طراحی الگوریتم

۱۶ مهر ۹۸

ملکی مجد

Topic	Reference
Recursion and Backtracking	Ch.1 and Ch.2 JeffE
Dynamic Programming	Ch.3 JeffE and Ch.15 CLRS
Greedy Algorithms	Ch.4 JeffE and Ch.16 CLRS
Amortized Analysis	Ch.17 CLRS
Elementary Graph algorithms	Ch.6 JeffE and Ch.22 CLRS
Minimum Spanning Trees	Ch.7 JeffE and Ch.23 CLRS
Single-Source Shortest Paths	Ch.8 JeffE and Ch.24 CLRS
All-Pairs Shortest Paths	Ch.9 JeffE and Ch.25 CLRS
Maximum Flow	Ch.10 JeffE and Ch.26 CLRS
String Matching	Ch.32 CLRS
NP-Completeness	Ch.12 JeffE and Ch.34 CLRS

Greedy approach

An activity-selection problem

- select a maximum-size subset of mutually compatible activities
  - 2 4 9 11
  - 1 4 8 11

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

Dynamic programming

- Substructure

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$$

$$f_0 \leq f_1 \leq f_2 \leq \cdots \leq f_n < f_{n+1}$$

A recursive solution

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

Converting a dynamic-programming solution to a greedy solution

Theorem 16.1

Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time:

$$f_m = \min\{f_k : a_k \in S_{ij}\} .$$

Then

1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
2. The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

Elements of the greedy strategy

- A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices
- For each decision point in the algorithm, the choice that seems best at the moment is chosen

Steps from DP to greedy algorithm

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.
- Show that all but one of the subproblems induced by having made the greedy choice are empty.
- Develop a recursive algorithm that implements the greedy strategy
- Convert the recursive algorithm to an iterative algorithm.

Steps from DP to greedy algorithm

- Determine the optimal substructure of the problem.
- Develop a recursive solution.
- Prove that at any stage of the recursion, one of the optimal choices is t
- S
  the greedy choice are empty.
- Develop a recursive algorithm that implements the greedy strategy
- Convert the recursive algorithm to an iterative algorithm.

In practice, we usually streamline the above steps when designing a greedy algorithm

design greedy algorithms

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there is always an optimal solution to the original problem that makes the greedy choice, so that the greedy choice is always safe.
- Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

design greedy algorithms

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there is always an optimal solution to the original problem

the greedy-choice property and optimal substructure are the two key ingredients.

- Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Greedy-choice property

- a globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from subproblems

Optimal substructure

- A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
    - a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms

Greedy versus dynamic programming

- 0-1 knapsack problem
  - A thief robbing a store finds n items; the ith item is worth vi dollars and weighs wi pounds, where vi and wi are integers. He 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?
- 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
- example
  - (10, \$60) (20,\$100) (30,\$120) knapsack capacity: 50

Huffman codes

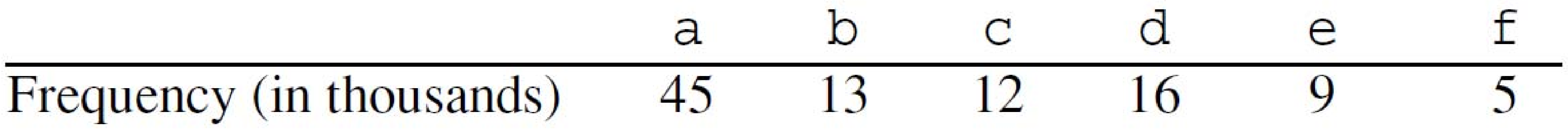
- a widely used and very effective technique for compressing data
    - savings of 20% to 90% are typical, depending on the characteristics of the data being compressed

- consider the data to be a sequence of characters.
- Huffman’s greedy algorithm uses a table of the frequencies of occurrence of the characters to build up an optimal way of representing each character as a binary string.

