بسم اللّه الرّحمن الرّحیم

دانشگاه صنعتی اصفهان ـ دانشکدهٔ مهندسی برق و کامپیوتر
(نیم‌سال تحصیلی ۴۰۲۲)

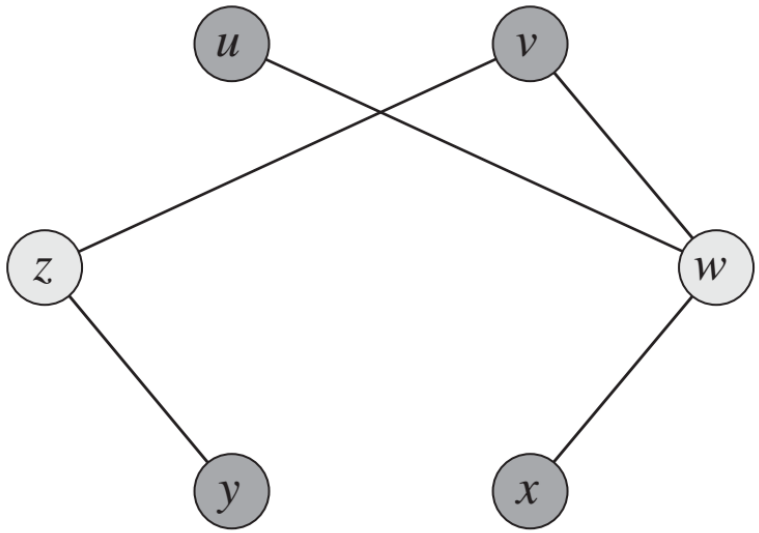
طراحی الگوریتم‌ها

حسین فلسفین

The vertex-cover problem

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). That is, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. The size of a vertex cover is the number of vertices in it.

For example, the following graph has a vertex cover $\{w, z\}$ of size 2.



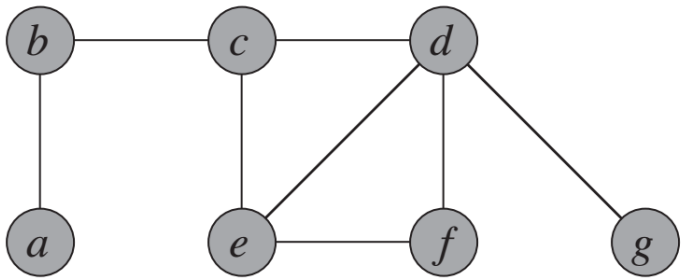
The vertex-cover problem is to find a vertex cover of minimum size in a given graph. We call such a vertex cover an optimal vertex cover.

The vertex-cover problem is NP-hard.

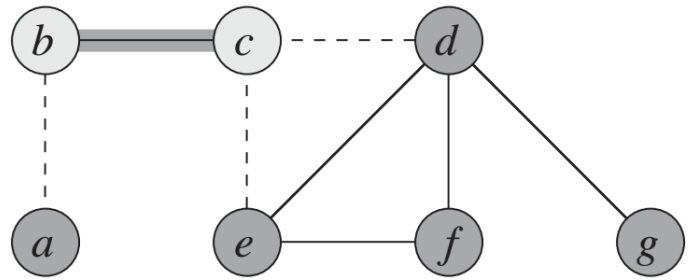
We can efficiently find a vertex cover that is near-optimal. The following approximation algorithm takes as input an undirected graph G and returns a vertex cover whose size is guaranteed to be no more than twice the size of an optimal vertex cover.

