طراحی الگوریتم ها

جلسه ۶، ۷ و ۸

ملکی مجد

مباحث

- حریصانه
  - روشی برای تحلیل و طراحی الگوریتم

- شامل
  - مسئله انتخاب فعالیت
  - قسمت های اصلی روش حریصانه
  - مسئله طراحی کد هافمن

بخش ۱۶.۴ و ۱۶.۵ جزو تمرکز درس نیستند.

مبحث الگوریتم های حریصانه از فصل ۱۶ کتاب CLRS تدریس می شود.

- در برخی موارد استفاده از برنامه نویسی پویا انجام کار اضافه است!
  - در نظر گرفتن انتخاب های مختلف و پیدا کردن بهترین آن، کار اضافه ای است.

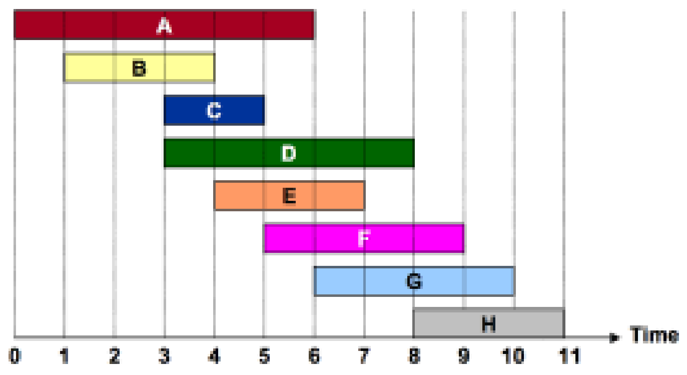
- الگوریتم حریصانه در هر مرحله انتخابی که به نظر بهترین است را برمی گزیند.
  - انتخاب به صورت محلی بهترین است و امید این است که به صورت کلی نیز بهترین باشد
  - الگوریتم حریصانه در همه مسئله ها بهترین جواب را پیدا نمی کند. (ما در این بخش درس، از الگوریتم حریصانه برای مسئله هایی استفاده می کنیم که بتوانیم بهترین جواب را پیدا کنیم.)

  - در بخش های بعدی درس نیز از روش حریصانه برای حل مسئله ها استفاده می شود. برای مثال می توان مسئله درخت فراگیر بهینه (MST) و الگوریتم دایجسترا برای کوتاهترین مسیرها از یک راس را نام برد.

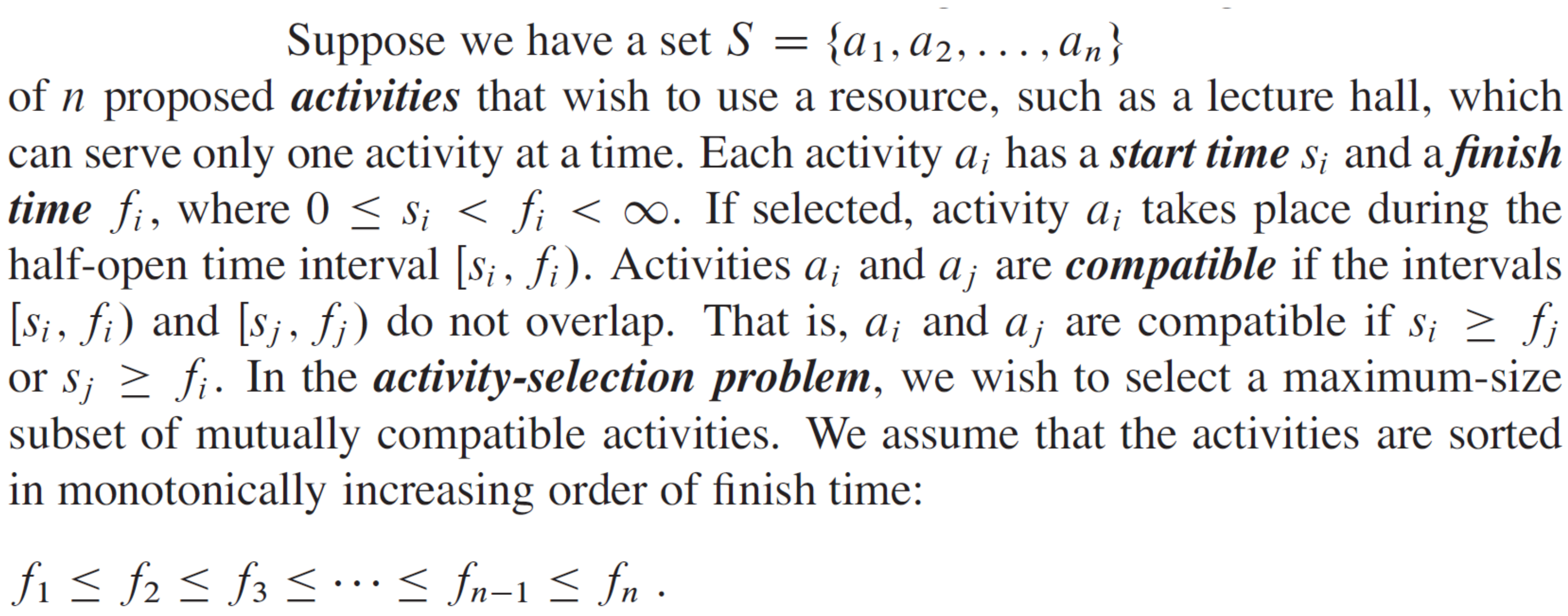
Activity selection مسئله انتخاب فعالیت

scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities.

we have a set $S = \{a_1, a_2, \dots a_n\}$ of **n proposed activities** that wish to use a resource, such as a lecture hall, which can serve only **one activity at a time**.