Example

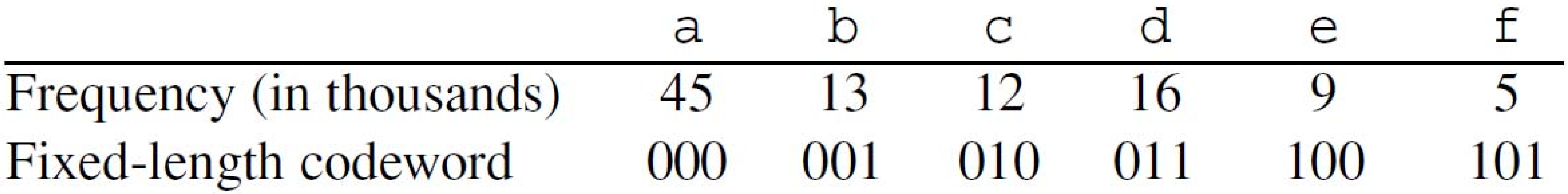
- a 100,000-character data file that we wish to store compactly
  - many ways to represent such a file of information

- consider the problem of designing a ***binary character code*** (or ***code*** for short) wherein each character is represented by a unique binary string.



	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5

- If we use a ***fixed-length code***,
- we need 3 bits to represent six characters: a = 000, b = 001, . . . , f = 101. This
- method requires 300,000 bits to code the entire file. Can we do better?



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

- A ***variable-length code*** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.
  - a savings of approximately 25%.

	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

Prefix codes

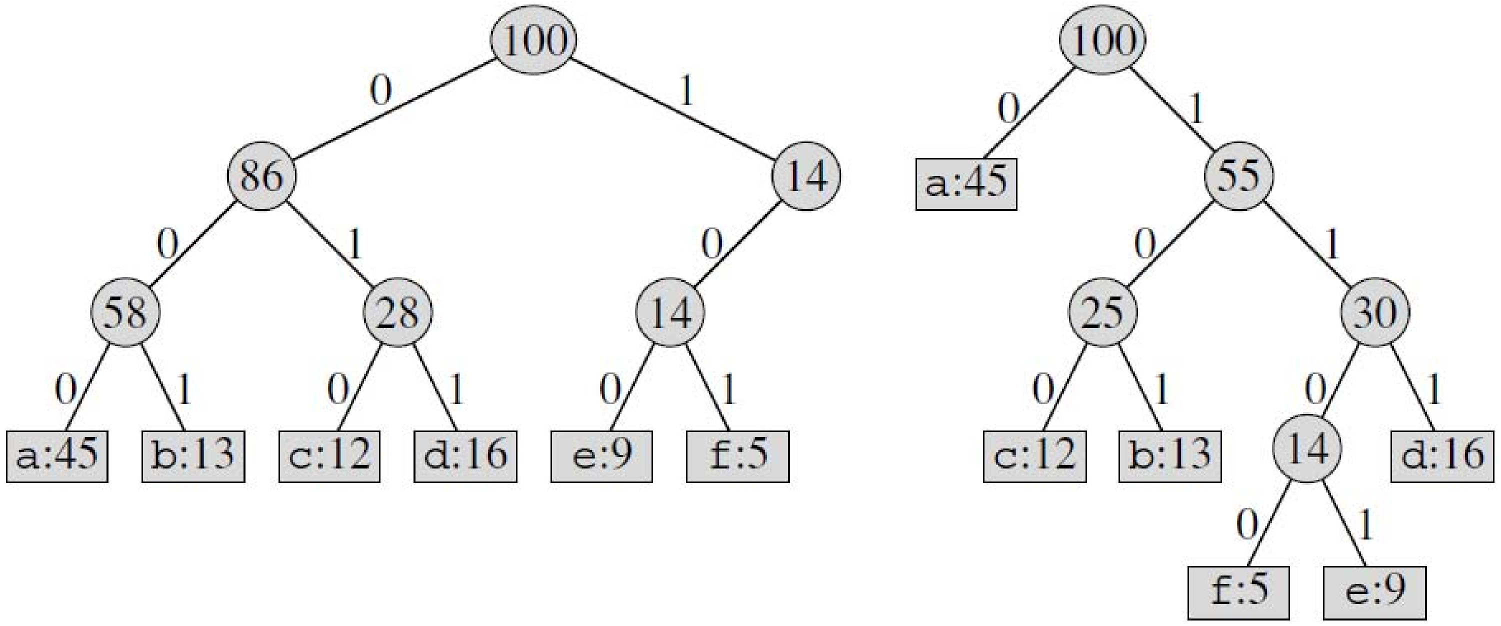
- consider only codes in which no codeword is also a prefix of some other codeword.
  - the optimal data compression achievable by a character code can always be achieved with a prefix code
- Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file

