

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. Greedy algorithms **do not** always yield **optimal solutions**, but for many problems they do.

An **important application** of greedy techniques: the design of **data-compression (Huffman)** codes. (16.3 p.8 會教)

16.1 An Activity-selection problem

Our first example is the problem of **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.

Suppose we have a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed activities that wish to use **a resource** which can be used by **only one activity at a time**. Each activity a_i has a start time s_i and a finish time f_i , where $0 \leq s_i \leq f_i \leq \infty$. If selected activity a_i takes place during **time interval** $[s_i, f_i)$. Two activities a_i and a_j are **compatible** (相容) if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ **do not overlap**.

The activity-selection problem is to select a **maximum-size** subset of mutually **compatible activities**.

For example,

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

(完成時間 由小到大排)

Then, the largest subset is $\{a_1, a_4, a_8, a_{11}\}$ or $\{a_2, a_4, a_9, a_{11}\}$.

First, we apply DP to solve this problem:

The optimal substructure of the activity-selection

Let us define a space of subproblem $S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$.

S_{ij} is the **subset of activities** in S that **start after a_i finished** and **before a_j start**. (工作 a_i 完成後 到工作 a_j 開始前，中間空的時間，可以進行工作的集合 = S_{ij}) Our solution to S_{ij} is the **union** of solution to S_{ik} and S_{kj} , along with the **activity a_k** . The optimal substructure of this problem is as follows. We define the optimal solution A_{ij} to S_{ij} **includes activity a_k** . Then

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}.$$

The optimal solution to the entire problem is a solution to $S_{0, n+1}$

A recursive solution

Developing a DP solution is to recursively define the value of an optimal solution. $C[i, j]$ is the **number of activities** in a maximum-size of subset of mutually compatible activities in S_{ij} . $C[i, j] = 0$, if $S_{ij} = \emptyset$ or $i \geq j$. If a_k is used in the optimal solution S_{ij} , then

$$C[i, j] = C[i, k] + C[k, j] + 1.$$

The recursive relation of $C[i, j]$ becomes

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

$$c[0,1] = c[1,2] = \dots = c[n, n+1] = 0, \quad c[i, j], j > i. i=0 \sim n; j=1 \sim n+1$$

Converting a DP solution to a greedy solution

Theorem 16.1 For any nonempty subproblem S_{ij} , and a_m is 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 S_{ij} .

(a_m 被選入 S_{ij} 可形成 最大子集)

2. The subproblem S_{im} is empty, so that choose a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

(S_{im} 一定為 empty, 只有剩下的 S_{mj} 可能不是 empty).

In DP solution, two subproblems are used in optimal solution, and there are $j-i-1$ choices for solving S_{ij} .

Theorem 16.1 Only one subproblem is used in an optimal solution: the one with the earliest finish time in S_{ij} .

In addition to reducing the number of subproblems and choices, theorem 16.1 yields another benefit: we can solve each subproblem in a top-down fashion, rather than the bottom-up manner (DP).


The activity a_m that we choose when solving a subproblem is always the one with the earliest finish time that can be legally scheduled.

A recursive greedy algorithm

We give a straightforward, recursive solution as the procedure RECURSIVE-ACTIVITY-SELECTOR. Two arrays s and f are used for the start and finish times, as well as the indices i and n that define the subproblem $S_{i, n+1}$ it is to solve. It returned a max-size set of mutually compatible activities in $S_{i, n+1}$. The sorting for finish time is $O(n \log n)$, and the initial call is RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, n$). (idea: 由排序好的 finish time 中, 找範圍 $S_{i, n+1}$ 下一個 start time m 比目前 activity a_i finish time 晚的 activity.)