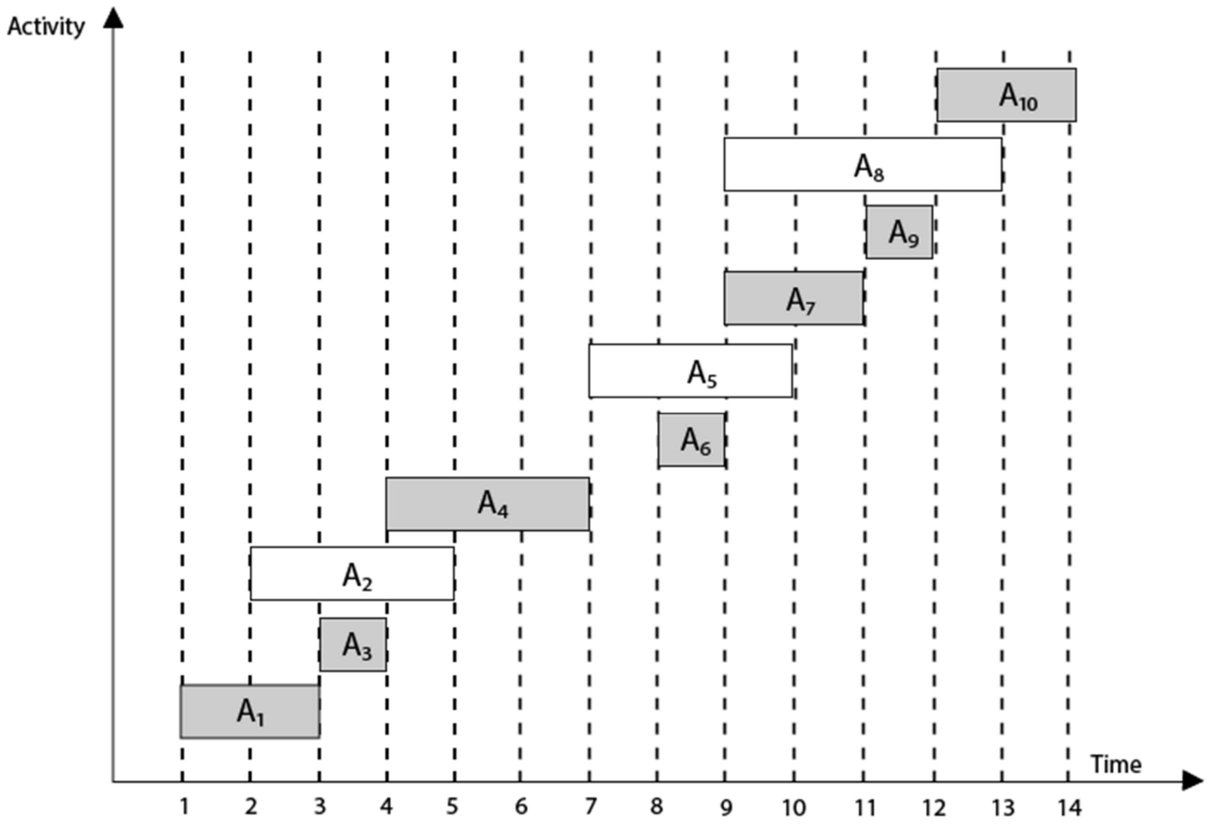


Formal definition of activity problem

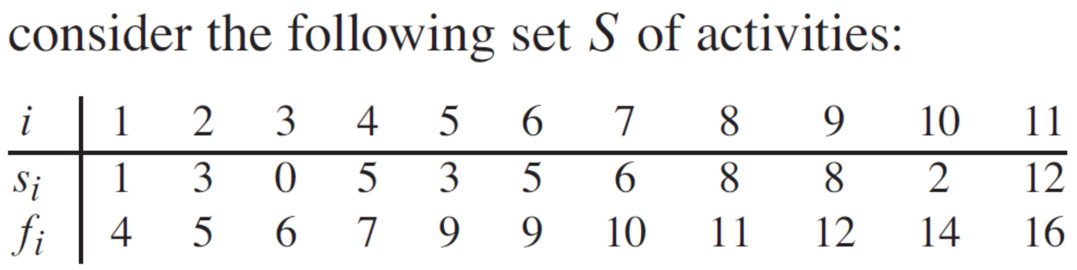


Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed ***activities*** that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity a_i has a ***start time*** s_i and a ***finish time*** f_i, where $0 \leq s_i < f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$. In the ***activity-selection problem***, we wish to select a maximum-size subset of mutually compatible activities. We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n .$$


Activity
Time
1
2
3
4
5
6
7
8
9
10
11
12
13
14
A_{10}
A_8
A_9
A_7
A_5
A_6
A_4
A_2
A_3
A_1

An example



consider the following set S of activities:

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

Find solution:

An example

Find solution

$\{a_3, a_9, a_{11}\}$: consists of mutually compatible activities however it is not a maximum subset

$\{a_1, a_4, a_8, a_{11}\}$: a largest subset of mutually compatible activities

$\{a_2, a_4, a_9, a_{11}\}$: a largest subset of mutually compatible activities

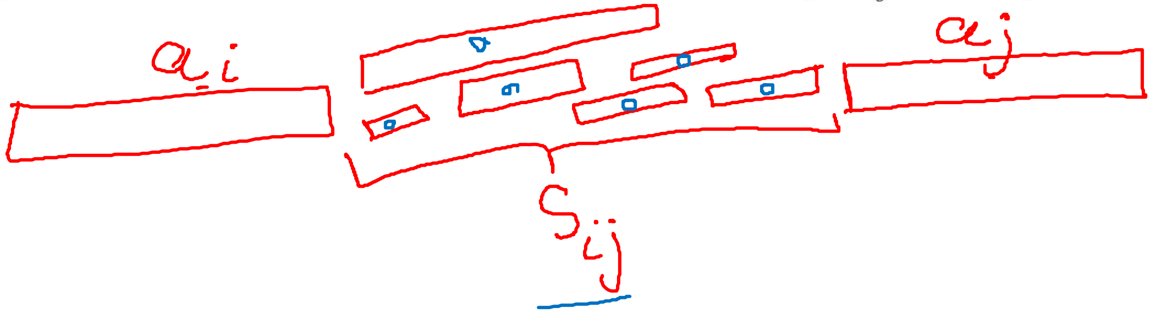
مراحل حل مسئله

- فکر کردن در مورد استفاده از برنامه نویسی پویا
  - در نظر گرفتین چندین انتخاب در هر مرحله
- مشاهده اینکه در هر مرحله نیاز داریم تنها یک انتخاب درنظر بگیریم
- ارایه راه حل بازگشتی با انتخاب حریصانه در هر مرحله
- تبدیل راه حل بازگشتی به راه حل با حلقه تکرار

- یاداوری – در عمل نیاز به طی این فرآیند با جزییات نیست و عموما ایده حریصانه زده می شود و بررسی می شود آیا جواب بهینه را پیدا می کند یا خیر

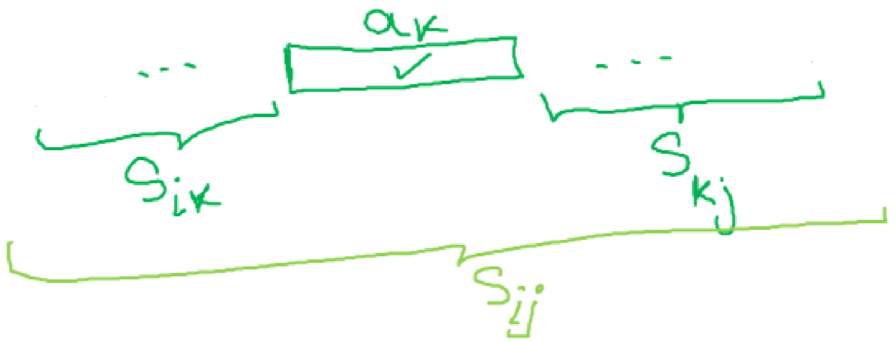
Optimal substructure

Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij}, and suppose further that such a maximum set is A_{ij}, which includes some activity a_k. By including a_k in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts).

