Greedy Algorithms

Greedy Methods

- 解最佳化問題的演算法, 其解題過程可看成是由一連串的決策步驟所組成, 而每一步驟都有一組選擇要選定.
- 一個 ***greedy method*** 在每一決策步驟總是選定那目前看來最好 的選擇.
- Greedy methods 並不保證總是得到最佳解, 但在有些問題卻可以得到最佳解.

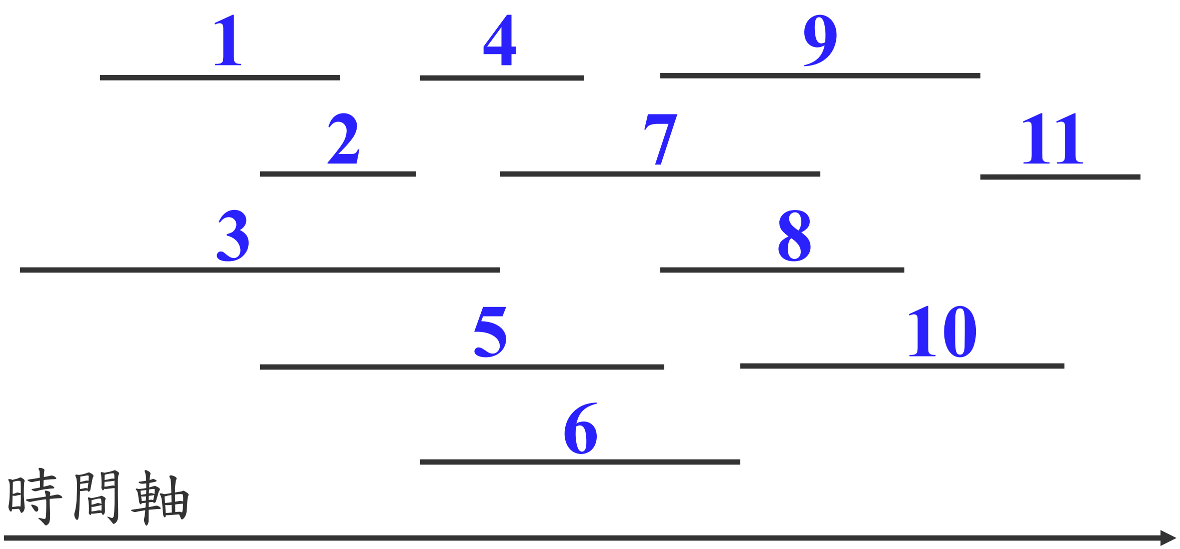
Greedy Methods

- Greedy 演算法經常是非常有效率且簡單的演算; 但較難證明其正確性 (與 DP 演算法比較).
- 很多 heuristic algorithms 都採用 greedy methods 的策略.

活動選擇問題 (定義)

假設有 n 個 活動 提出申請要使用一個場地, 而這場地在同一時間點時最多只能讓一個活動使用. 問題是：從這 n 個活動選一組數量最多, 且可以在這場地舉辦的活動集.

假設活動 i, 其提出申請使用場地的時段為一半關半開的區間 $[s_i, f_i)$, 並以符號 I_i 代表.

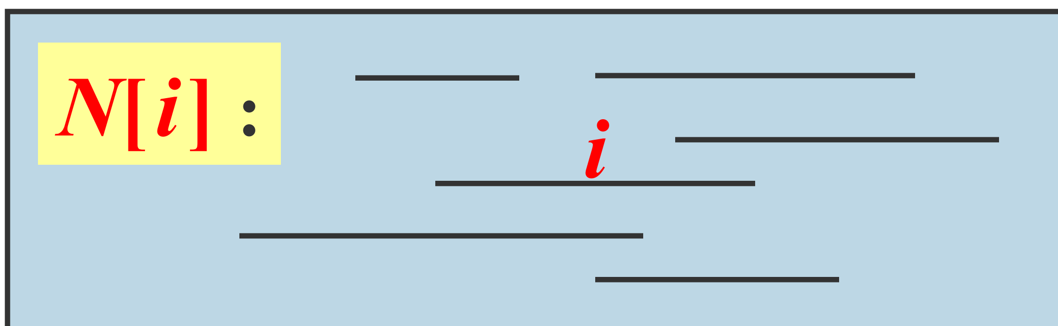


活動選擇問題 (設計 1)

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

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



活動選擇問題 (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 : _____.