RECURSIVE-ACTIVITY-SELECTOR(s, f, i, n)

```
1   $m \leftarrow i + 1$ 
2  while  $m \leq n$  and  $s_m < f_i$     ▷ Find the first activity in  $S_{i,n+1}$ .
3      do  $m \leftarrow m + 1$ 
4  if  $m \leq n$ 
5      then return  $\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6  else return  $\emptyset$ 
```

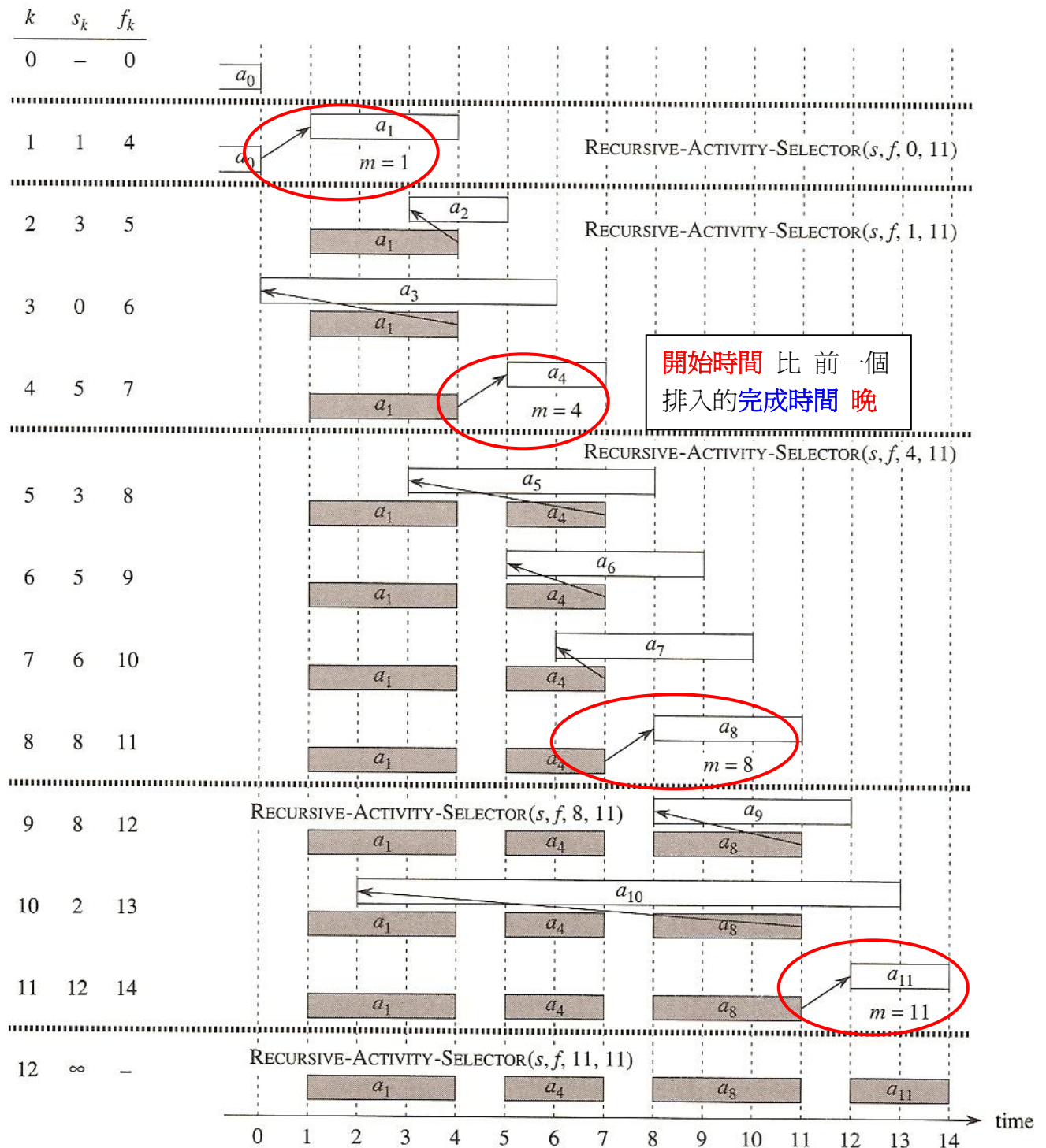


The loop examine a_{i+1}, a_{i+2}, \dots , until it finds the **first activity** a_m that is **compatible with** a_i ; such an activity has $s_m \geq f_i$. (執行前要先以完成時間 **sorting** 完，再呼叫此副程式)

Each activity is **examined exactly once** in while loop test of **line 2**. The **running time** of the call this procedure is $\Theta(n)$. **Figure 16.1** shows the procedure. Result: $\{a_1, a_4, a_8, a_{11}\}$

RECURSIVE-ACTIVITY-SELECTOR($s, f, 0, 11$)

$a_1 \sim a_{11}$ 已經以 完成時間 排序：



An iterative greedy algorithm.