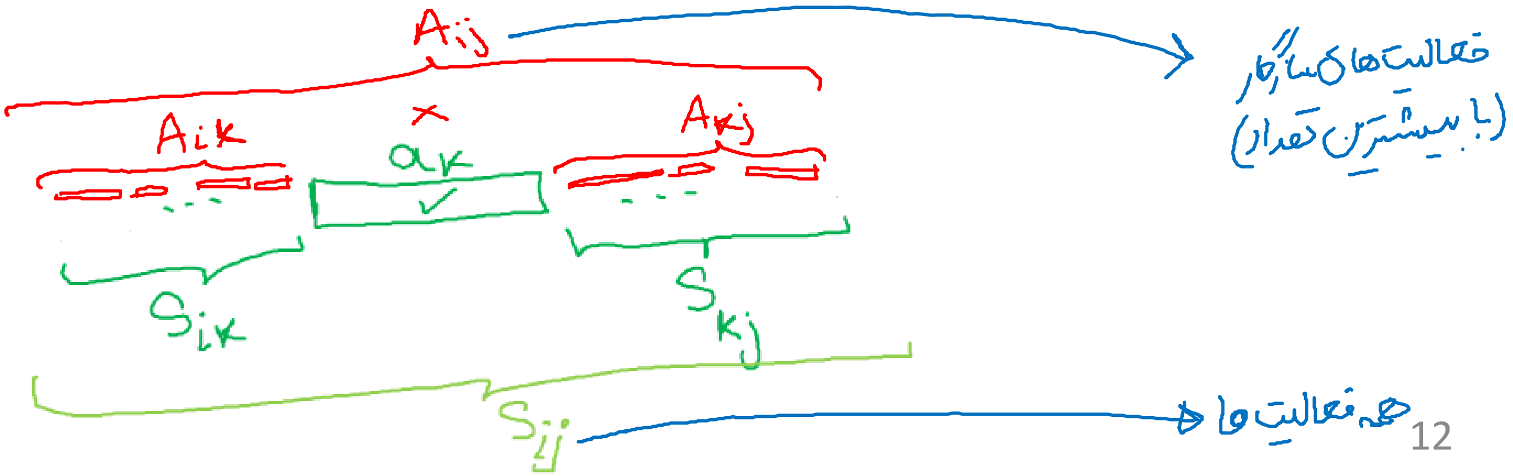


Optimal substructure

Let us denote by S_{ij} the set of activities that start after activity a_i finishes and that finish before activity a_j starts. Suppose that we wish to find a maximum set of mutually compatible activities in S_{ij}, and suppose further that such a maximum set is A_{ij}, which includes some activity a_k. By including a_k in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts).



Let $A_{ik} = A_{ij} \cap S_{ik}$
and $A_{kj} = A_{ij} \cap S_{kj}$, so that A_{ik} contains the activities in A_{ij} that finish before a_k starts and A_{kj} contains the activities in A_{ij} that start after a_k finishes. Thus, we have $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} of mutually compatible activities in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities.



Cut and past

in this problem

The usual cut-and-paste argument shows that the optimal solution A_{ij} must also include optimal solutions to the two subproblems for S_{ik} and S_{kj}. If we could find a set A'_{kj} of mutually compatible activities in S_{kj} where $|A'_{kj}| > |A_{kj}|$, then we could use A'_{kj}, rather than A_{kj}, in a solution to the subproblem for S_{ij}. We would have constructed a set of $|A_{ik}| + |A'_{kj}| + 1 > |A_{ik}| + |A_{kj}| + 1 = |A_{ij}|$ mutually compatible activities, which contradicts the assumption that A_{ij} is an optimal solution. A symmetric argument applies to the activities in S_{ik}.

DP – button up

This way of characterizing optimal substructure suggests that we might solve the activity-selection problem by dynamic programming. If we denote the size of an optimal solution for the set S_{ij} by $c[i, j]$, then we would have the recurrence

$$c[i, j] = c[i, k] + c[k, j] + 1 \; .$$

Of course, if we did not know that an optimal solution for the set S_{ij} includes activity a_k, we would have to examine all activities in S_{ij} to find which one to choose, so that

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

Greedy choice

Of the activities we end up choosing, one of them must be the first one to finish.

choose **the activity in S with the earliest finish time**, since that would leave the resource available for **as many of the activities** that follow it as possible

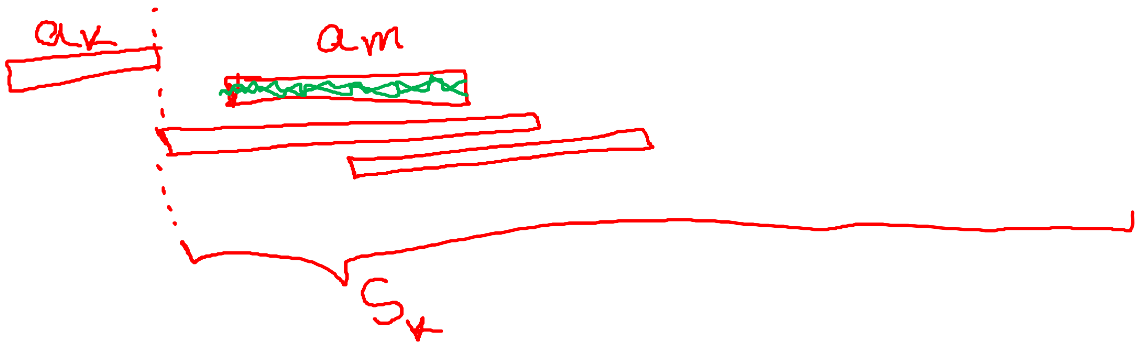
If we make the greedy choice, we have only one remaining subproblem to solve:

finding activities that start after a_1 finishes

Why don't we have to consider activities that finish before a1 starts?