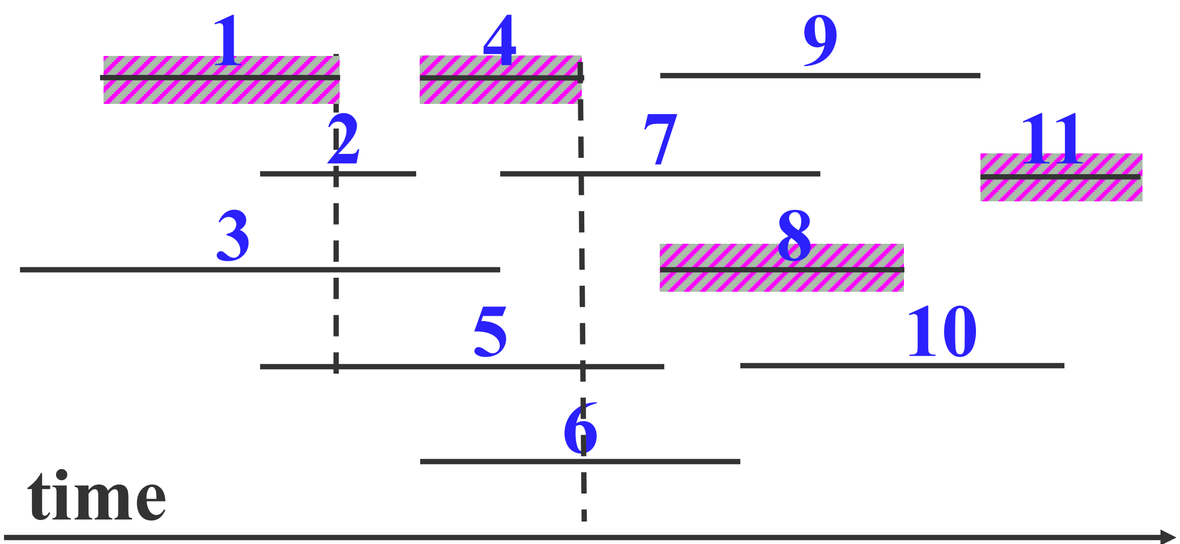
Proof : Let $f_1 = \min\{f_i\}$ and S be an optimal solution of $P(A)$. If $1 \notin S$ then there is one and only one activity in S, say j, such that $I_j \cap I_1 \neq \varnothing$. Then $S \setminus \{j\} \cup \{1\}$ is also an optimal solution.