APPROX-VERTEX-COVER(G)

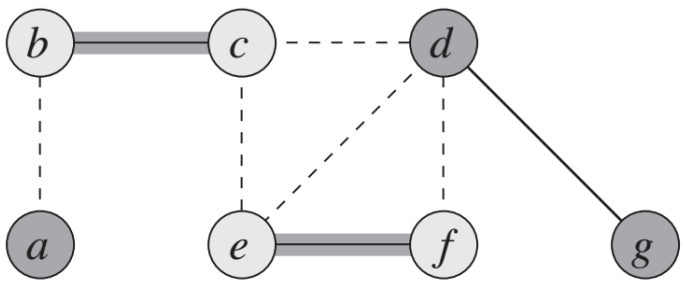
```
1  C = ∅
2  E' = G.E
3  while E' ≠ ∅
4      let (u, v) be an arbitrary edge of E'
5      C = C ∪ {u, v}
6      remove from E' every edge incident on either u or v
7  return C
```



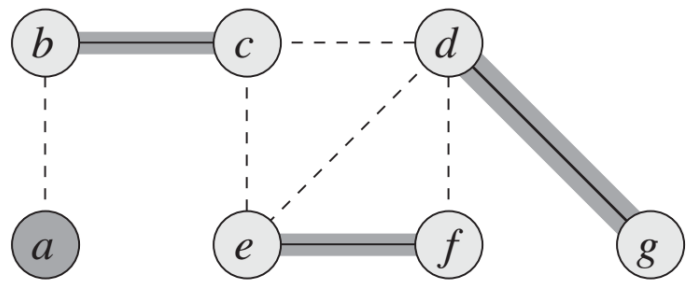
(a)



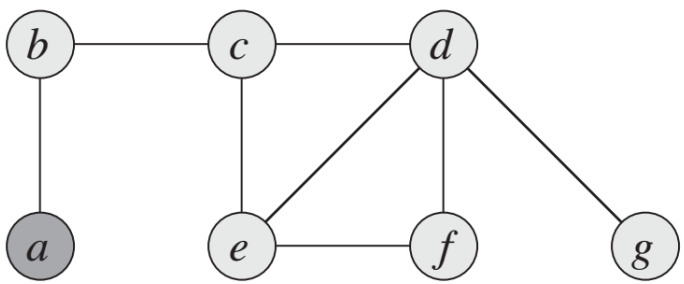
(b)



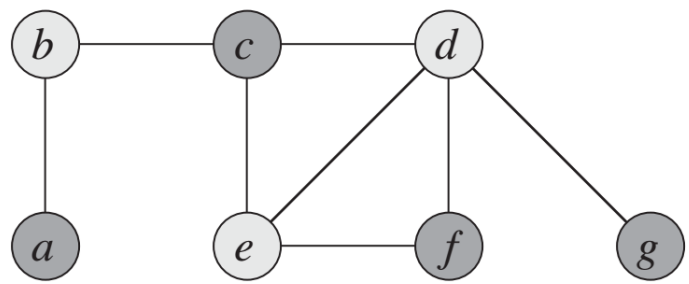
(c)



(d)



(e)



(f)

توضیحات مربوط به شکل بالا:

The operation of APPROX-VERTEX-COVER. (a) The input graph G, which has 7 vertices and 8 edges. (b) The edge (b, c), shown heavy, is the first edge chosen by APPROX-VERTEX-COVER. Vertices b and c, shown lightly shaded, are added to the set C containing the vertex cover being created. Edges (a, b), (b, c), and (c, d), shown dashed, are removed since they are now covered by some vertex in C. (c) Edge (e, f) is chosen; vertices e and f are added to C. (d) Edge (d, g) is chosen; vertices d and g are added to C. (e) The set C, which is the vertex cover produced by APPROX-VERTEX-COVER, contains the six vertices b, c, d, e, f, g. (f) The optimal vertex cover for this problem contains only three vertices: b, d, e.

قضیهٔ زیر و اثباتش مهم هستند:

Theorem: *APPROX-VERTEX-COVER is a polynomial-time 2-approximation algorithm.*

Proof: ☞ *It can easily be seen that APPROX-VERTEX-COVER runs in polynomial time.*

☞ *The set C of vertices that is returned by APPROX-VERTEX-COVER is a vertex cover, since the algorithm loops until every edge in $G.E$ has been covered by some vertex in C.*

