Unit 6
Greedy Algorithms

T.H. Cormen et al., "Introduction to Algorithms", 3rd ed., Chapter 16.

Greedy Methods (描述1)

- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- A ***greedy method*** always makes the choice that **looks best** at the moment.
- Greedy methods **do not always yield optimal solutions**, but for several problems they do.

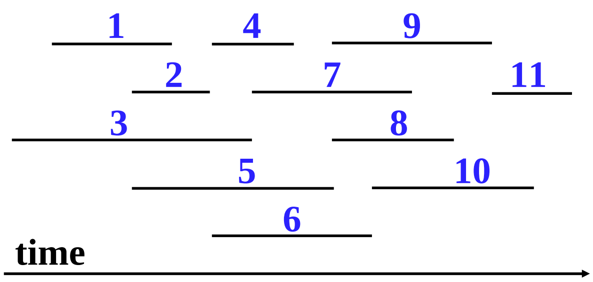
Greedy Methods (描述2)

- Greedy algorithms often **lead to very efficient and simple algorithms**; however they are **harder to prove** the correctness (compared to DP algorithms).
- Many heuristic algorithms apply greedy methods.

An Activity-Selection Problem (定義)

Suppose we have a set of ***n*** proposed **activities** that wish **to use a resource** which can be **used by only one activity at a time**. The problem is to select a max-size subset of activities that can be allowed to use the resource.

Assume activity $\boldsymbol{i}$, if selected, takes place during the time interval $[s_i, f_i)$ denoted as I_i



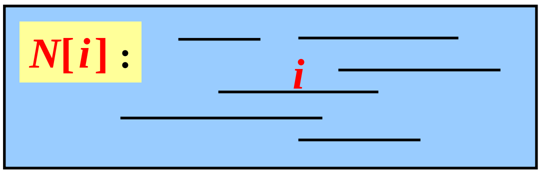
An Activity-Selection Problem (設計 1)

- Let $P(A)$ denote the problem with A as the given set of proposed activities and S denote an optimal solution of $P(A)$. For any activity i in A, we have

1. $i \notin S \Rightarrow S$ is an optimal solution of $P(A \backslash \{i\})$.

2. $i \in S \Rightarrow S \backslash \{i\}$ is an optimal solution of $P(A \backslash N[i])$ but not necessary an optima solution of $P(A \backslash \{i\})$.

$$N[i]=\{j \in A: I_j \cap I_i \neq \varnothing\}$$



An Activity-Selection Problem (設計 2)

- What kind of activity i in A will be contained in an optimal solution of $P(A)$: an activity with

1. minimum $f_i - s_i$ or
2. minimum $|N[i]|$ or
3. minimum f_i or
4. minimum s_i.

Answer : _____.

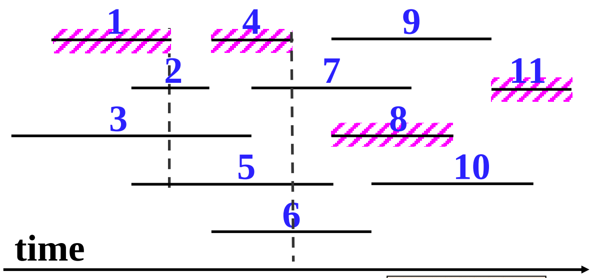
Proof : Let f_1 = min $\{ f_i \}$ and S be an optimal solution of $P(A)$. If $1 \notin S$ then there is one and only one activity in S, say j, such that $I_j \cap I_1 \neq \varnothing$.
Then, $S \backslash \{j\} \cup \{1\}$ is also an optimal solution.

An Activity-Selection Problem (p碼1+例子)

Input:

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	9	13
11	12	14

```
Greedy-ASP(A)
{
    if A == ∅ return ∅
    i = arg min {f_k | k ∈ A }
    return {i} ∪ Greedy-ASP(A\ N[i])
}
```



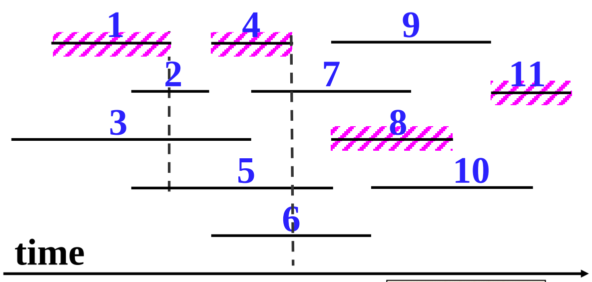
$T(n)$ = ?