活動選擇問題 (程式+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(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; }
}
```



Greedy 演算法的要素

- 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

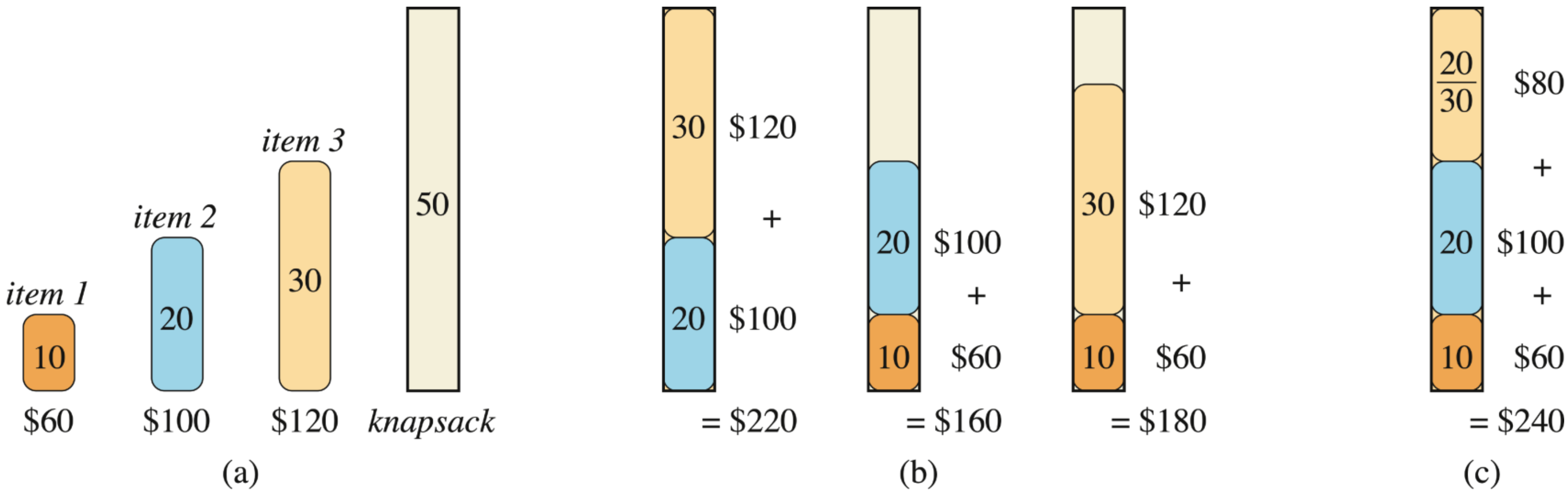
Knapsack Problem (Greedy vs. DP)

Given n items: **weights:** $w_1 \quad w_2 \ldots \quad w_n$
values: $v_1 \quad v_2 \ldots \quad v_n$
a knapsack of capacity W

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

Example:

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


item 1
10
