Chapter 2

The Complexity of Algorithms and the Lower Bounds of Problems

Outlines

2-1 The Time-Complexity of an Algorithm

2-2 The Best, Average and Worst Case Analysis of Algorithms

2-3 The Lower Bound of a Problem

2-4 The Worst Case Lower Bound of Sorting

2-5 Heapsort - A Sorting Algorithm which Is Optimal in Worst Cases

2-6 The Average Case Lower Bound of Sorting

2-7 The Improving of a Lower Bound through Oracles

2-8 The Finding of Lower Bound by Problem Transformation

1.1 Introduction

- **How do we measure the goodness of an algorithm?**
- **How do we measure the difficulty of a problem?**
- **How do we know that an algorithm is optimal for a problem?**
- **How can we know that there does not exist any other better algorithm to solve the same problem?**

Example 2-1 Straight insertion sort

input: 7,5,1,4,3

7,5,1,4,3

5,7,1,4,3

1,5,7,4,3

1,4,5,7,3

1,3,4,5,7

Algorithm 2.1 Straight Insertion Sort

Input: $x_1, x_2, \ldots, x_n$

Output: The sorted sequence of $x_1, x_2, \ldots, x_n$

```
For j := 2 to n do
Begin
    i := j-1
    x := x_j
    While x<x_i and i > 0 do
    Begin
        x_{i+1} := x_i
        i := i-1
    End
    x_{i+1} := x
End
```

Always do

input: 7,5,1,4,3

7,$\underline{5}$,1,4,3

5,7,$\underline{1}$,4,3

1,5,7,$\underline{4}$,3

1,4,5,7,$\underline{3}$

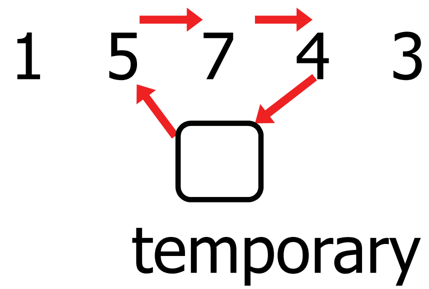
1,3,4,5,7

Inversion table

- $(a_1, a_2, \ldots, a_n)$: a permutation of $\{1, 2, \ldots, n\}$
- $(d_1, d_2, \ldots, d_n)$: the inversion table of $(a_1, a_2, \ldots, a_n)$
- d_i: the number of elements to the left of i that are greater than i
- e.g. permutation (7 5 1 4 3 2 6)
  inversion table 2 4 3 2 1 1 0
- e.g. permutation (7 6 5 4 3 2 1)
  inversion table 6 5 4 3 2 1 0
- d_i: the number of movements executed for x_i in the inner do loop.

Analysis of # of movements

- ***M***: # of data movements in straight insertion sort



- e.g. $d_4=2$
- $$X = \sum_{i=2}^{n} (2 + d_i) = 2(n-1) + \sum_{i=2}^{n} (d_i)$$

Analysis by inversion table

- best case: already sorted

  $d_i = 0$ for $1 \le i \le n$

  $\Rightarrow X = 2(n - 1) = O(n)$

- worst case: reversely sorted

  $d_1 = 0$

  $d_2 = 1$

  $\vdots$

  $d_i = n - i$

  $d_n = n-1$

$$X = \sum_{i=2}^{n} (2 + d_i) = 2(n-1) + \frac{n(n-1)}{2} = O(n^2)$$

- **average case**:

  x_i is being inserted into the sorted sequence

  $x_1\ x_2\ \ldots\ x_{i-1}$
- the probability that x_i is the largest: 1/i
  - takes 2 data movements ($2+d_i=2$, $d_i=0$)
- the probability that x_i is the second largest : 1/i
  - takes 3 data movements

  1 4 7 5
- # of movements for inserting x_i:

$$2 + d_i = \frac{2}{i} + \frac{3}{i} + \cdots + \frac{i+1}{i} = \sum_{j=1}^{i} \frac{j+1}{i} = \frac{i+3}{2}$$

$$X = \sum_{i=2}^{n} \frac{i+3}{2} = \frac{1}{2}\left(\sum_{i=2}^{n} i + \sum_{i=2}^{n} 3 \right) = \frac{(n+8)(n-1)}{4} = O(n^2)$$

Formula

$$\sum_{k=1}^{n} k = \frac{1}{2}(n^2 + n) = \frac{1}{2}n(n+1)$$

$$\sum_{k=1}^{n} k^2 = \frac{1}{6}(2n^3 + 3n^2 + n) = \frac{1}{6}n(n+1)(2n+1)$$

$$\sum_{k=1}^{n} k^3 = \frac{1}{4}(n^4 + 2n^3 + n^2) = \frac{1}{4}n^2(n+1)^2$$

$$\sum_{k=1}^{n} k^4 = \frac{1}{30}(6n^5 + 15n^4 + 10n^3 - n) = \frac{1}{30}n(n+1)(2n+1)(3n^2 + 3n - 1)$$

$$\sum_{j=1}^{n} \frac{1}{k(k+1)} = \frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{n(n+1)} = \frac{n}{(n+1)}$$

Analysis of # of exchanges

- Method 1 (straightforward)
- x_i is being inserted into the sorted sequence

  $x_1\ x_2\ \ldots\ x_{i-1}$
- If x_j is the kth ($1 \le k \le i$) largest, it takes ($k-1$) exchanges.
- e.g. 1 5 7↔4