An Activity-Selection Problem (p碼2)

Input:

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	9	13
11	12	14

```
Greedy-ASP(s, f, n)
{                          /* f[1] ≤ f[2] ≤ … ≤ f[n]*/
    Ans = {1};
    for(i=2, j=1; i≤n; i++)
        if(s[i]≥f[j]) {Ans = Ans ∪ {i}; j = i; }
}
```



$T(n)$ = ?

Elements of the Greedy Strategy

- Optimal substructure (a problem exhibits ***optimal substructure*** if an optimal solution to the problem contains within it optimal solutions to subproblems)
- Greedy-choice property
- Priority queue or sorting

Knapsack Problem (Greedy vs. DP)

Given ***n*** items: weights: w_1 w_2 … w_n
values: v_1 v_2 … v_n
a knapsack of capacity ***K***

Find the most valuable load of the items that fit into the knapsack.

Example:

item	*weight*	*value*	*Knapsack capacity K=16*
1	2	$20	
2	5	$30	
3	10	$50	
4	5	$10	

0-1 and Fractional Knapsack Problem

- Constraints of 2 variants of the knapsack problem:
  - ***0-1 knapsack problem***: each item must either be taken or left behind.
  - ***Fractional knapsack problem***: the thief can take fractions of items.
- The greedy strategy of taking as mush as possible of the item with greatest v_i / w_i only works for the fractional knapsack problem.

0-1 Knapsack Problem (設計)

- $P[i, k]$ = the value of the most valuable load of the subproblem: consider only the first i items and a knapsack of size k, for any i, k $0 \le i \le n$, $k \le K$.
- The optimal load either include i–th item or not. Hence we have:

$$P[i, k] = \max \{P[i-1, k], P[i-1, k-w_i] + v_i\}$$
$$P[0, k] = 0, k > 0; P[i, 0] = 0, i \ge 0$$

Assume $k \ge w_i$

0-1 Knapsack Problem (例+分析)

$$P[i, k] = \max \{P[i-1, k], P[i-1, k-w_i] + v_i\}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$20/2	0	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
$30/5	0	20	20	20	30	30	50	50	50	50	50	50	50	50	50	50
$50/10	0	20	20	20	30	30	50	50	50	50	50	70	70	70	80	80
$10/5	0	20	20	20	30	30	50	50	50	50	50	70	70	70	80	80

- Time: $O(nK)$
- It is possible that $K > 2^n$.
- A pseudo-polynomial time algorithm.

Greedy 解 = ?
最佳解 = ?

0-1 Knapsack Problem (另一實做)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$20/2	0	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
$30/5	0	20	20	20	30	30	50	50	50	50	50	50	50	50	50	50
$50/10	0	20	20	20	30	30	50	50	50	50	50	70	70	70	80	80
$10/5	0	20	20	20	30	30	50	50	50	50	50	70	70	70	80	80

■ → 2, $20 —item2→ ■ → 2, $20 → 5, $30 → 7, $50 —item3→

■ → 2, $20 → 5, $30 → 7, $50 → 12, $70 → 15, $80

- $\therefore$ $T(n) = O(n\min(K, 2^n))$,
  $S(n) = O(\min(K, 2^n))$.

The idea can be used for other DP algorithms, such as LCS …etc.

Huffman Codes

- A very effective technique for compressing data.
- Consider the problem of designing a binary character code.
- Fixed length code vs. variable-length code, e.g.:

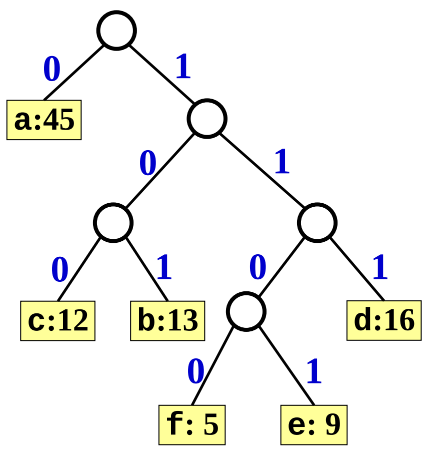
Alphabet:	a	b	c	d	e	f
Frequency in a file	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

file length 1 = 300; file length 2 = 224

Compression ratio = (300–224)/300·100% ≈ 25%

Prefix Codes & Coding Trees

- We consider only codes in which no codeword is also a ***prefix*** of some other codeword.
- The assumption is crucial for decoding variable-length code (using a binary tree). E.g. if we use “01” for ‘a’ and “011” for ‘b’, then …