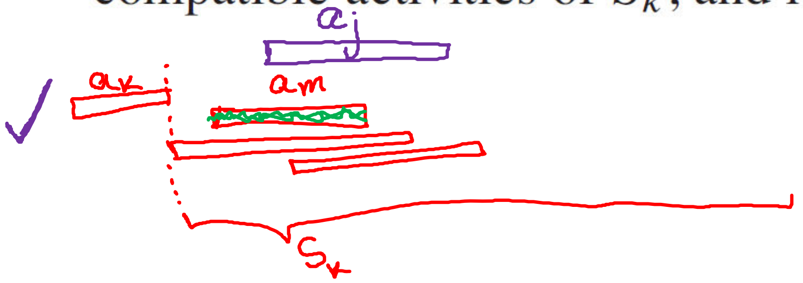
is our intuition correct?

Theorem 16.1
Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.



Let $S_k = \{a_i \in S : s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes.

Proof Let A_k be a maximum-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since we have shown that a_m is in some maximum-size subset of mutually compatible activities of S_k. If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituting a_m for a_j. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m. ■

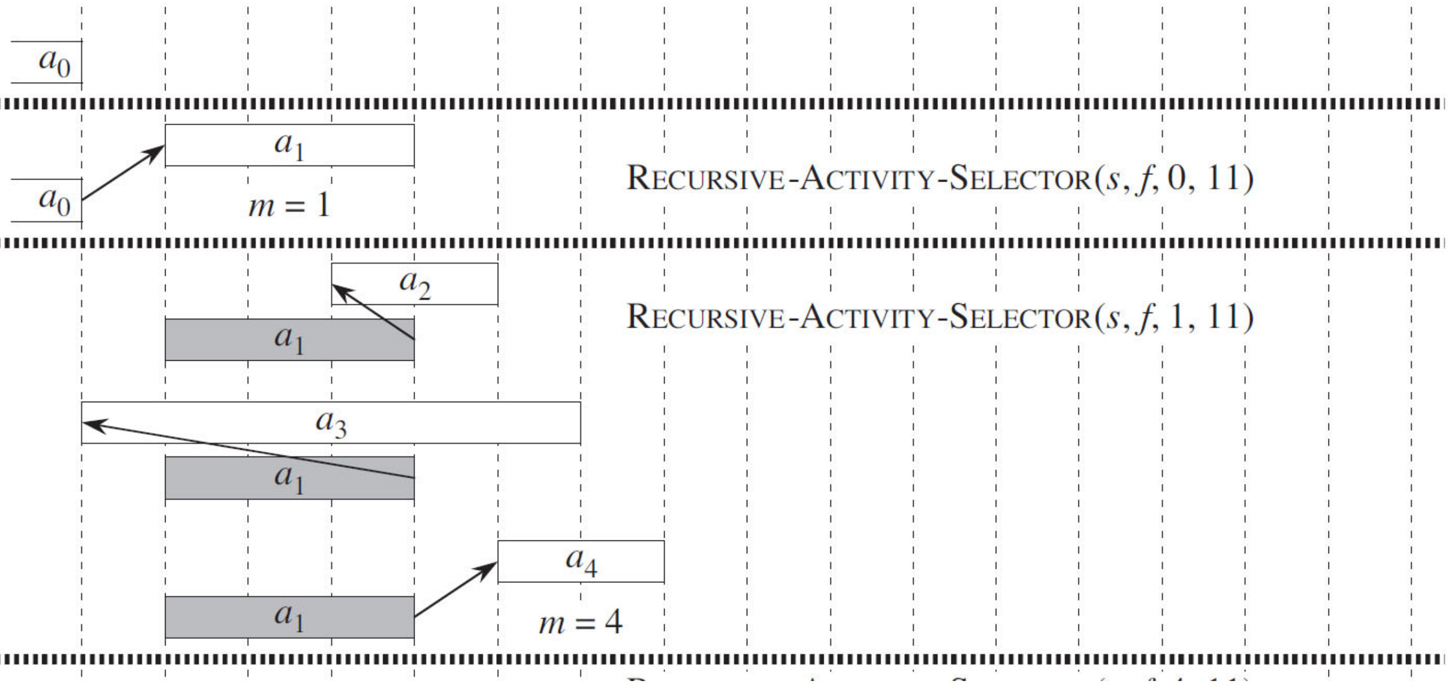


Recursive greedy algorithm

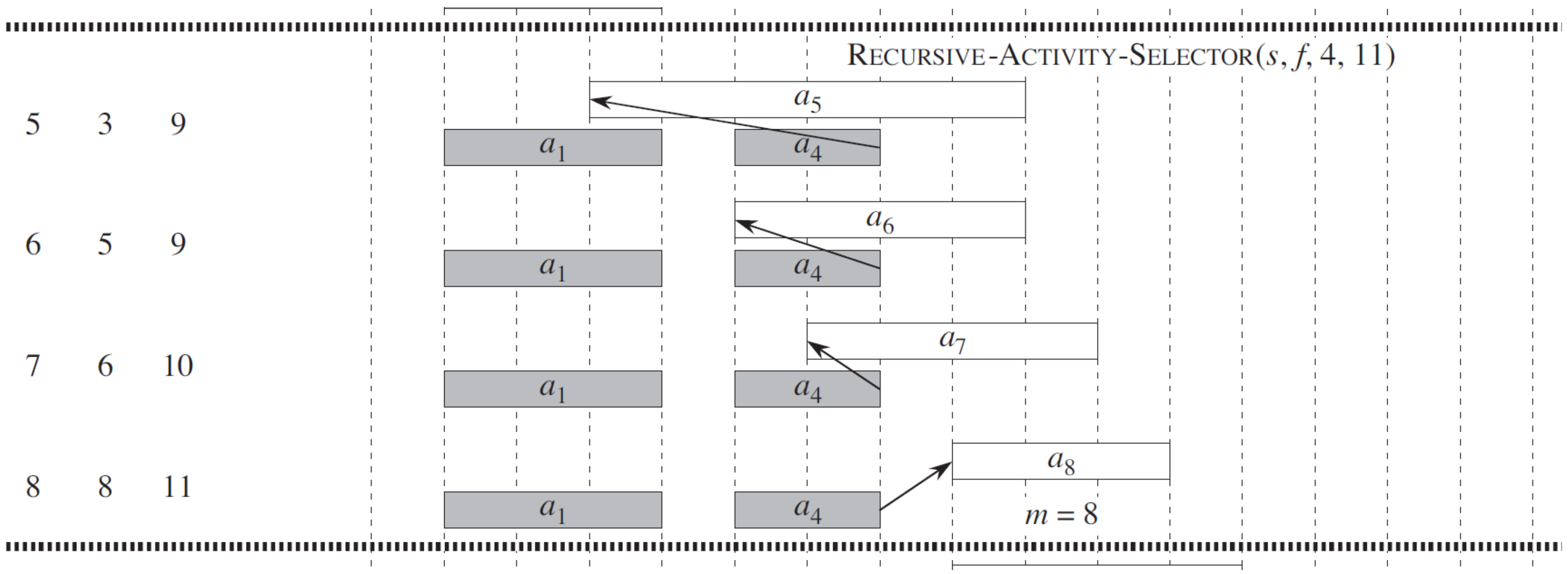
```
RECURSIVE-ACTIVITY-SELECTOR$(s, f, k, n)$
1  $m = k + 1$
2  while $m \le n$ and $s[m] < f[k]$        // find the first activity in $S_k$ to finish
3      $m = m + 1$
4  if $m \le n$
5      return $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR$(s, f, m, n)$
6  else return $\emptyset$
```