$60
item 2
20
$100
item 3
30
$120
50
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)

Knapsack Problem

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

T[i, j]: the optimal solution using item 1,...,i with weight at most j.

If $w_i > j$: T[i, j] = T[i-1, j];

else: T[i, j] = max{ T[i-1, j], v_i + T[i-1, j - w_i]}.

How good is this method?

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.

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.:**

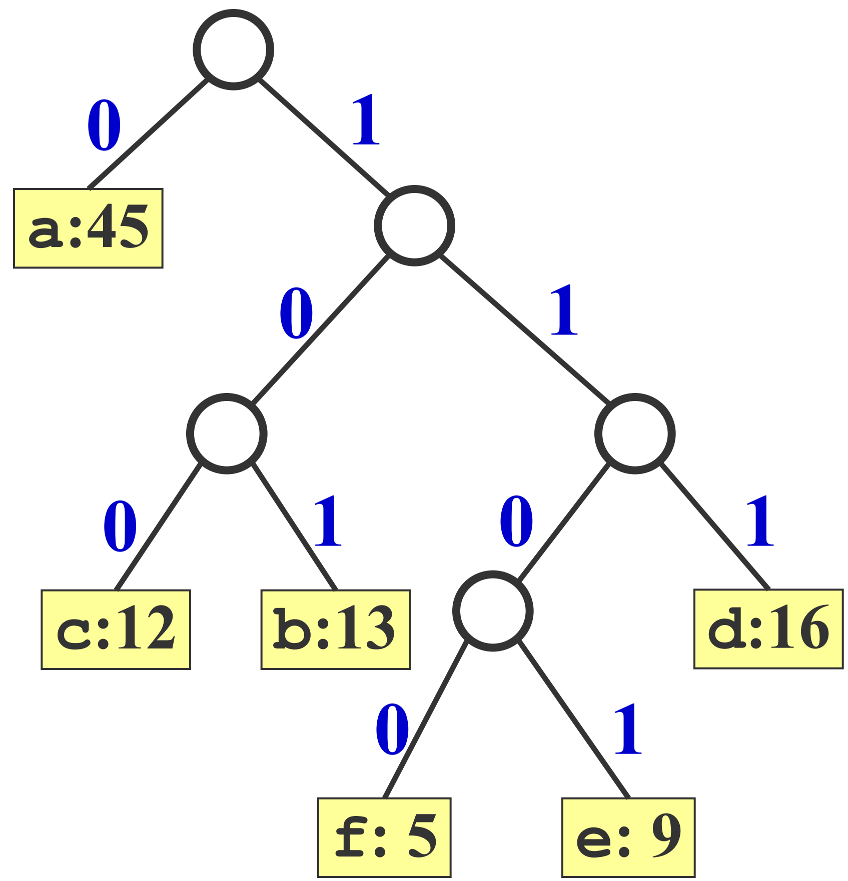
Alphabet:	`a`	`b`	`c`	`d`	`e`	`f`