It is usually a straightforward task to transform a **tail-recursive** (剩下部份 recursive) procedure to an **iterative form**. (loop 方式完成)

GREEDY-ACTIVITY-SELECTOR(s, f)

```

1   $n \leftarrow \text{length}[s]$ 
2   $A \leftarrow \{a_1\}$ 
3   $i \leftarrow 1$ 
4  for  $m \leftarrow 2$  to  $n$ 
5      do if  $s_m \geq f_i$ 
6          then  $A \leftarrow A \cup \{a_m\}$ 
7               $i \leftarrow m$ 
8  return  $A$ 

```

Line 2: 第一個最早結束的活動(a_1)先放入 A 中。

Lines 4~7: 以結束時間依序檢查： a_m 起始時間比剛加入 set A 的活動 a_i 結束時間晚(為一合法活動)，則加入。

The GREEDY-ACTIVITY-SELECTOR schedules a set of n activities in time, assuming that the activities were already sorted initially by their finish times.

Exercise: An activity selection problem is shown as followed:

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

用 Greedy algorithm 求解(說明：演算法、程式、結果)

16.2 Elements of the greedy strategy

Develop a greedy algorithm often went through the following steps:

1. Determine the optimal substructure of the problem.
2. Develop a recursive solution.
3. Prove that any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy

choice.

4. Show that **all but** one of the subproblems induced by having made the **greedy choice** are **empty**. (所有最佳解**非 greedy** 產生的為**空集合**；即**非 greedy** 產生的解均**非最好的**)
5. Develop a **recursive algorithm** that implements the greedy strategy.
6. Convert the recursive algorithm to an **iterative algorithm**.
(熟了後，就可以直接以 **iterative algorithm** (迴圈演算法)解)

However, more generally, we design **greedy algorithms according to the following sequence** of steps.

1. Cast (鑄造) the **optimal problem** as one in which we **make a choice** and are left with **one** subproblem to solve. (一個最佳解問題，是當做一決定後，剩下的 **subproblem** 也是求最佳解)。
2. **Prove** that there is always **an optimal solution** the original problem that makes the **greedy choice**, so that the greedy choice is always safe. (此部分較困難)
3. **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 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 a greedy algorithm, we make whatever choice seems **best at the moment** and then solve the subproblem arising after the choice is made. A greedy strategy usually progresses in a **top-down** fashion.

Optimal substructure

A problem exhibits **optimal substructure** if an optimal solution to the **problem** contains within it optimal solutions to **subproblems**. For example, in the activity schedule problem, if an optimal solution to subproblem S_{ij} **includes an activity a_k** , then it must also contain **optimal solutions** to the subproblems S_{ik} and S_{kj} .

Greedy vs. DP

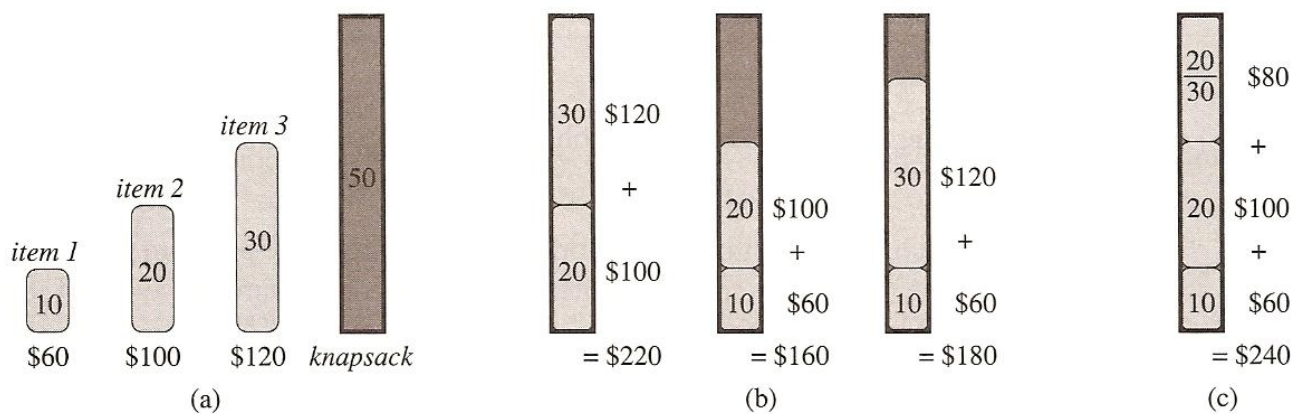
We investigate **two variants** of a classical optimization problem.