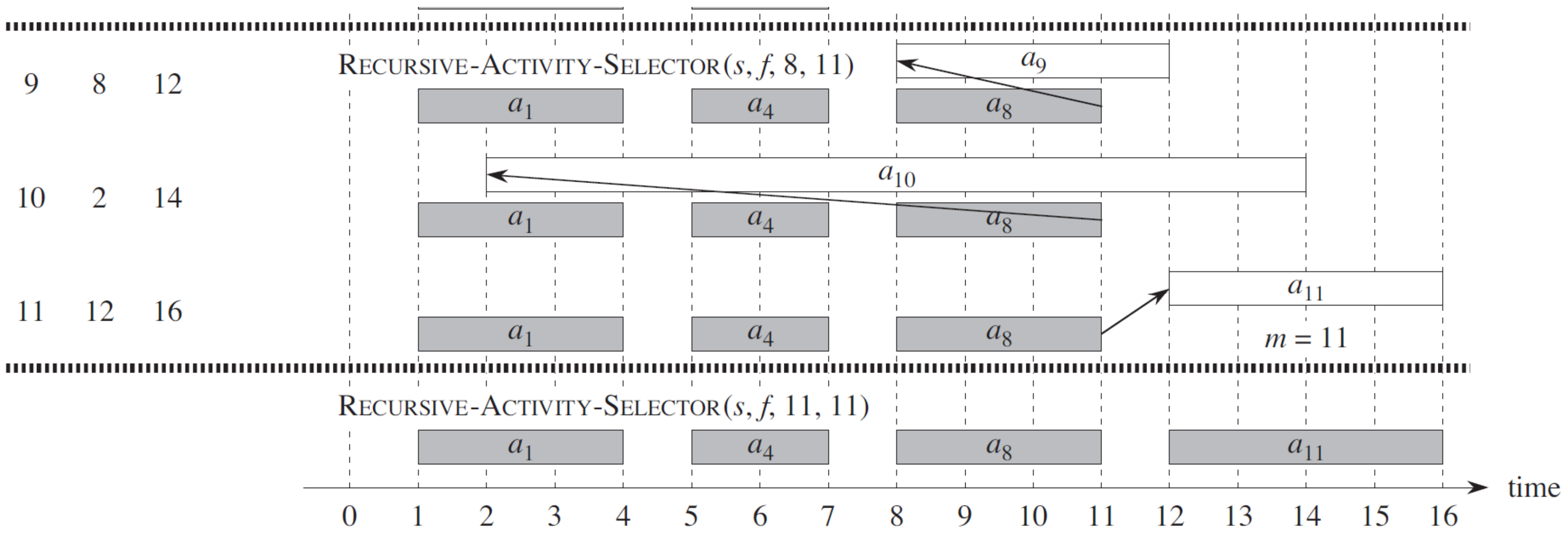
Let $S_k = \{a_i \in S : s_i \ge f_k\}$ be the set of activities that start after activity a_k finishes.

k	s_k	f_k
0	–	0
1	1	4
2	3	5
3	0	6
4	5	7



a_0
a_0
a_1
$m = 1$
RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, 11$)
a_2
a_1
RECURSIVE-ACTIVITY-SELECTOR($s, f, 1, 11$)
a_3
a_1
a_4
a_1
$m = 4$


5 3 9
6 5 9
7 6 10
8 8 11
RECURSIVE-ACTIVITY-SELECTOR$(s, f, 4, 11)$
a_5
a_1
a_4
a_6
a_1
a_4
a_7
a_1
a_4
a_8
a_1
a_4
$m = 8$



RECURSIVE-ACTIVITY-SELECTOR$(s, f, 8, 11)$
9 8 12
a_9
a_1
a_4
a_8
10 2 14
a_{10}
11 12 16
a_{11}
$m = 11$
RECURSIVE-ACTIVITY-SELECTOR$(s, f, 11, 11)$
time
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Iterative greedy algorithm

```
GREEDY-ACTIVITY-SELECTOR(s, f)
1  $n = s.length$
2  $A = \{a_1\}$
3  $k = 1$
4  for $m = 2$ to $n$
5      if $s[m] \ge f[k]$
6          $A = A \cup \{a_m\}$
7          $k = m$
8  return $A$
```

Time complexity

Both versions, schedule a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times.

Greedy elements

Greedy heuristic

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices.

At each decision point, the algorithm makes choice that seems best at the moment.

This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does.

Elements of the greedy strategy

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subproblem remains.
4. Prove that it is always safe to make the greedy choice. (Steps 3 and 4 can occur in either order.)
5. Develop a recursive algorithm that implements the greedy strategy.
6. Convert the recursive algorithm to an iterative algorithm.

General steps of designing Greedy algorithms

I. Cast the optimization problem as one in which we **make a choice** and are **left with one subproblem** to solve.

II. 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.

III. Demonstrate **optimal substructure** by showing 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

Assemble a globally optimal solution by making locally optimal (greedy) choices.

    not considering results from subproblems!

DP versus Greedy

In **dynamic programming**:

we make a choice at each step, but the choice usually depends on the solutions to subproblems

progressing from smaller subproblems to larger subproblems

In a **greedy algorithm:**

we make whatever choice seems best at the moment and then solve the subproblem that remains.