Frequency in a file	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

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.:



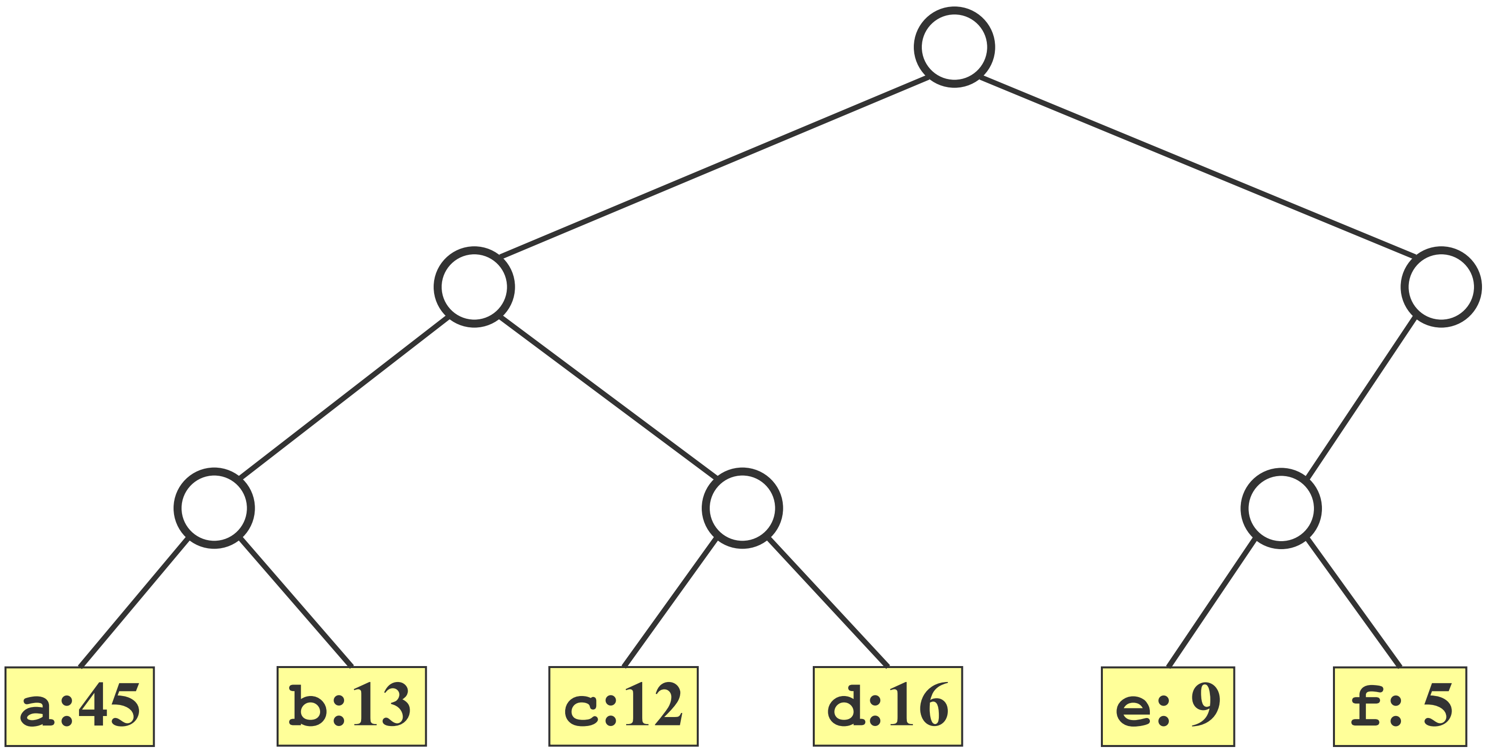
Optimal Coding Trees

- For a alphabet C, and its corresponding coding tree T, let $c.freq$ 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:

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

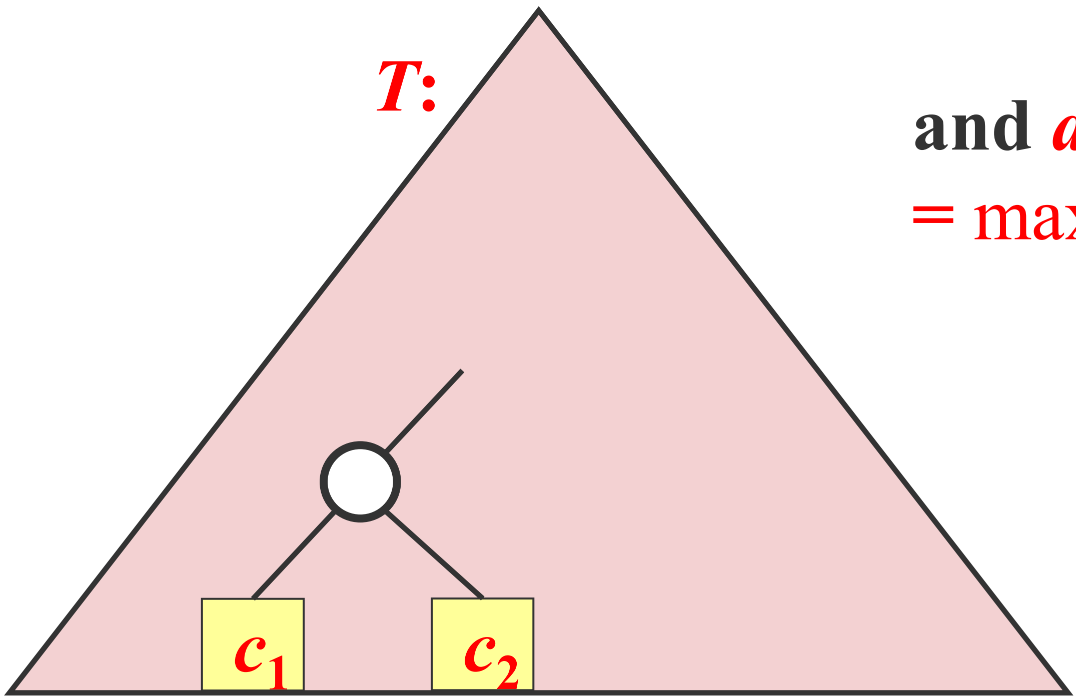
Observation 1

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

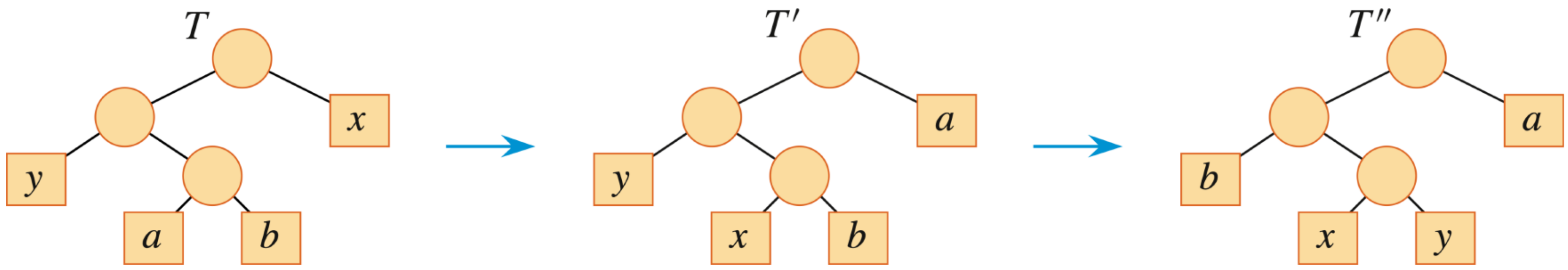