1. The 0-1 knapsack problem

A thief robbing a store finds n items; the i th item is worth v_i **dollars** and **weights w_i** pounds, where v_i and w_i are **integers**. He wants to take as valuable a **load as possible**, but he can carry **at most W pounds**. (an NPC problem (ch. 34)介紹概念)

2. The fractional knapsack problem

Like the 0-1 knapsack problem, but the thief can take **fractions** of items. The **fractional knapsack** problem is solvable by a **greedy strategy**, whereas the **0-1 problem is not**. To solve the fractional problem, we first compute the value v_i/w_i for each item. And then solve them by **decreasing order** and carry them. The greedy algorithm runs in $O(n \log n)$. **Figure 16.2 (b) shows 0-1 knapsack problem can not be solved by greedy algorithm**, and **16.2 (c) yields an optimal solution of fractional knapsack problem by greedy strategy**.



16.3 Huffman codes

Huffman codes are a widely used and very effective technique for **compressing data**. Huffman's greedy algorithm uses a table of frequencies of occurrence of the **100,000 characters** to build up an optimal way of representing each character as a **binary string**. **Figure 16.3** shows “a” occurs 45,000 times among **six** different characters.

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

If we use a **fixed-length code**, we need **3 bits** to represent **six characters**. Then, we required **300,000 bits** (全部 100,000 characters).

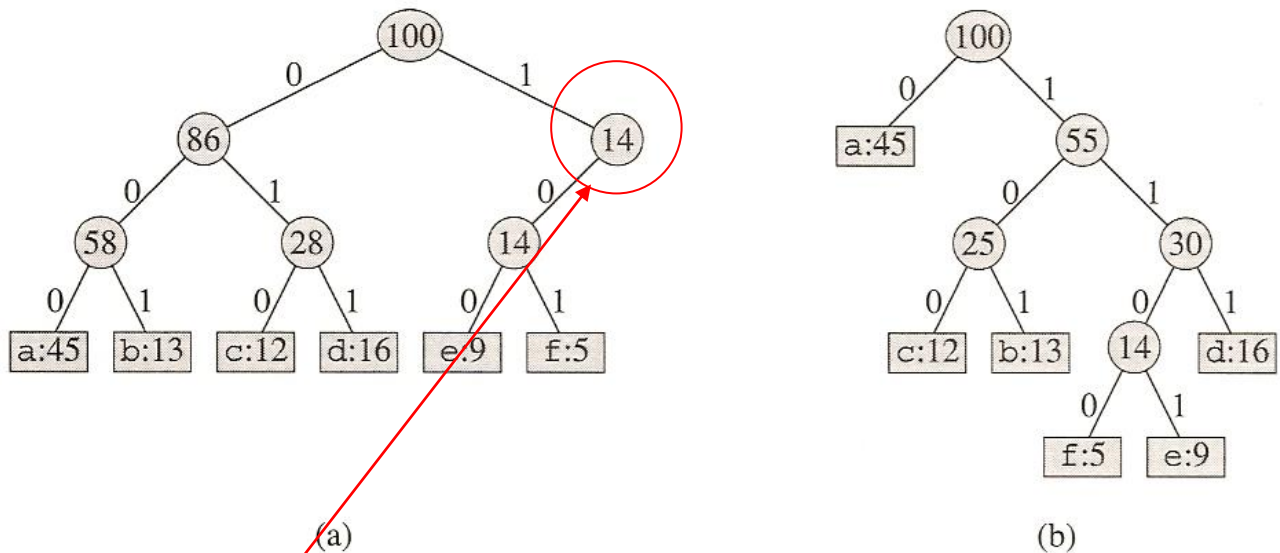
A **variable-length code** can do better than a fixed-length code. For example, a bit **0** for “a”, this code requires

$$(45 \times 1 + 13 \times 3 + 12 \times 3 + 16 \times 3 + 9 \times 4 + 5 \times 4) \times 1000 = \mathbf{224,000 \text{ bits}}$$

It **saves** approximately **25%**.