- code the 3-character file abc as 0·101·100 = 0101100
- string 001011101 parses uniquely as 0 · 0 · 101 · 1101, which decodes to aabe

decoding process

- A binary tree whose leaves are the given characters provides a convenient representation

- We interpret the binary codeword for a character as the path from the root to that character, where 0 means “go to the left child” and 1 means “go to the right child.”
    - these are not binary search trees
    - optimal code for a file is always represented by a *full* binary tree
    - an optimal prefix code has exactly $|C|$ leaves,


100
0
1
86
14
0
1
0
58
28
14
0
1
0
1
0
1
a:45
b:13
c:12
d:16
e:9
f:5
100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

cost

- the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and exactly $|C|-1$ internal nodes

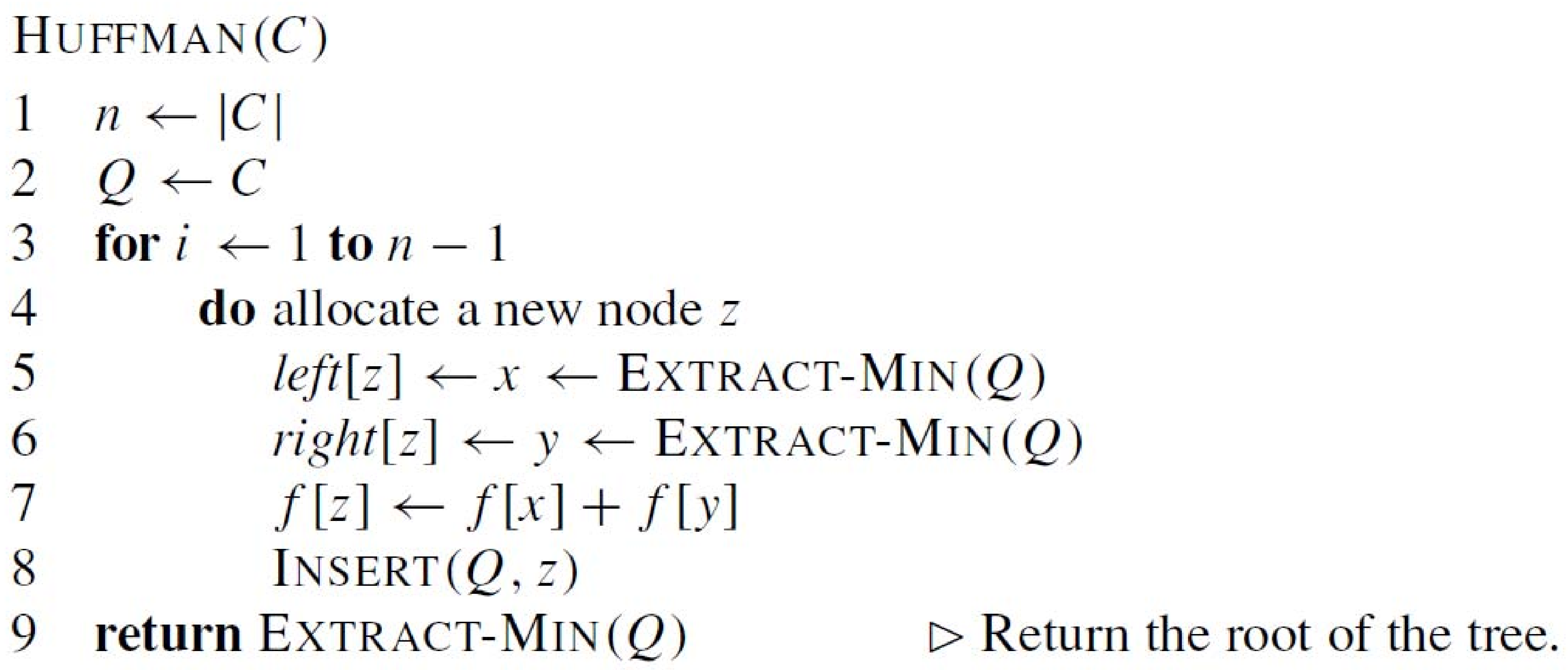
- the ***cost*** of the tree T (number of bits required to encode a file)
  - the length of the codeword for character *c*
  - frequency of *c* in the file