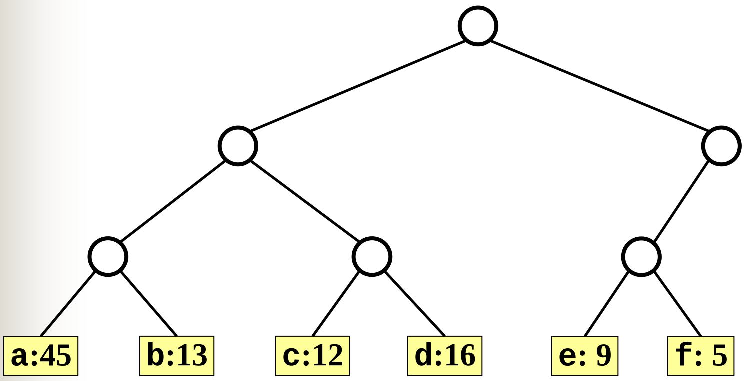
Optimal Coding Trees

- For a alphabet C, and its corresponding coding tree T, let $f(c)$ denote the frequency of $c \in C$ in a file, and let $d_T(c)$ denote the depth of c's leaf in T. ($d_T(c)$ is also the length of the codeword for c.)
- The size required to encode the file is thus:

  $B(T) = \Sigma_{c \in C}\ f(c) d_T(c)$
- We want to find a coding tree with minimum $B(T)$.

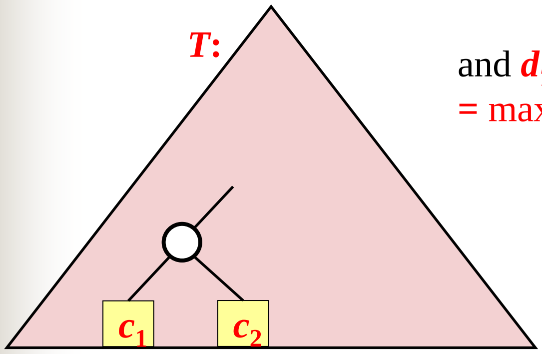
Observation 1

- Any optimal coding tree for C, $|C| > 1$, must be a <u>*full binary tree*</u>, in which every nonleaf node has two children. E.g.: for the fixed-length code:



Observation 2 (greedy-choice)

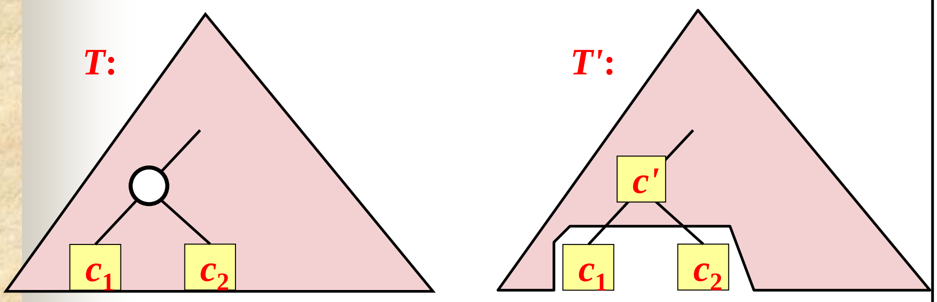
- Assume $C = \{ c_1, c_2, \ldots , c_n\}$, and $f(c_1) \leq f(c_2) \leq \ldots \leq f(c_n)$. Then there exists an optimal coding tree T such that :