Prefix codes

We only consider codes in which **no codeword** is also a **prefix of some other codeword**. Such codes are called **prefix codes** (**prefix-free codes** would be better). A binary tree whose **leaves** can be used to represent the **codes**. Figure 16.4 shows the tree ((a) is **fixed length**; (b) is **variable length**).



An **optimal code** for a file is always represented by a **full binary tree**, (**滿二元樹**) in which every **nonleaf node** two children. Figure 16.4 (a) is **not optimal**.

For a file with **characters** set **C**, then there are **|C| leaves** and **| C-1 | internal nodes**. For each character **c** in **C**, **f(c)** denotes the **frequency of c** and **d_T(c)** denote the **depth** of **c's leaf** in the tree. (**from root**) Then, the **number of bits** required to encode a **file** is

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

The **cost of the tree T**.

Constructing a Huffman code

Huffman invented a **greedy algorithm** that constructs an optimal prefix code called a Huffman code. The optimal code is constructed 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 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.

```

HUFFMAN( $C$ )
1   $n \leftarrow |C|$ 
2   $Q \leftarrow C$ 
3  for  $i \leftarrow 1$  to  $n - 1$ 
4      do allocate a new node  $z$ 
5           $left[z] \leftarrow x \leftarrow \text{EXTRACT-MIN}(Q)$ 
6           $right[z] \leftarrow y \leftarrow \text{EXTRACT-MIN}(Q)$ 
7           $f[z] \leftarrow f[x] + f[y]$ 
8           $\text{INSERT}(Q, z)$ 
9  return  $\text{EXTRACT-MIN}(Q)$             $\triangleright$  Return the root of the tree.

```

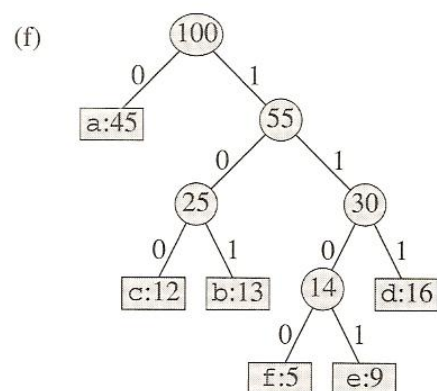
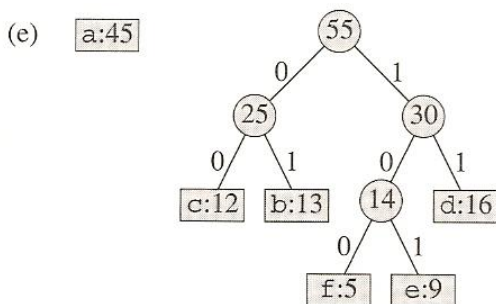
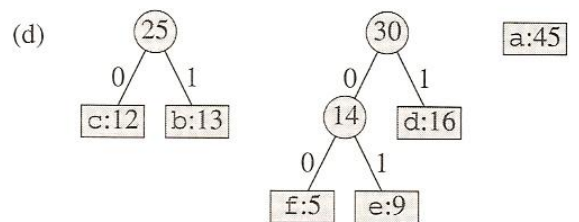
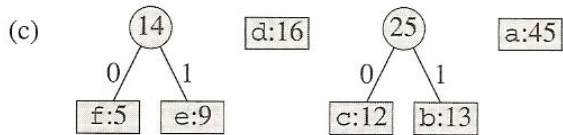
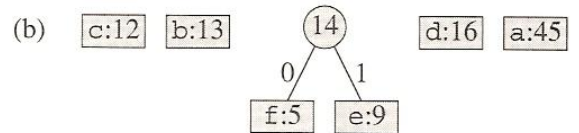
Line 4: 加入一個**新 node z**

Line 5~7: 將 Q 中最小兩個**取出**(Q 就少兩個 **data**)，**相加結果**放入 **z**

Line 8: 將 z 放入 Q 中。

Figure 16.5 shows the procedure of the algorithm. There are **6 letters** in the alphabet, the initial queue size is $n = 6$, and **5 merge steps** are required to build the tree. The Huffman’s algorithm assumes that Q is implemented as a **binary min-heap** (Chap 6). That is required **$O(n)$** time using the **BUILD-MIN-HEAP**. Each **heap operation** (**extract-min**) requires time **$O(\log n)$** , total running time in **$O(n \log n)$** .