Observation 2

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



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

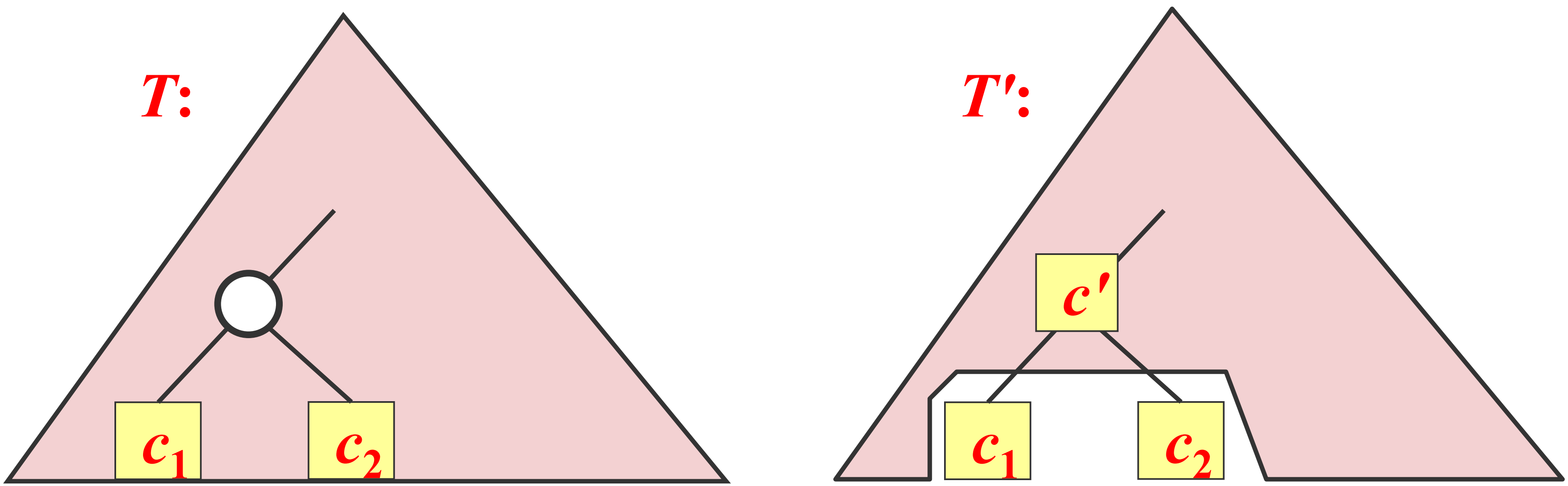


$B(T) := \Sigma_{c \in C}\ c.freq\ d_T(c)$

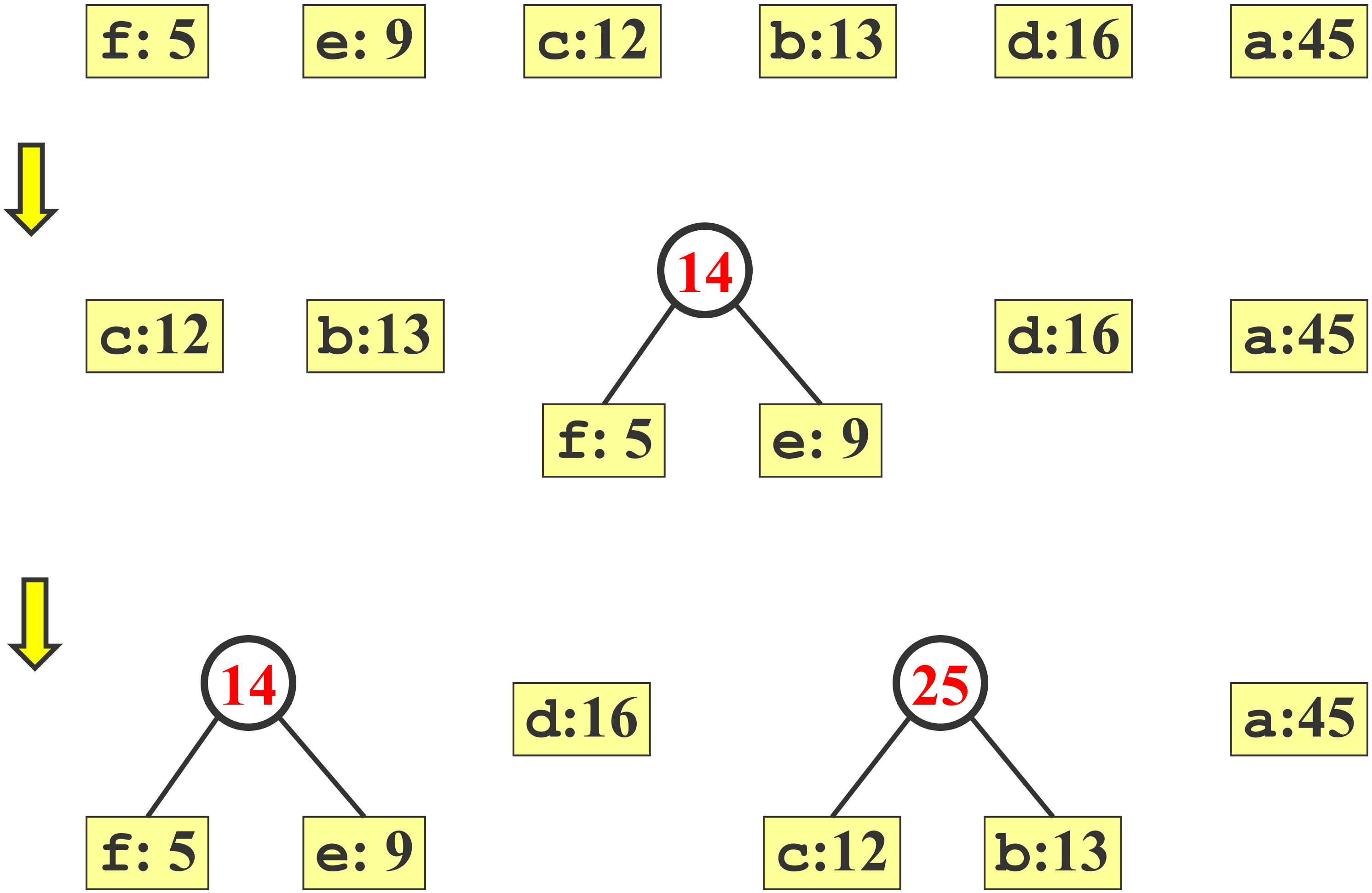
$$
\begin{aligned}
B(T) - B(T') \\
&= \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x)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

Observation 3

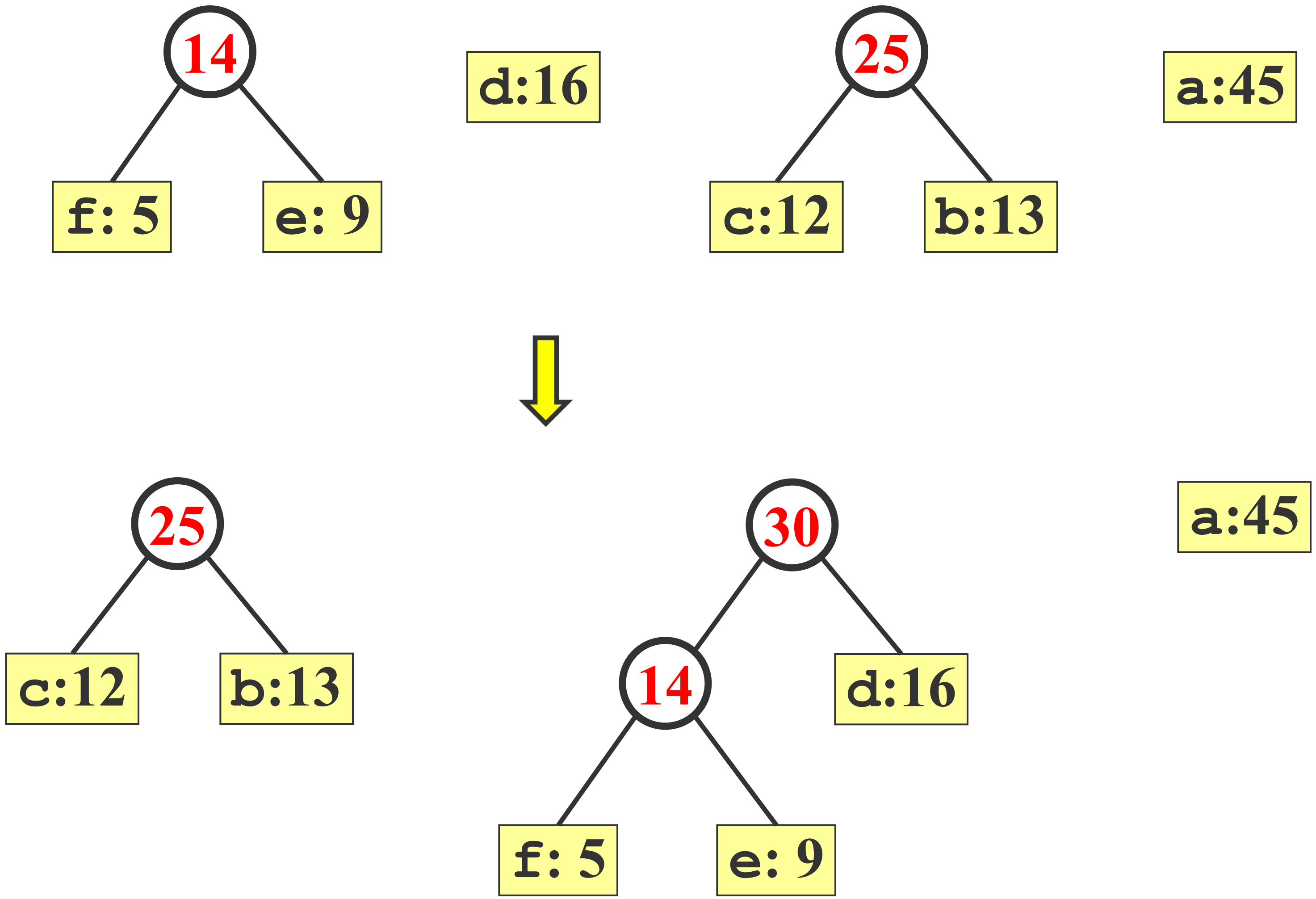