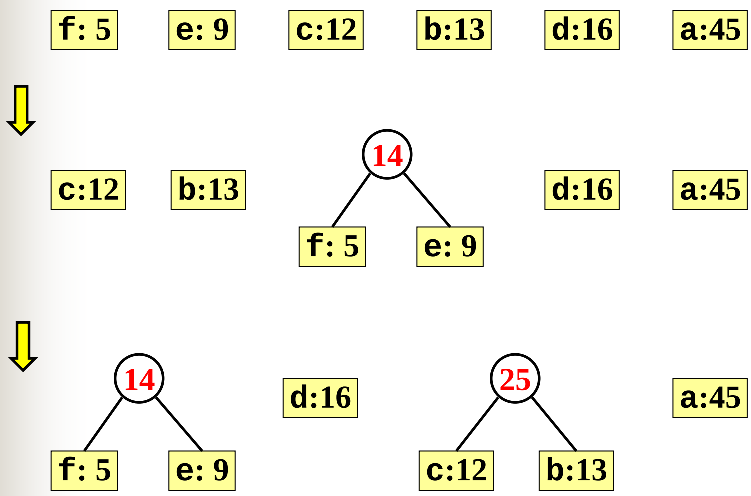
and $d_T(c_1) = d_T(c_2) = \max_{c \in C} d_T(c)$

Observation 3 (optimal substructure)

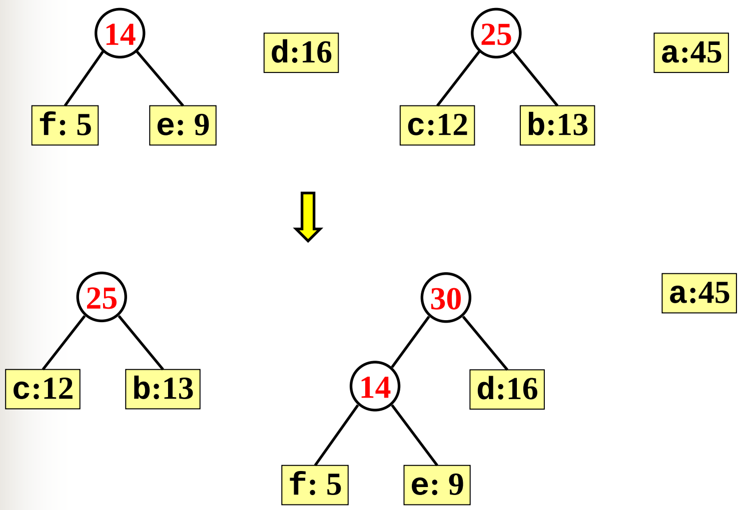
- If T is an optimal coding tree for C, then T' is an optimal coding tree for $C \setminus \{c_1, c_2\} \cup \{c'\}$ with $f(c') = f(c_1) + f(c_2)$.



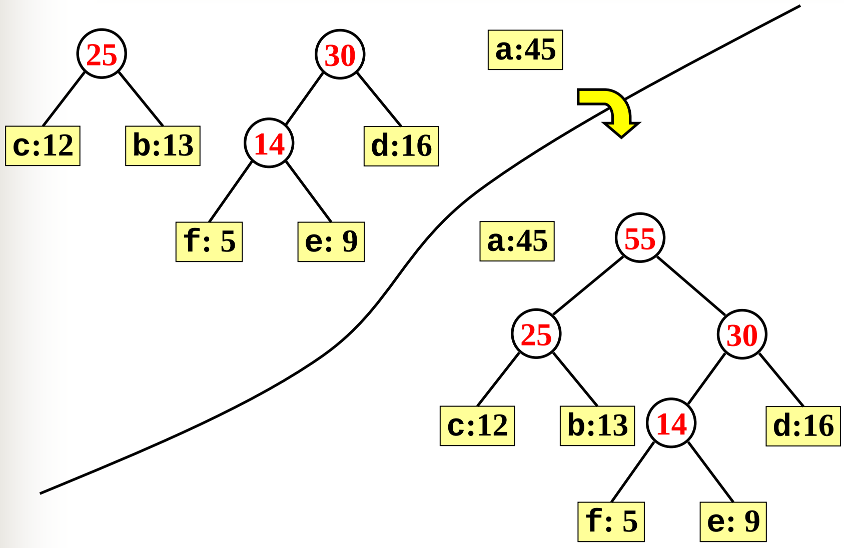
Huffman's Algorithm (例)