☞ *To see that APPROX-VERTEX-COVER returns a vertex cover that is at most twice the size of an optimal cover, let A denote the set of edges that line 4 of APPROX-VERTEX-COVER picked. In order to cover the edges in A, any vertex cover —in particular, an optimal cover C^*— must include at least one endpoint of each edge in A. No two edges in A share an endpoint, since once an edge is picked in line 4, all other edges that are incident on its endpoints are deleted from E' in line 6. Thus, no two edges in A are covered by the same*

vertex from C^*, and we have the lower bound

$$|C^*| \geq |A|$$

on the size of an optimal vertex cover. Each execution of line 4 picks an edge for which neither of its endpoints is already in C, yielding an upper bound (an exact upper bound, in fact) on the size of the vertex cover returned:

$$|C| = 2|A|$$

Combining the above two equations, we obtain

$$|C| = 2|A| \leq 2|C^*|$$

thereby proving the theorem. □

یک نکته دربارهٔ استفاده از دو لفظ *NP-complete* و *NP-hard*

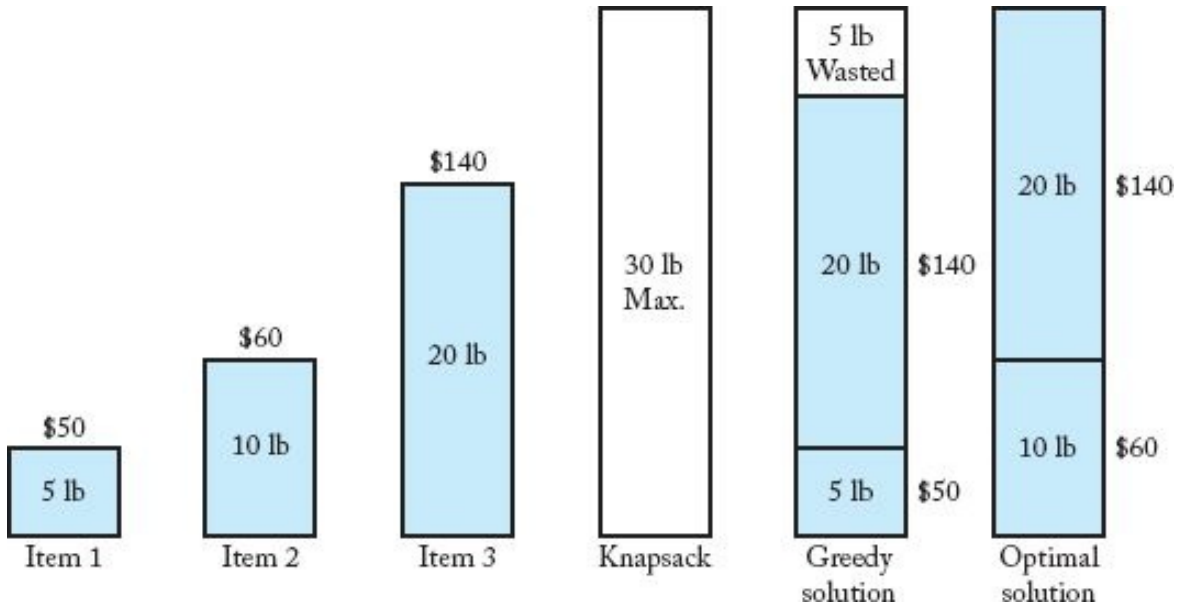
We use the term NP-hard to mean "at least as hard as an NP-complete problem." We avoid referring to optimization problems as NP-complete, since technically this term applies only to decision problems.

A Greedy Approach to the 0-1 Knapsack Problem

An obvious greedy strategy is to steal the items with the largest profit first; that is, steal them in nonincreasing order according to profit. This strategy, however, would not work very well if the most profitable item had a large weight in comparison to its profit. For example, suppose we had three items, the first weighing 25 pounds and having a profit of $10, and the second and third each weighing 10 pounds and having a profit of $9. If the capacity W of the knapsack was 30 pounds, this greedy strategy would yield only a profit of $10, whereas the optimal solution is $18.

Another obvious greedy strategy is to steal the lightest items first. This strategy fails badly when the light items have small profits compared with their weights.