The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to subproblems

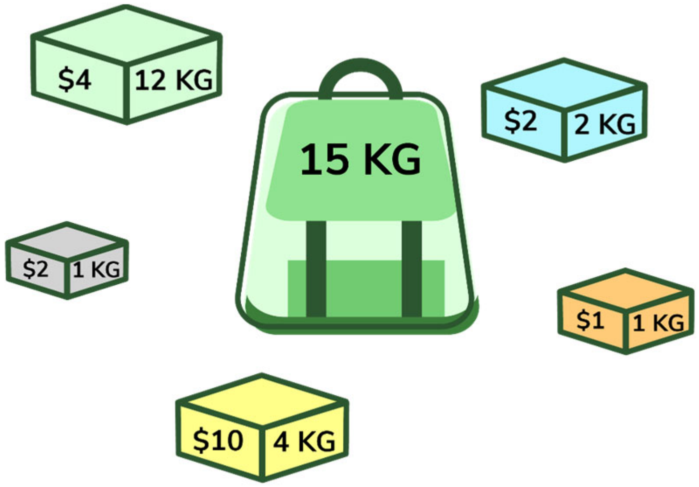
making its first choice before solving any subproblems

knapsack problem

two versions regarding DP and Greedy

A thief robbing a store, finds $\boldsymbol{n}$ **items** while **item** $\boldsymbol{i}$ is worth $\boldsymbol{v_i}$ dollars and weighs $\boldsymbol{w_i}$ pounds.

The thief wants to take as valuable a load as possible, but he can carry at most $\boldsymbol{W}$ pounds in his knapsack.



knapsack problem

two versions regarding DP and Greedy

A thief robbing a store, finds $\boldsymbol{n}$ **items** while **item** $\boldsymbol{i}$ is worth $\boldsymbol{v_i}$ dollars and weighs $\boldsymbol{w_i}$ pounds.
The thief wants to take as valuable a load as possible but he can carry at most $\boldsymbol{W}$ pounds in his knapsack.

- 0-1 knapsack problem
  - vi, wi and W are integers
  - the thief must either take an item or leave it behind; he cannot take a fractional amount of an item or take an item more than once.
- fractional knapsack problem
  - the thief can take fractions of items

You can think of an item in the 0-1 knapsack problem as being like a **gold ingot** and an item in the fractional knapsack problem as more like **gold dust**.

knapsack problems : optimal-substructure property.

0-1 problem

consider the **most valuable** load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j .

fractional problem

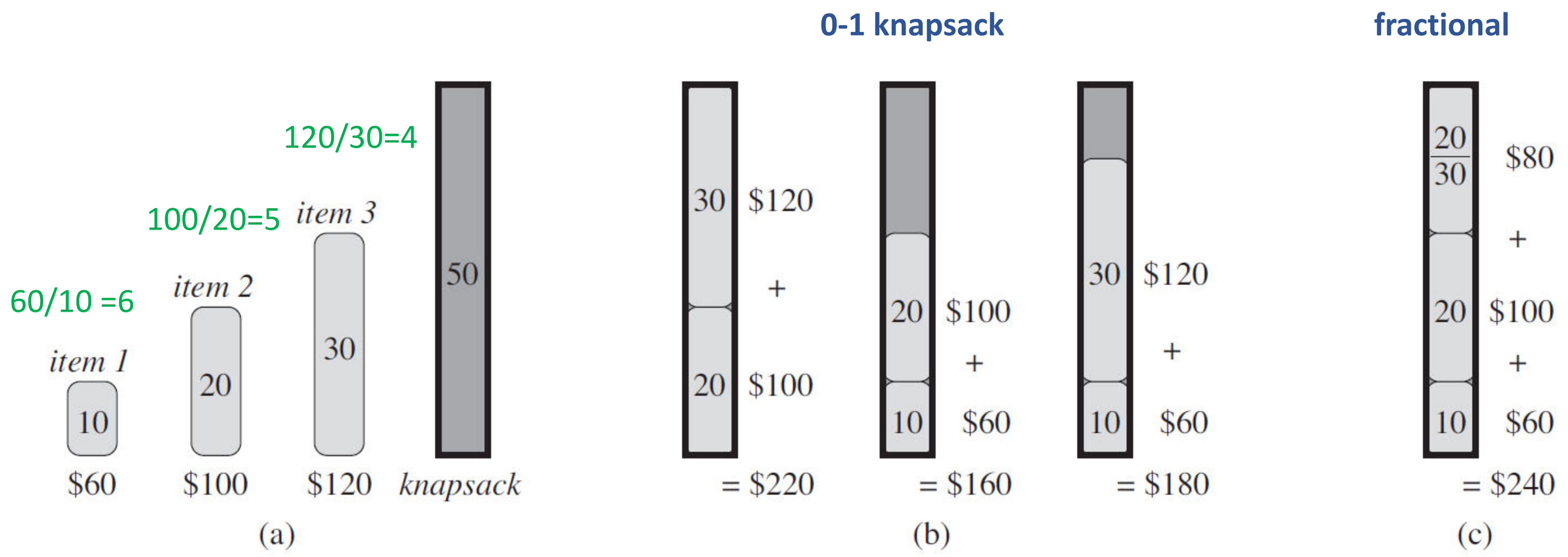
consider that if we remove a weight w of one item j from the optimal load, the remaining load must be the most valuable load weighing at most $W - w$ that the thief can take from the $n - 1$ original items plus $w_j - w$ pounds of item j.

Knapsack solution

- fractional knapsack
  - Greedy heuristics
  - Compute the value per pound v_i/w_i for each item
  - Take as much as possible of the item with the greatest value per pound.
  - Continue until reaching the weight limit
  - Runs in $O(n \lg n)$ time

- 0-1 knapsack
  - Greedy not apply! (next page example)
  - Solve it by DP as an exercise.

