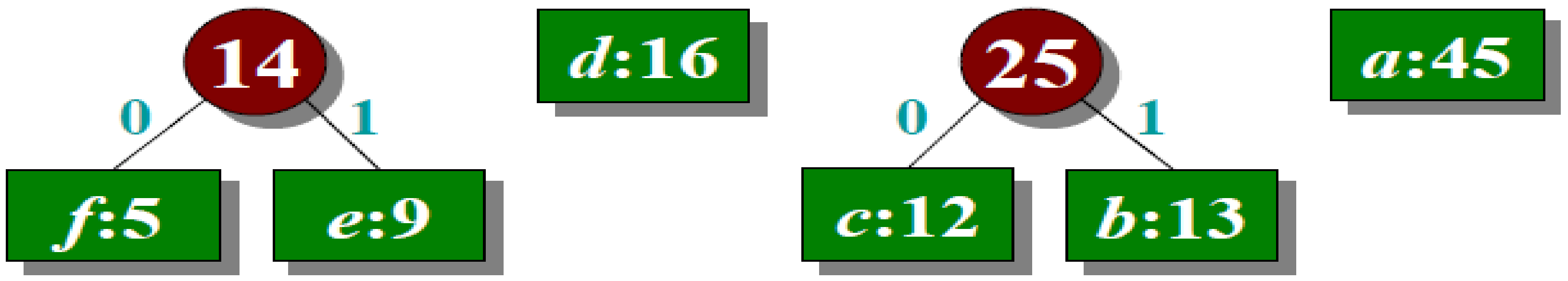
Huffman code construction



Idea: repeatedly pick two characters with the lowest frequencies and make them children of a new node whose frequency is their frequency sum.

```
HUFFMAN(C)
  n ← |C|
  Q ← C
  for i ← 1 to n − 1 do
    allocate a new node z
    left[z] ← x ← EXTRACT-MIN(Q)
    right[z] ← y ← EXTRACT-MIN(Q)
    f[z] ← f[x] + f[y]
    INSERT(Q, z)
  return EXTRACT-MIN(Q)
```

Huffman code construction

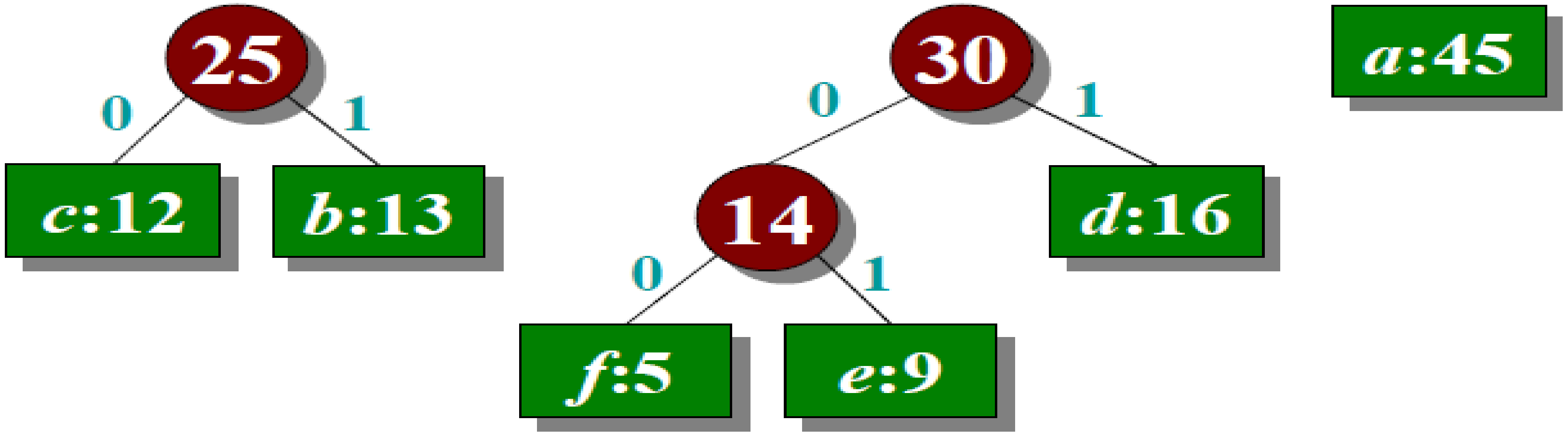


Running time: depends on min-priority queue implementation.