To avoid the pitfalls of the previous two greedy algorithms, a more sophisticated greedy strategy is to steal the items with the largest profit per unit weight first. That is, we order the items in nonincreasing order according to profit per unit weight, and select them in sequence. An item is put in the knapsack if its weight does not bring the total weight above W.



This approach is illustrated in the above figure. We have the following profits per unit weight:

$$item_1 : \frac{\$50}{5} = \$10, \quad item_2 : \frac{\$60}{10} = \$6, \quad item_3 : \frac{\$140}{20} = \$7.$$

Ordering the items by profit per unit weight yields

$$item_1, item_3, item_2.$$

As can be seen in the figure, this greedy approach chooses $item_1$ and $item_3$, resulting in a total profit of \$190, whereas the optimal solution is to choose $item_2$ and $item_3$, resulting in a total profit of \$200. The problem is that after $item_1$ and $item_3$ are chosen, there are 5 pounds of capacity left, but it is wasted because item 2 weighs 10 pounds. Even this more sophisticated greedy algorithm does not solve the 0-1 Knapsack problem.

The following example shows that $f(s_a)$ can be arbitrarily far from the optimal value.

Let us consider the following instance of 0-1 Knapsack problem defined over n items: $v_i = w_i = 1$ for $i = 1, \ldots, n-1$, $v_n = W - 1$, and $w_n = W = kn$ where k is an arbitrarily large number. In this case, $f(s^*) = W - 1 = kn - 1$ while the greedy algorithm finds a solution whose value is $f(s_a) = n - 1$: hence, $f(s^*)/f(s_a) > k$. An analogous result can be easily shown if the items are sorted in nondecreasing order with respect to their profit or in non-increasing order with respect to their occupancy.

یک نگاه دقیق‌تر به مثال فوق نشان می‌دهد که رفتار ضعیف روش حریصانهٔ مبتنی بر v_i/w_i به‌سبب آن است که الگوریتم، آیتم با بیشترین قیمت را برنمی‌دارد درحالی‌که جواب بهینه تنها شامل همین آیتم است. این نقطه‌ضعف، تنها با اعمال یک تغییر ساده در روند حریصانه برطرف خواهد شد:

Theorem: *Given an instance x of the 0-1 Knapsack problem, let $f(s'_a) = \max(v_{\max}, f(s_a))$, where $v_{\max}$ is the maximum profit of an item in x. Then $f(s'_a)$ satisfies the following inequality: $f(s^*)/f(s'_a) < 2$.*

Proof: *Let j be the index of the first item not inserted in the knapsack by the greedy algorithm with input x (because of its high weight). The profit achieved by the algorithm at that point is*