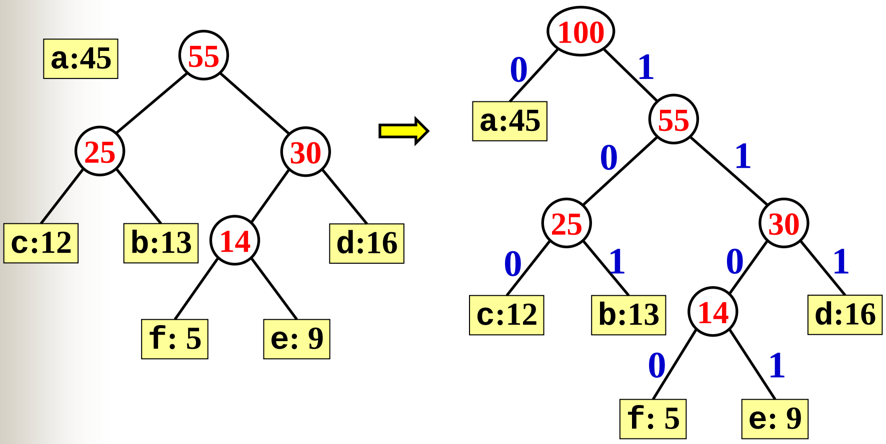
Huffman's Algorithm (例-續1)



Huffman's Algorithm (例-續2)



Huffman's Algorithm (例-續3)



Huffman's Algorithm (pseudo-code)

```
Huffman(C)
  Q ← C        // Q :priority queue
  while(|Q| > 1)
      z ← Allocate-Node( )
      x ← left[z] ← Extract-Min(Q)
      y ← right[z] ← Extract-Min(Q)
      f [z] ← f [x] + f [y]
      insert(Q, z)
  return Extract-Min(Q)
```

Time efficiency:_____.

A Task-Scheduling Problem

Schedule ***n*** unit-time tasks for a single processor with:

deadlines: d_1 $d_2 \ldots$ d_n

profits: p_1 $p_2 \ldots$ p_n (or penalties)

Find a schedule for these tasks that maximize (or minimize) the total profit (or penalty).

- A set ***S*** of tasks is ***feasible*** (***independent***) if there is a schedule for these tasks such that no tasks are late.
- The problem is equivalent to find a feasible task (sub-)set with maximum profit sum.

A Task-Scheduling Problem (例)

task	Deadline	Profit
1	2	30
2	1	35
3	2	25
4	1	40

Schedule	*Total Profits*
[1, 3]	30 + 25 = 55
[2, 1]	35 + 30 = 65
[2, 3]	35 + 25 = 60
[3, 1]	25 + 30 = 55
[4, 1]	40 + 30 = 70
[4, 3]	40 + 25 = 65

A Greedy-Choice Property

- What kind of task i will be contained in an optimal schedule : a task with

1. minimum d_i ,
2. maximum d_i ,
3. minimum p_i ,
4. maximum p_i .

Answer : _____.

Proof : Assume task 1 is a task with maximum profit, and S is an optimal schedule. If $1 \notin S$ then we can replace any task in S that is scheduled before or at d_1 with task 1 and obtain a schedule without decreasing the total profit.

A Greedy Algorithm

```
Sort the tasks in nonincreasing order by profit;
S = ∅;
while(there are tasks unprocessed) {
  select next task;
  if(S is feasible with this task added)
    add it to S;
}
```

☛ How to check that S is feasible?

Feasibility Testing Method 1

Lemma 16.12: The 3 statements are equivalent:

1. S is feasible
2. $|\{i \in S : d_i \le t\}| \le t$ for $t = 1, 2, \ldots, \max_i d_i$.
3. If the tasks in S are scheduled in order of non-decreasing deadlines, then no task is late.

Example : tasks	1	2	3	4	5	6	7
Deadlines	3	1	1	3	1	3	2

Assume that profits $p_1 \ge p_2 \ge \ldots \ge p_7$

A naive implementation of the lemma needs $O(n^2)$ time.

Feasibility Testing Method 2 (see Problem 16-4)

tasks :	1	2	3	4	5	6	7	8
deadlines :	7	7	7	10	11	9	10	11

			T_8	T_3	T_2	T_1	T_7	T_6	T_4	T_5			
1	2	3	4	5	6	7	8	9	10	11	12	13	14

The implementation based on union-find operations (or disjoint set unions) needs $O(n\alpha(n))$ time (not including sorting time.)

$\alpha(n)$ = 4 even for $n = 2^{2048}$ (p.574).