- If is T an optimal coding tree for C , then T' is an optimal coding tree for $C \setminus \{c_1, c_2\} \cup \{c'\}$ with $c'.freq = c_1.freq + c_2.freq$.



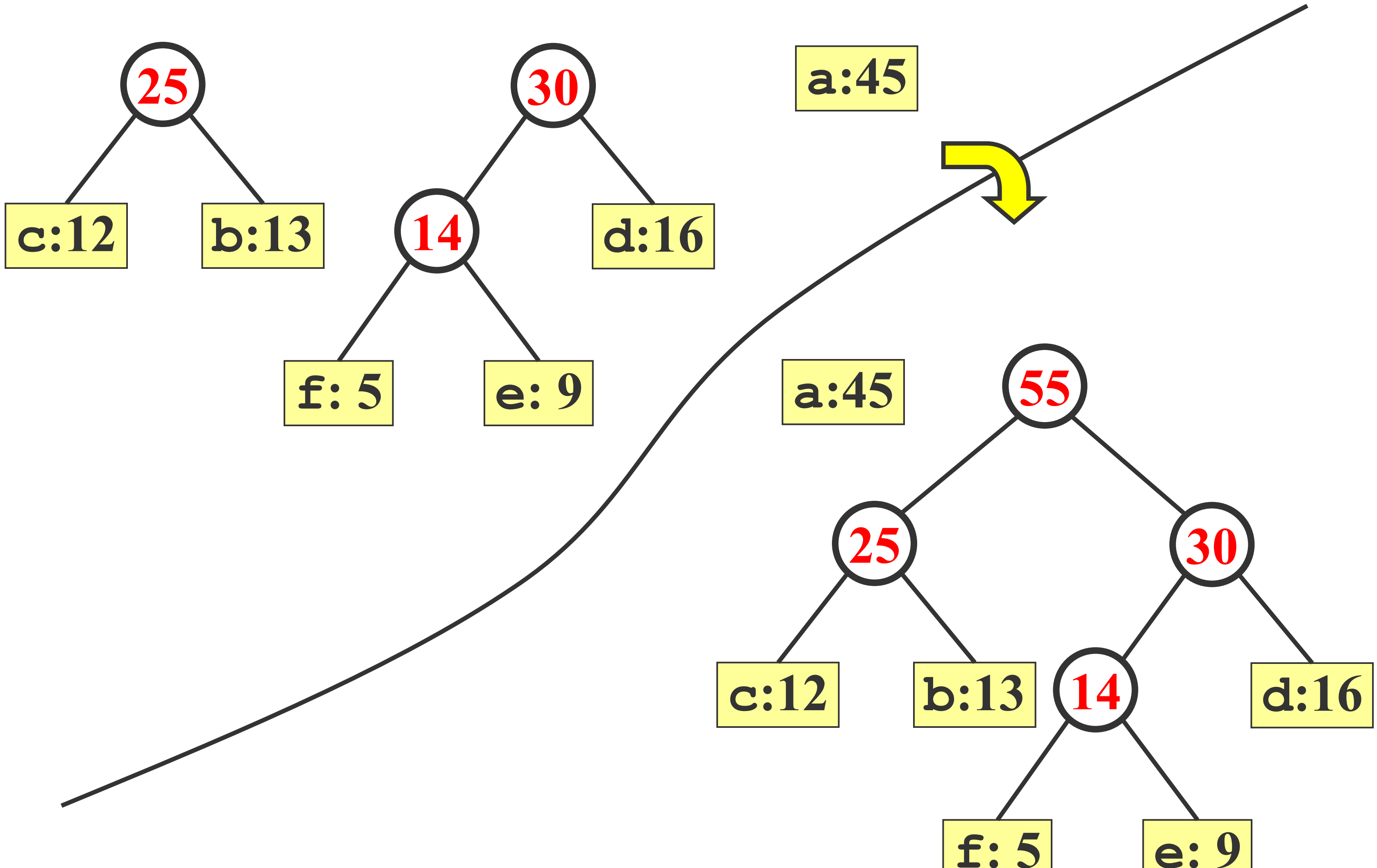
Huffman's Algorithm (例)



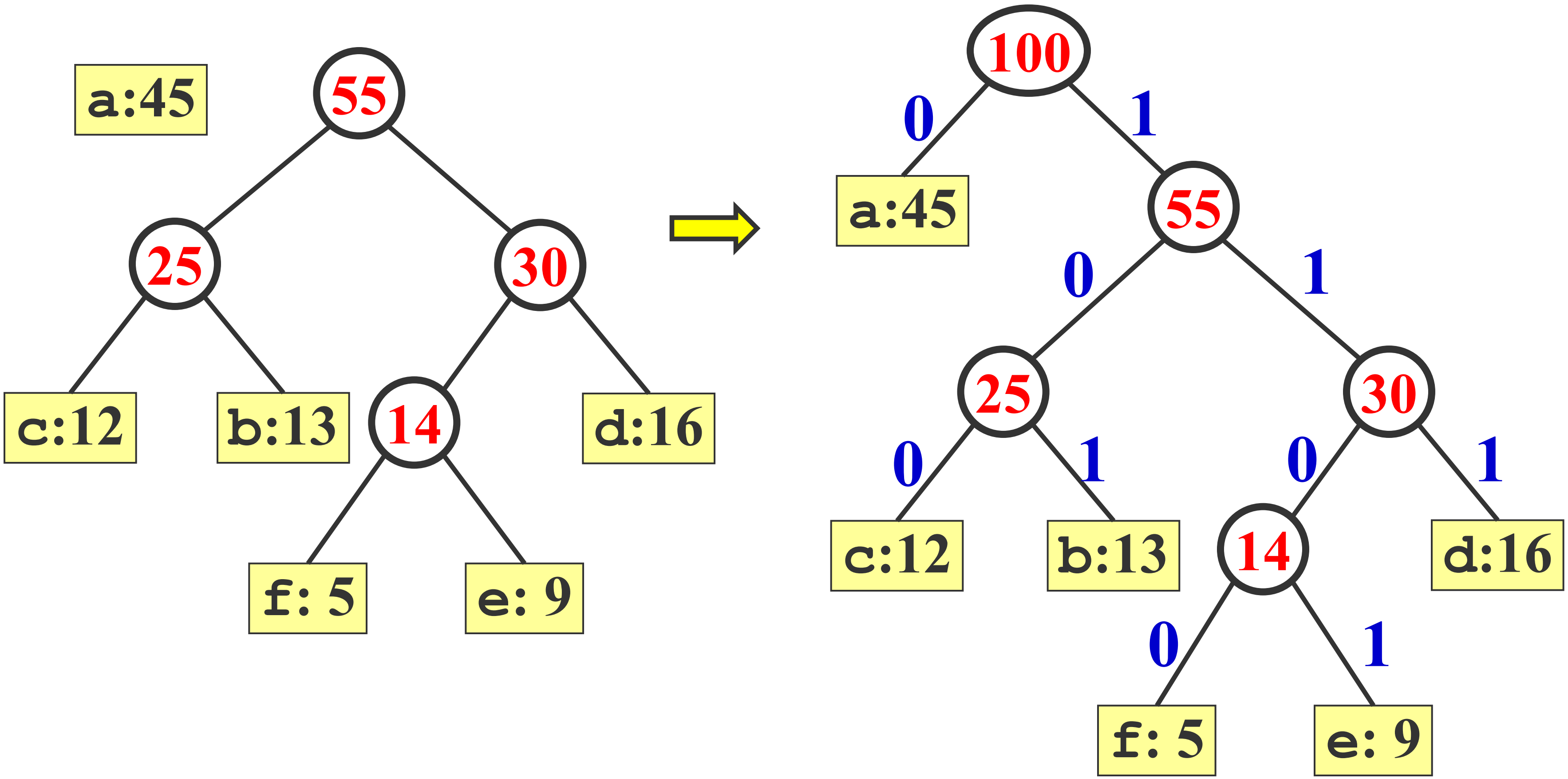
Huffman's Algorithm (例-續1)



Huffman's Algorithm (例-續2)



Huffman's Algorithm (例-續3)



Huffman's Algorithm

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    x = EXTRACT-MIN(Q)
    y = EXTRACT-MIN(Q)
    z.left = x
    z.right = y
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // the root of the tree is the only node left
```