$$\overline{v}_j = \sum_{i=1}^{j-1} v_i \leq f(s_a)$$

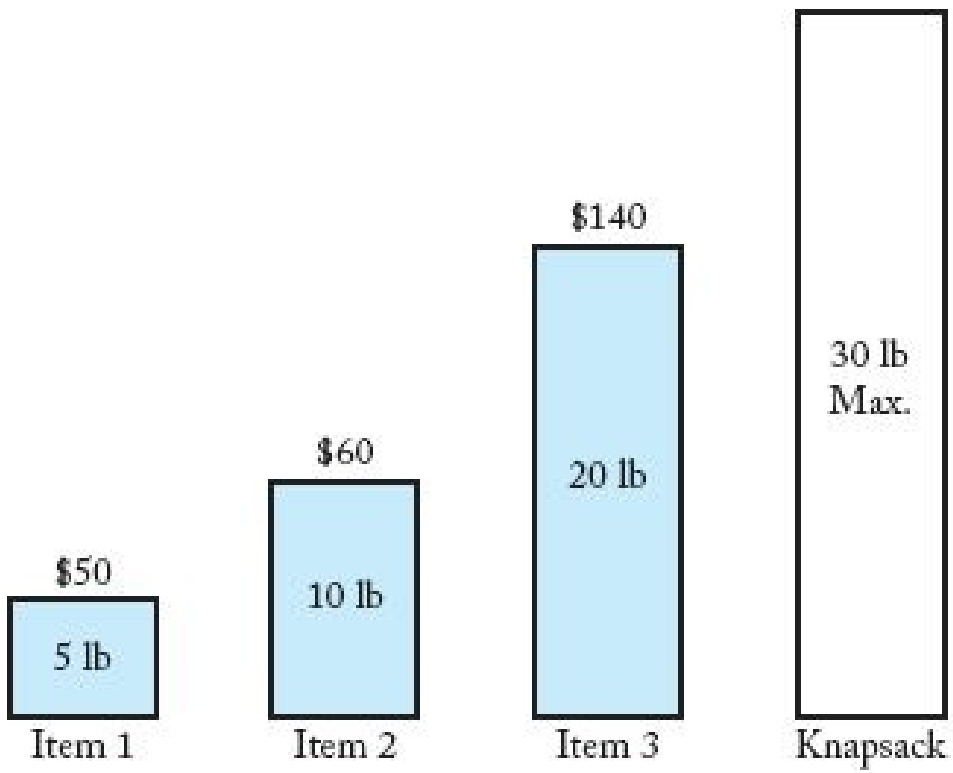
and the overall weight is

$$\overline{w}_j = \sum_{i=1}^{j-1} w_i \leq W.$$

We first show that any optimal solution of the given instance must satisfy the following inequality: $f(s^*) < \overline{v}_j + v_j$. *The optimal value is bounded by* $\overline{v}_j$ *plus the maximum profit obtainable by filling the remaining available space of the knapsack (that is,* $W - \overline{w}_j$*) with items whose profit/weight ratio is at most* v_j/w_j *(notice that the items are ordered in non-increasing order with respect to the ratio* v_j/w_j*). Since* $\overline{w}_j + w_j > W$*, we obtain* $f(s^*) \leq \overline{v}_j + (W - \overline{w}_j)v_j/w_j < \overline{v}_j + v_j$. *To complete the proof we consider two possible cases. If* $v_j \leq \overline{v}_j$ *then* $f(s^*) < 2\overline{v}_j \leq 2f(s_a) \leq 2f(s'_a)$. *On the other hand, if* $v_j > \overline{v}_j$ *then* $v_{\max} > \overline{v}_j$*; in this case we have that* $f(s^*) < \overline{v}_j + v_j \leq \overline{v}_j + v_{\max} < 2v_{\max} \leq 2f(s'_a)$. *Thus, in both cases the theorem follows.* □

A Greedy Approach to the Fractional Knapsack Problem

In the Fractional Knapsack problem, the thief does not have to steal all of an item, but rather can take any fraction of the item. We can think of the items in the 0-1 Knapsack problem as being gold or silver ingots and the items in the Fractional Knapsack problem as being bags of gold or silver dust.

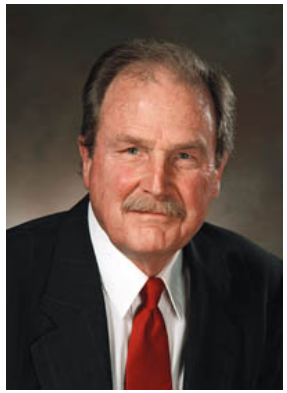


If our greedy strategy is again to choose the items with the largest profit per unit weight first, all of $item_1$ and $item_3$ will be taken as before. However, we can use the 5 pounds of remaining capacity to take $5/10$ of $item_2$. Our total profit is

$$\$50 + \$140 + \frac{5}{10}(\$60) = \$220.$$

Our greedy algorithm never wastes any capacity in the Fractional Knapsack problem as it does in the 0-1 Knapsack problem. As a result, it always yields an optimal solution.

Huffman codes



David A. Huffman (1925–1999)

Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed. We consider the data to be a sequence of characters. Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.

	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

A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated.

☞ ***If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits (fixed-length code).***

☞ ***Using the variable-length code shown, we can encode the file in only 224,000 bits (In fact, this is an optimal character code for this file.)***

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000 \textbf{ bits}$$

A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short code-words and infrequent characters long codewords.