An example



0-1 knapsack
fractional
60/10 =6
100/20=5
120/30=4
item 1
item 2
item 3
10
20
30
50
$60
$100
$120
knapsack
(a)
30 $120
+
20 $100
= $220
20 $100
+
10 $60
= $160
30 $120
+
10 $60
= $180
(b)
20/30 $80
+
20 $100
+
10 $60
= $240
(c)

Huffman codes

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

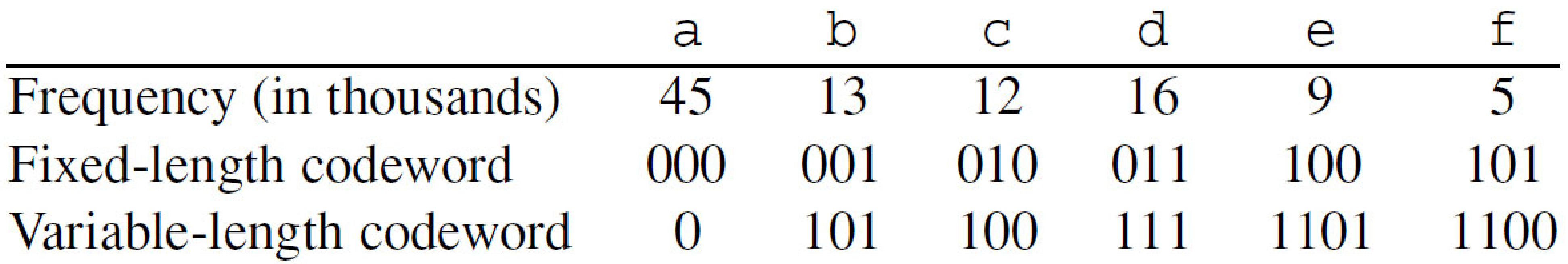
- consider the problem of designing a ***binary character code*** (or ***code*** for short) wherein each character is represented by a unique binary string.

	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.
    - This method requires 224,000 bits to code the entire file
    - a savings of approximately 25%.



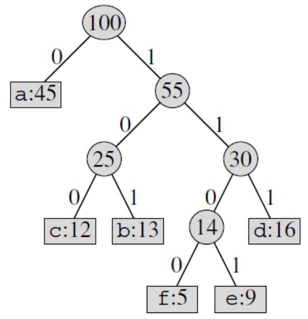
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

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

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

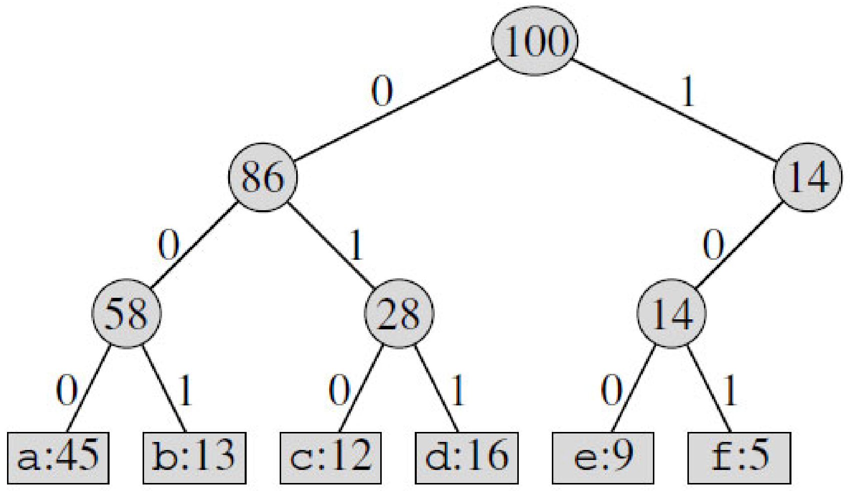
Decoding process

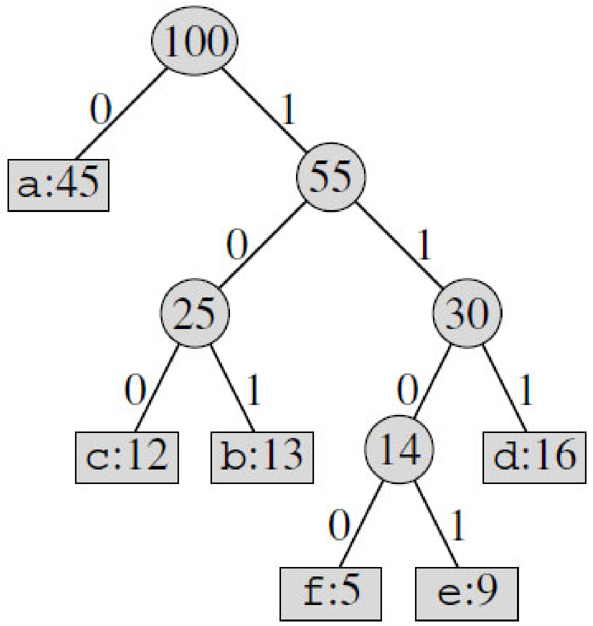


- An optimal code for a file is always represented by a ***full* binary tree**, in which every nonleaf node has two children
  - A binary tree whose leaves are the given characters provides a convenient representation