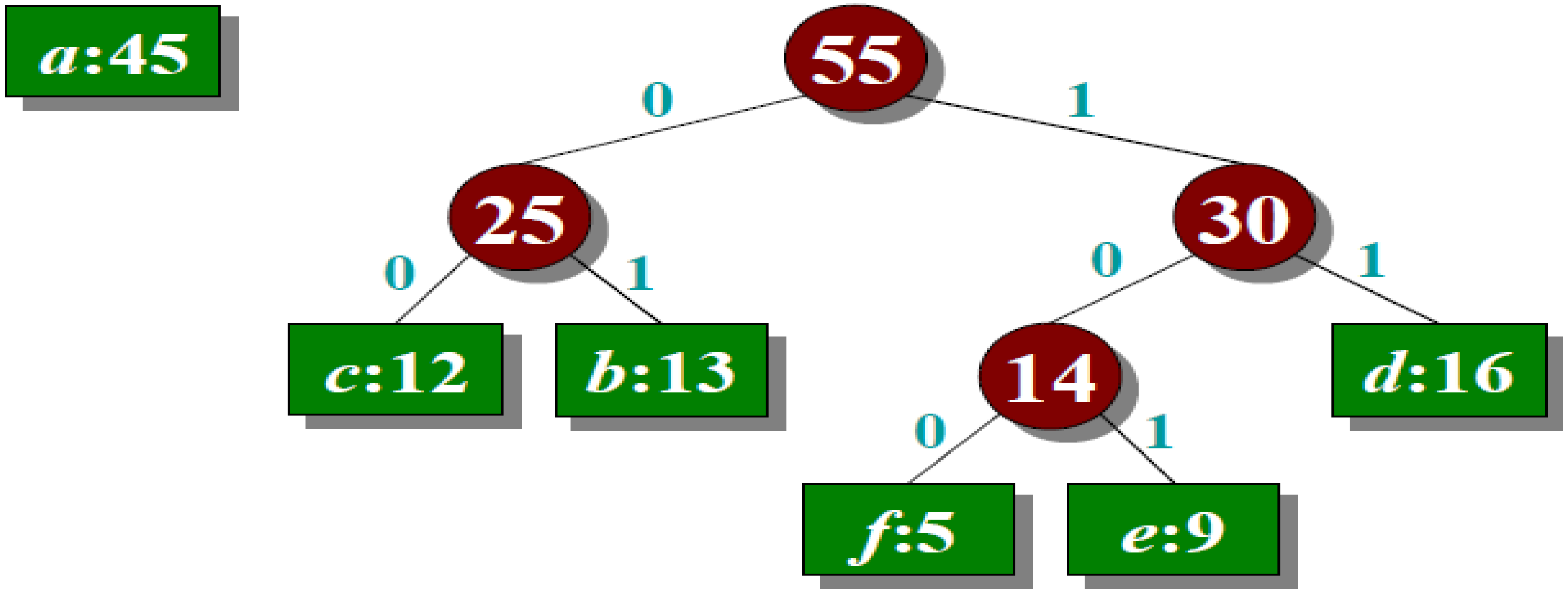
O(n lg n) with a binary min-heap.

```
HUFFMAN(C)
  n ← |C|
  Q ← C
  for i ← 1 to n − 1 do
     allocate a new node z
     left[z] ← x ← EXTRACT-MIN(Q)
     right[z] ← y ← EXTRACT-MIN(Q)
     f[z] ← f[x] + f[y]
     INSERT(Q, z)
  return EXTRACT-MIN(Q)
```