☞ *We consider here only codes in which no codeword is also a prefix of some other codeword. Such codes are called prefix codes.*

☞ *Perhaps "prefix-free codes" would be a better name, but the term "prefix codes" is standard in the literature.*

☞ *Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file. For example, with the variable-length prefix code of the above figure, we code the 3-character file abc as*

$$0 \cdot 101 \cdot 100 = 0101100,$$

where "$\cdot$" denotes concatenation.

☞ *Prefix codes are desirable because they simplify decoding. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.*

☞ *We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file. In our example, the string* 001011101 *parses uniquely as* $0 \cdot 0 \cdot 101 \cdot 1101$*, which decodes to* $aabe$*.*

چرا کدگذاری زیر نامطلوب است؟ جواب: یکتا بودن برقرار نیست.

$$a \to 01 \quad b \to 0 \quad c \to 00.$$

چرا کدگذاری زیر نامطلوب است؟ جواب: یکتا بودن برقرار است اما عملیات کدگشایی کارآمد نیست.

$$a \to 0 \quad b \to 01 \quad c \to 11.$$

فرض کنید که دنبالهٔ دریافتی بشکل زیر باشد:

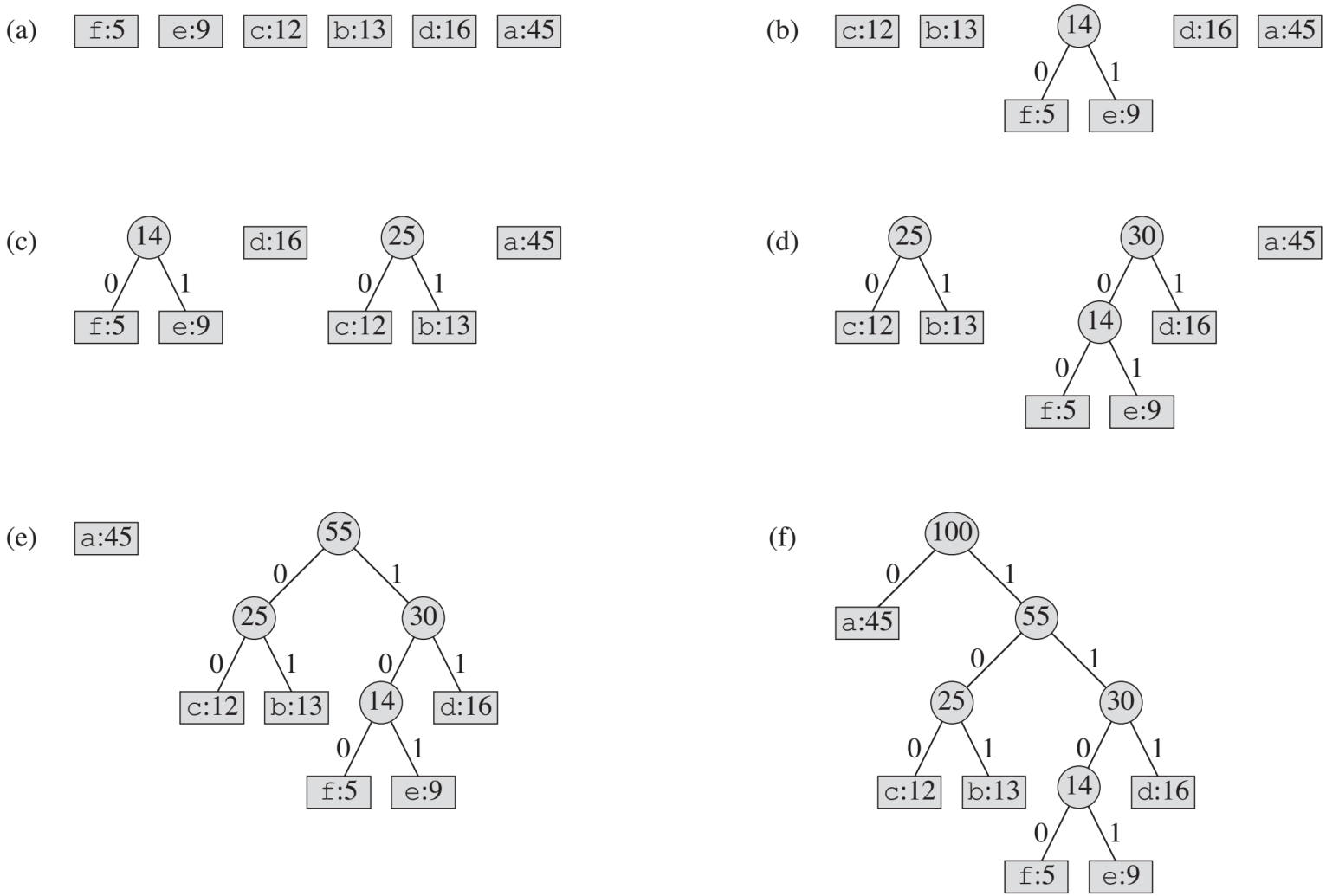
$$0\underbrace{11111111111111111111\cdots}$$