Offline caching

- Computer systems can decrease the time to access data by storing a subset of the main memory in the ***cache***: a small but faster memory.
- ***Cache blocks*** typically comprise 32, 64, or 128 bytes.
- Think of main memory as a cache for disk-resident data in a virtual-memory system. The blocks are called ***pages***, and 4096 bytes is a typical size.
- As a computer program executes, it makes a sequence of memory requests. Say that there are n memory requests, to data in blocks $b_1, b_2, \dots, b_n$ in order.

Offline caching

- Example: a program that accesses four distinct blocks p, q, r, s might make a sequence of requests to blocks $s, q, s, q, q, s, p, p, r, s, s, q, p, r, q$.
- Assume the cache can hold fixed number of cache blocks starting with empty.
- Each request causes at most one block to enter the cache and at most one block to be evicted from the cache.
- Upon a request for block b_i , any one of three scenarios may occur:
  1. Block b_i is already in the cache, cache hit
  2. Block b_i is not in the cache at that time, but not full
  3. Block b_i is not in the cache at that time, but full

Offline caching

- Goal: minimize the number of cache misses or, maximize the number of cache hits, over the n requests.
- A greedy strategy furthest-in-future: chooses to evict the block in the cache whose next access in the request sequence comes furthest in the future.
- C: cache configuration, subset of the set of blocks with $|C| \leq k$.
- Define the subproblem (C, i) as processing requests for blocks $b_i, b_{i+1}, \dots, b_n$ with cache configuration C at the time that the request for block b_i occurs.

Offline caching

Optimal substructure of offline caching:

- Let S be an optimal solution to subproblem (C, i) and C' be the contents of the cache after processing the request for block b_i in solution S .
- Let S' be the subsolution of S for the resulting subproblem $(C', i + 1)$.
- If $b_i \in C$, then $C = C'$; else $C \neq C'$.
- Claim: S' is an optimal solution to subproblem $(C', i + 1)$

Offline caching

Optimal substructure of offline caching:

- Let $R_{C,i}$ be the set of all cache configurations that can immediately follow configuration C after processing a request for block b_i.
- **if** cache hit: $R_{C,i} = \{C\}$;
  else if cache miss and $|C| < k$: $R_{C,i} = \{C \cup \{b_i\}\}$
  else: $R_{C,i} = \{(C - \{\mathrm{x}\}) \cup \{b_i\}, x \in C\}$
- Let $miss(C, i)$ denote the minimum number of cache misses in a solution for subproblem (C, i).

$$miss(C,i) = \begin{cases} 0 & \text{if } i = n \text{ and } b_n \in C , \\ 1 & \text{if } i = n \text{ and } b_n \notin C , \\ miss(C, i+1) & \text{if } i < n \text{ and } b_i \in C , \\ 1 + \min\{miss(C', i+1) : C' \in R_{C,i}\} & \text{if } i < n \text{ and } b_i \notin C . \end{cases}$$

Offline caching

Greedy-choice property:

Theorem: Consider a subproblem (C, i) when $|C| = k$, so that it is full, and a cache miss occurs. When block b_i is requested, let $z = b_m$ be the block in C whose next access is furthest in the future.
Then evicting block z upon a request for block b_i is included in some optimal solution for the subproblem (C, i).