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

Constructing a Huffman code

- proof of correctness relies on the greedy-choice property and optimal substructure
- The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner
- It begins with a set of $|C|$ leaves and performs a sequence of $|C| - 1$ "merging" operations to create the final tree
  - min-priority queue Q, keyed on f , is used to identify the two least-frequent objects to merge together
  - The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

pseudocode

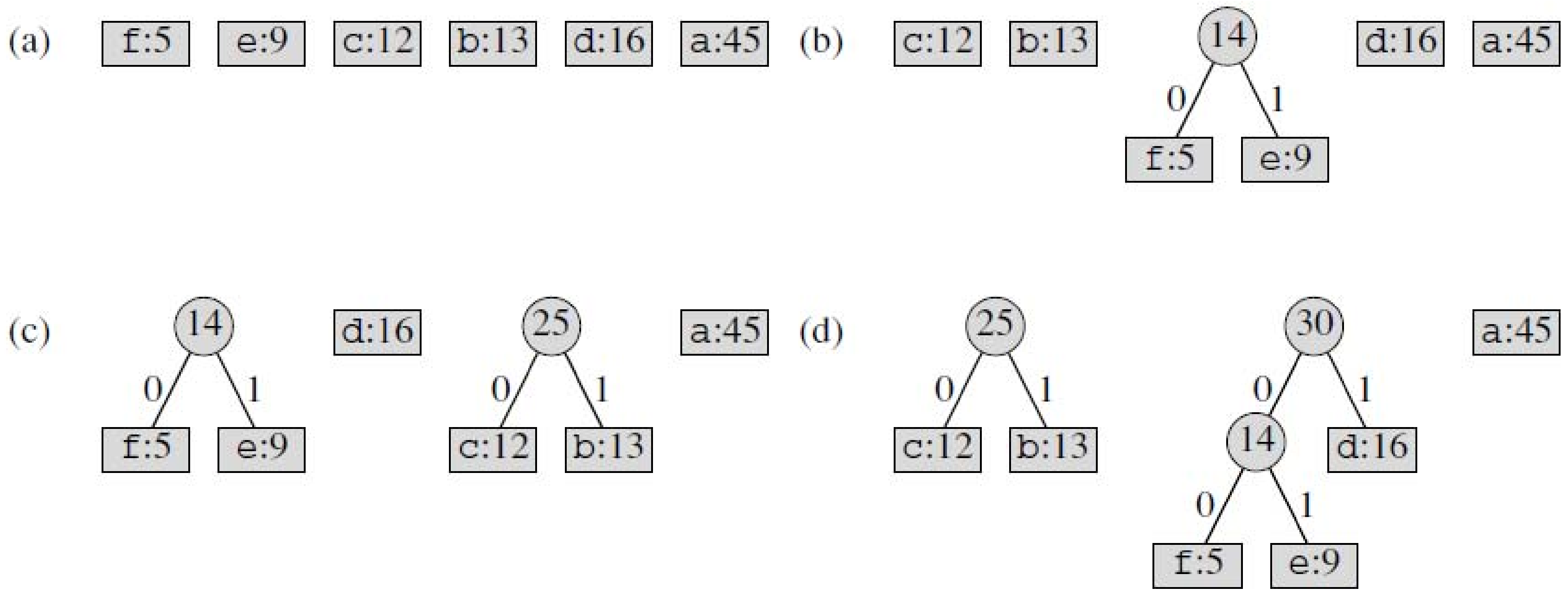


```
HUFFMAN($C$)
$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ to $n - 1$
    do allocate a new node $z$
        $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
        $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
        $f[z] \leftarrow f[x] + f[y]$
        INSERT($Q, z$)
return EXTRACT-MIN($Q$)                ▷ Return the root of the tree.
```