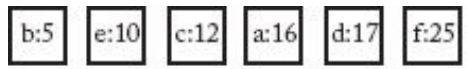
طول این دنباله زوج است یا فرد؟!

وقتی از کدهای پیشوندی استفاده میکنیم خیالمان از هر دو نظر راحت است.

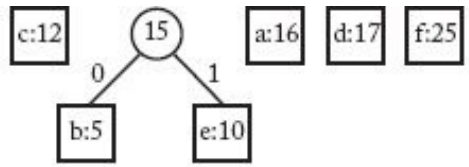
The advantage of a prefix code is that we need not look ahead when parsing the file.

Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code.

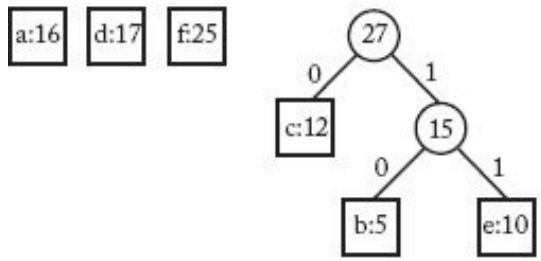



b:5
e:10
c:12
a:16
d:17
f:25

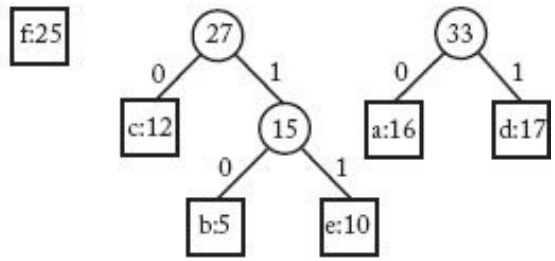
(0)


c:12
15
0
1
a:16
d:17
f:25
b:5
e:10

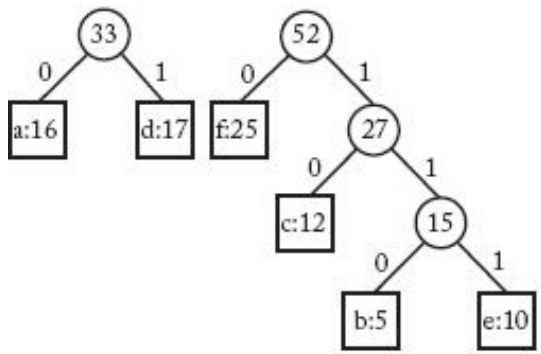
(1)