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



16.5 A task-scheduling problem

An interesting problem is the problem of optimally scheduling **unit-time** tasks on a **single processor**, where **each task has a deadline**, along with a **penalty** that must be paid if the **deadline is missed**.

The problem of scheduling **unit-time** tasks with deadlines and **penalties** for a **single processor** has the following inputs:

N jobs, & **one machine**, and each job requires **one unit** processing **time** & a **deadline** $d_i \geq 0$ & a **profit** $p_i \geq 0$. (若是 **penalty** 則為 **minimum**)

How to order those **n jobs** s.t. obtain the **max. profit**.

[Notes: 1. The **profit** can be updated to **penalty**.

2. When the processing time $t_i \geq 0$ (**not an unit time**),

$t_i = 1, 2, \dots, n$, it becomes an **NP-hard** problem.]

Suppose J is a subset of jobs & each job in J can be finished before its deadline. Then J is **independent** (or a “**feasible solution**”).

Example : $n=4$, $(p_1, p_2, p_3, p_4) = (100, 10, 15, 27)$

$(d_1, d_2, d_3, d_4) = (2, 1, 2, 1)$

feasible solution	sequence	value	
(i) {1, 2}	2, 1	110	
(ii) {1, 3}	1, 3 or 3, 1	115	
(iii) {1, 4}	4, 1	127 optimal
(vi) {2, 3}	2, 3	25	
(v) {3, 4}	4, 3	42	
{1}	1	100	
{2}	2	10	
	.	.	

Greedy approach :

1. **Sort** the **profit** for n jobs: $j_1 \geq j_2 \geq j_3 \geq \dots \geq j_n$
2. Suppose a **feasible solution** $J = \{p_{j_1}, p_{j_2}, p_{j_3}, \dots, p_{j_{k-1}}\}$

We want to **insert a job** j_k

- (1)The j_k can be inserted into J , if it is still a **feasible solution**.
- (2)The j_k can **not** be inserted, if it **can't obtain a feasible solution**.

Then we consider the next job j_{k+1} until no job can be inserted.

To **check a feasible solution** for J : $O(k!)$ (所有可能排列)

節省 Check the feasible solution 時間 :

$J = \{j_1, j_2, \dots, j_k\}$ & **sort** these jobs with their **deadline**,

s.t. $d_{i1} \leq d_{i2} \leq \dots \leq d_{ik}$.

Then, the jobs in J can be finished before deadlines iff J is **feasible**

The new **inserted job** $d_{ik} \geq k$ (**inserted job** 放在 k 的位置) (or $d_{ik} > k-1$), then J is **feasible**. (\because each job's processing time is one unit.) The **time complexity** for **checking** the feasible solution J is ck when a job j_k is inserted (i.e., $T(n) = cn$ for checking the feasible solution when insert a job).

(因為 job j_k 加入在 **deadline** 的位置 x , 會將 x 之後的 **jobs** 延後一單位時間, 這些 **jobs** 都將再檢查一次, 是否在 **deadline** 前可完成, 即 $d_{iy} \geq y, \forall y > x$).

The total **time complexity** = **sorting** + **insert & checking**

$$\begin{aligned} &= O(n \log n) + \underline{c \cdot 1 + c \cdot 2 + c \cdot 3 + \dots + c \cdot n} \\ &\quad \text{(worst case)} \\ &= O(n \log n) + O(n^2) \\ &= O(n^2) \end{aligned}$$

EXERCISE: $n=10$,

$P = (100, 10, 15, 27, 36, 58, 62, 43, 52, 65)$

$D = (3, 2, 5, 5, 4, 2, 2, 4, 3, 4)$

同前: **Coding & 說明過程 & 結果**