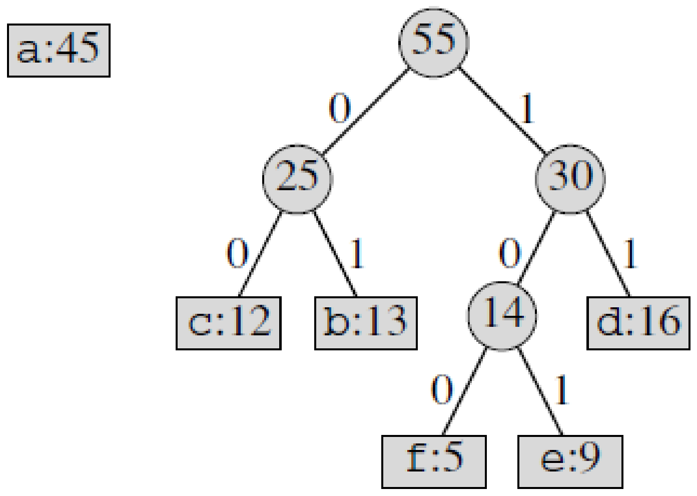
running time

- $O(n \lg n)$

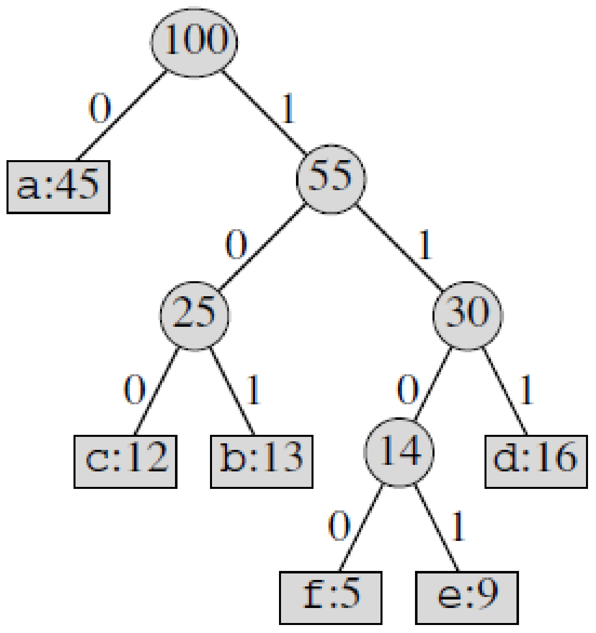
Construct tree



(e)


a:45
55
0
1
25
30
c:12
b:13
14
d:16
f:5
e:9

(f)


100
0
1
a:45
55
25
30
c:12
b:13
14
d:16
f:5
e:9