- 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,

	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





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

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

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 of Huffman's algorithm

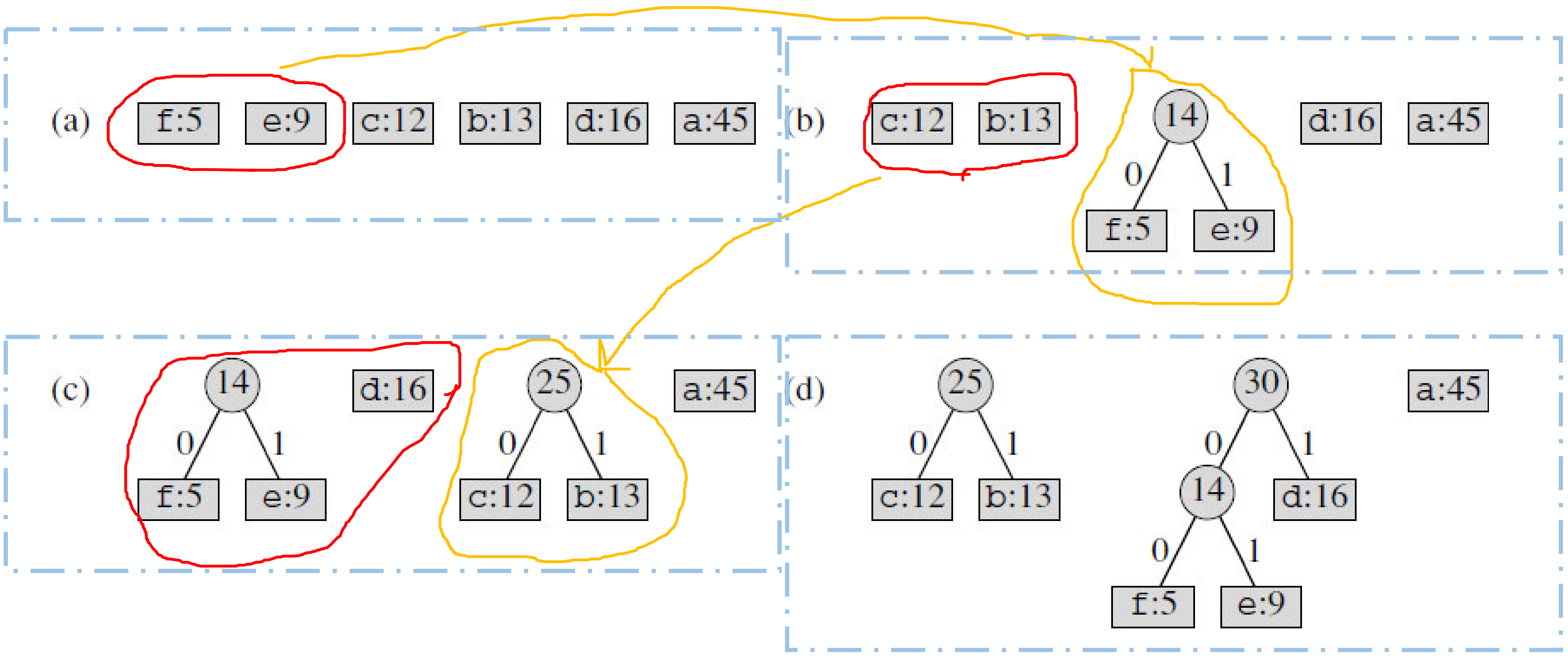
```
HUFFMAN($C$)
$n \leftarrow |C|$
$Q \leftarrow C$
for $i \leftarrow 1$ to $n - 1$
    do allocate a new node $z$
       $left[z] \leftarrow x \leftarrow$ EXTRACT-MIN($Q$)
       $right[z] \leftarrow y \leftarrow$ EXTRACT-MIN($Q$)
       $f[z] \leftarrow f[x] + f[y]$
       INSERT($Q$, $z$)
return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

Running time

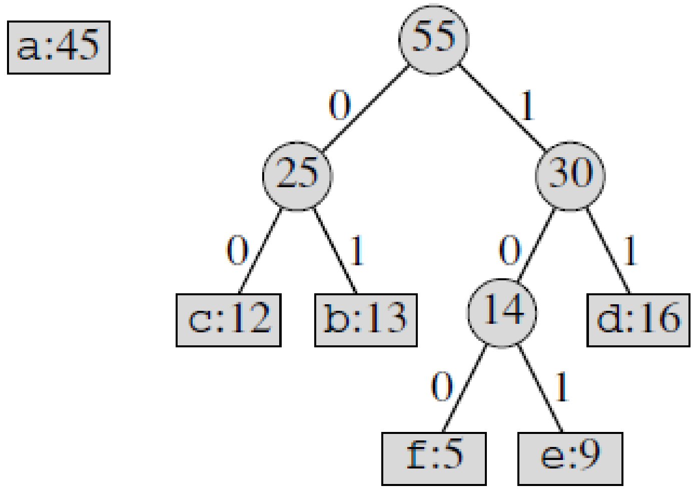
- $O(n \lg n)$

Construct tree

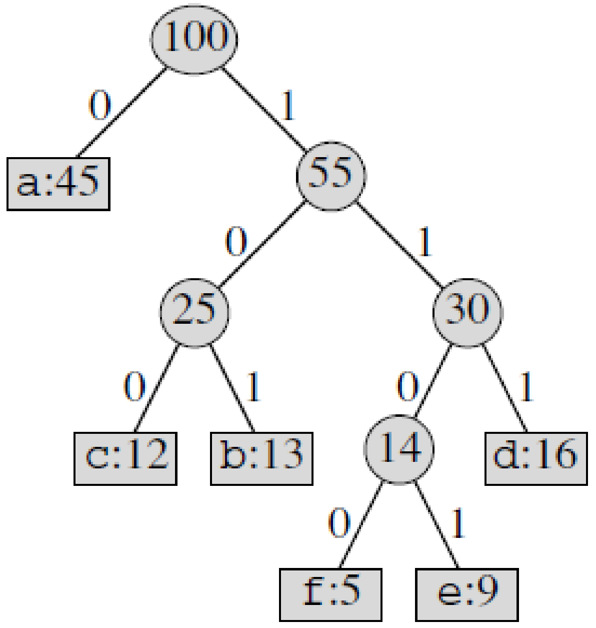
Each part shows the contents of the queue sorted into increasing order by frequency



(e)


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

(f)

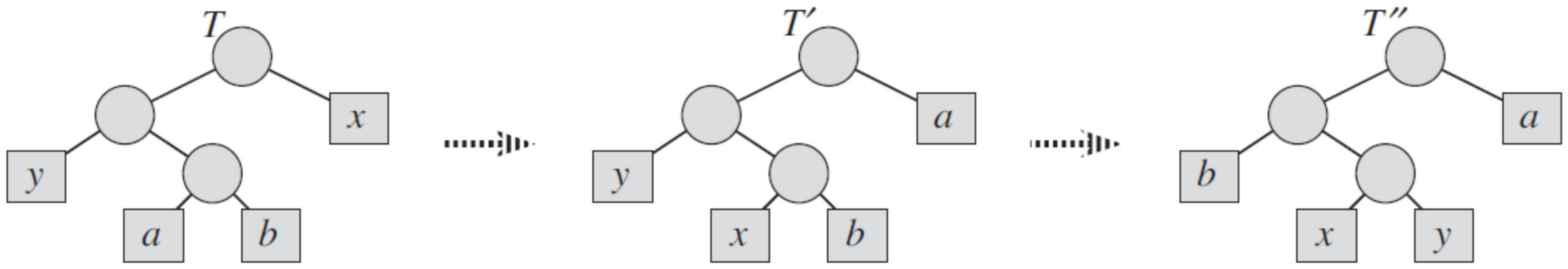

100
0
1
a:45
55
0
1
25
30
0
1
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

Correctness of Huffman’s algorithm

Lemma 16.2

Let C be an alphabet in which each character $c \in C$ has frequency $f[c]$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

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



Lemma 16.3

Let C be a given alphabet with frequency $c.\mathit{freq}$ defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C' be the alphabet C with the characters x and y removed and a new character z added, so that $C' = C - \{x, y\} \cup \{z\}$. Define f for C' as for C, except that $z.\mathit{freq} = x.\mathit{freq} + y.\mathit{freq}$. Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.

Proof

idea:

Express the cost $B(T)$ tree T in terms of the cost $B(T')$ of tree T'
Then use contradiction.

Theorem 16.4

Procedure HUFFMAN produces an optimal prefix code.

Proof: Immediate from Lemmas 16.2 and 16.3.