Huffman code construction

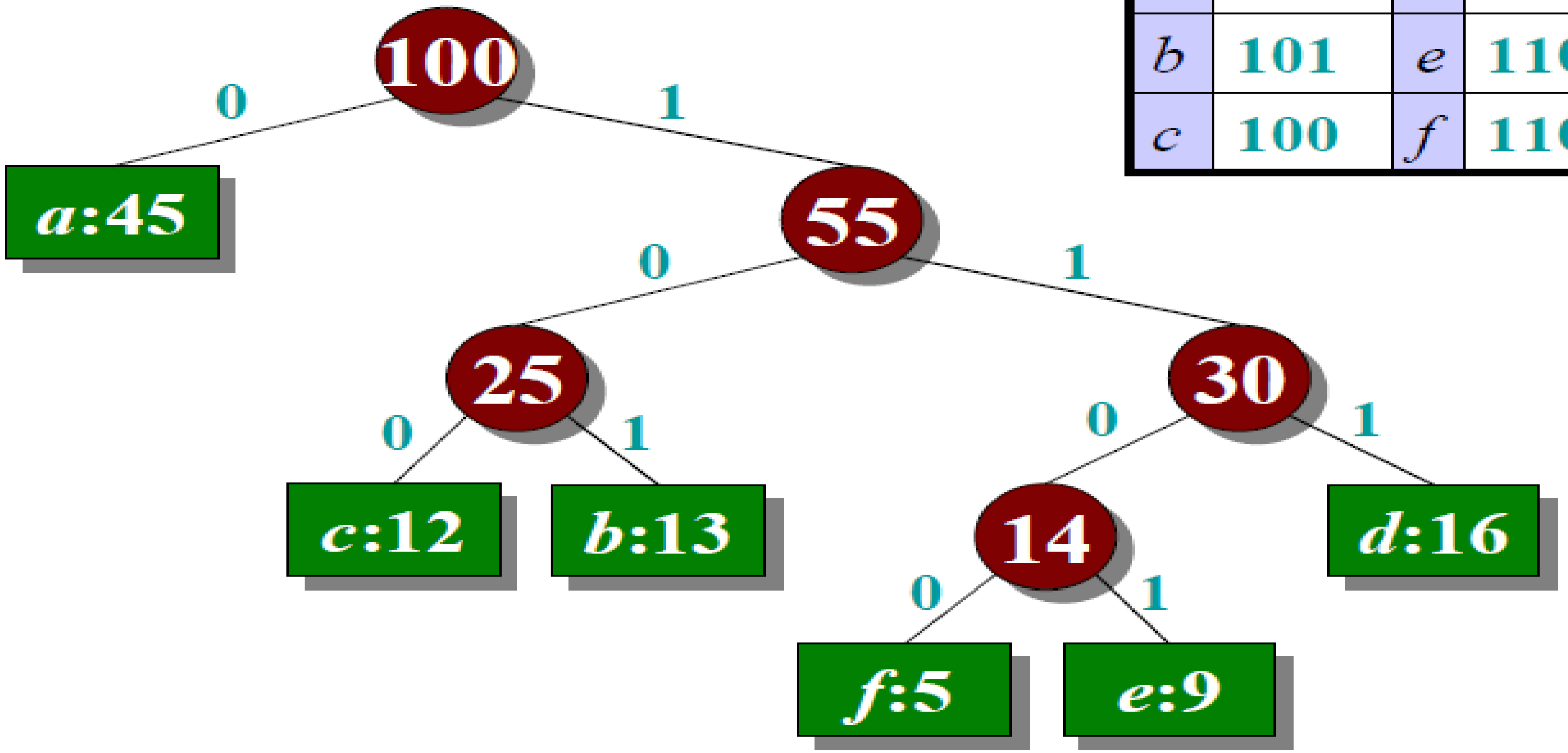


Huffman code construction



Huffman code

a	0	*d*	111
b	101	*e*	1101
c	100	*f*	1100



Optimal prefix code

Proof of correctness

Outline: Let T be an optimal tree whose deepest leaves are a and b. We can replace a and b by the greedy choices x and y to obtain another optimal tree.

Once these greedy choices have been made, the remaining problem reduces to a similar problem on $C' = C - \{x, y\} \cup \{z\}$ whose optimal tree T' produces an optimal tree for C after replacing z by an internal node with x and y as children.

THANK YOU