a:16
d:17
f:25
27
0
1
c:12
15
0
1
b:5
e:10

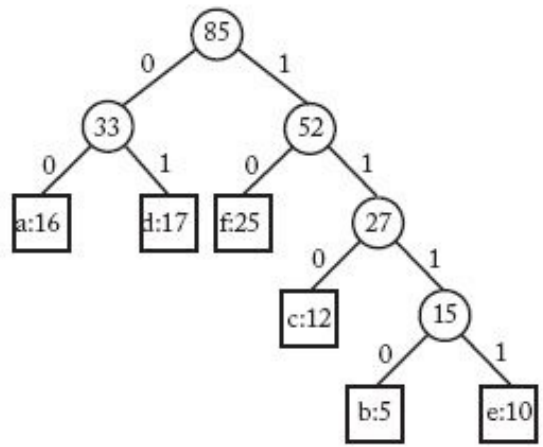
(2)


f:25
27
0
1
c:12
15
0
1
b:5
e:10
33
0
1
a:16
d:17

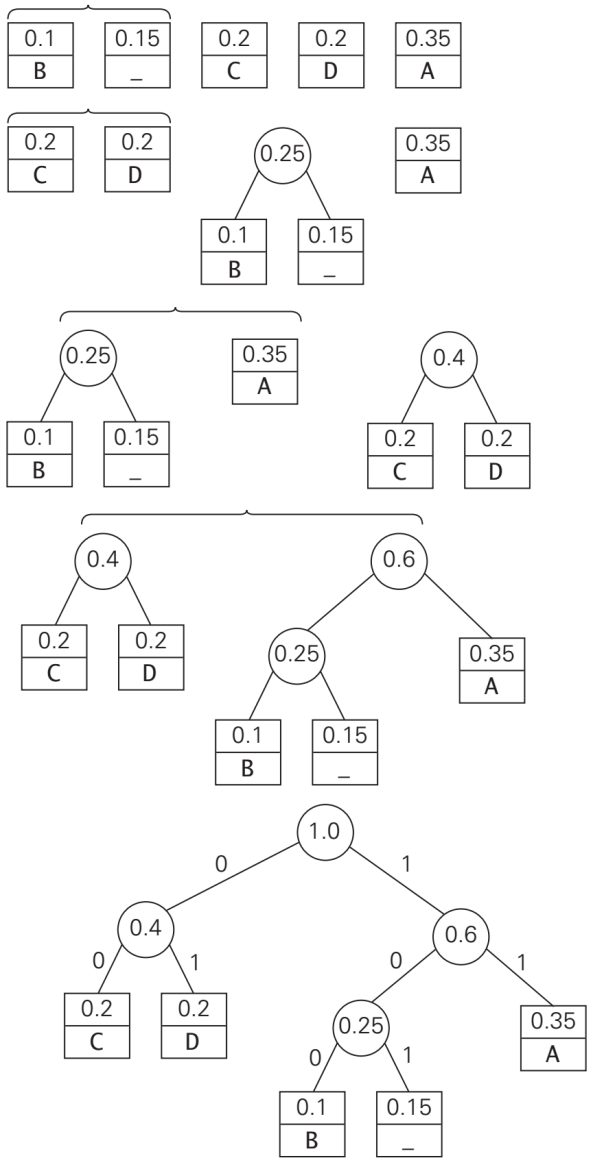
(3)


33
0
1
a:16
d:17
52
0
1
f:25
27
0
1
c:12
15
0
1
b:5
e:10

(4)


85
0
1
33
52
0
1
0
1
a:16
d:17
f:25
27
0
1
c:12
15
0
1
b:5
e:10

(5)


0.1
B
0.15
_
0.2
C
0.2
D
0.35
A
0.25
0.4
0.6
0
1

In the pseudocode that follows, we assume that C is a set of n characters and that each character $c \in C$ is an object with an attribute $c.freq$ giving its frequency. 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. The algorithm uses a min-priority queue Q, keyed on the $freq$ attribute, to identify the two least-frequent objects to merge together. When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.

```
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 = EXTRACT-MIN(Q)
6       z.right = y = EXTRACT-MIN(Q)
7       z.freq = x.freq + y.freq
8       INSERT(Q, z)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

If a priority queue is implemented as a heap, it can be initialized in $\Theta(n)$ time. Furthermore, each heap operation requires $\Theta(\log(n))$ time. Since there are $n-1$ passes through the for-i loop, the algorithm runs in $(n\log(n))$ time.