  1 5↔4 7

  1 4 5 7
- # of exchanges required for x_i to be inserted:

$$\frac{0}{i} + \frac{1}{i} + \cdots + \frac{i-1}{i} = \frac{i-1}{2}$$

- # of exchanges for sorting:

$$\sum_{i=2}^{n} \frac{i-1}{2}$$

$$= \sum_{i=2}^{n} \frac{i}{2} - \sum_{i=2}^{n} \frac{1}{2}$$

$$= \frac{1}{2} \cdot \frac{(n-1)(n+2)}{2} - \frac{n-1}{2}$$

$$= \frac{n(n-1)}{4}$$

Example 2-2 Binary search

- sorted sequence : (search 9)

	1	4	5	7	9	10	12	15
step 1				↑				
step 2						↑		
step 3					↑			

- best case: 1 step = O(1)
- worst case: ($\lfloor \log_2 n \rfloor$+1) steps = O(log n)
- average case: O(log n) steps

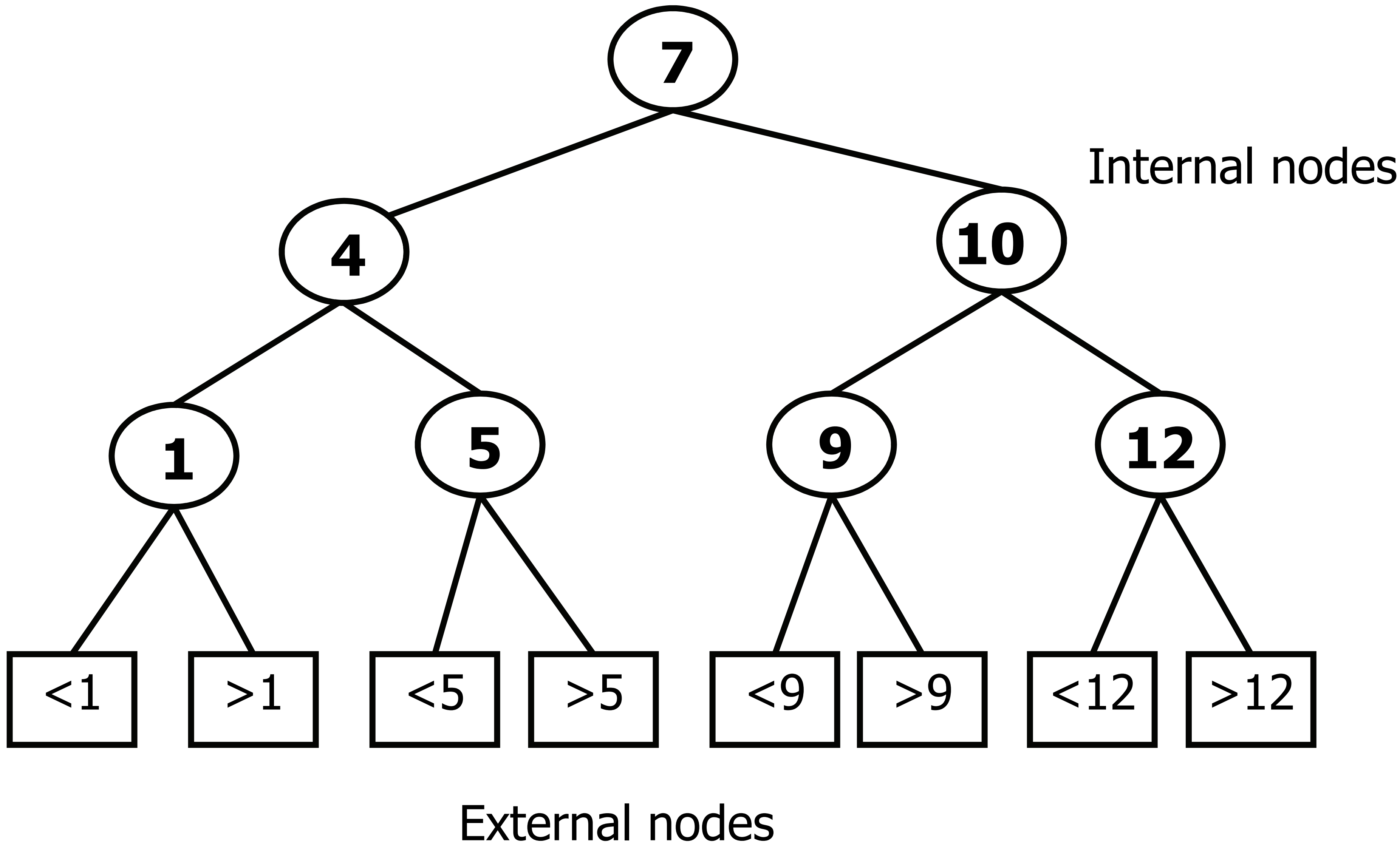
Binary Search Algorithm

Input： $a_1, a_2, \ldots, a_n$, $n > 0$, with $a_1 \le a_2 \le \ldots \le a_n$, and x

Output： j if $a_j = X$ and 0 if no j exists such that $a_j = X$.

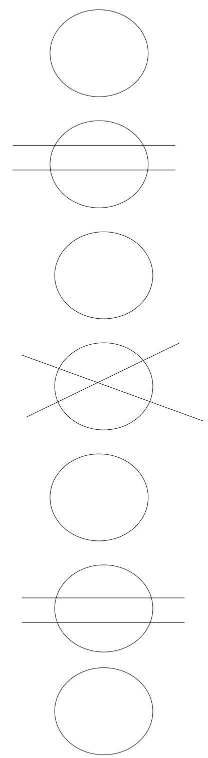
```
i := 1
m := n
while (i ≤ m) do
      begin  j := ⌊(i+m)/2⌋
             if (x = a_j) then output j & stop
             if (x < a_j)  then m := j–1
             else i := j+1
      end
j := 0
output j
```

Binary Searching Tree



The binary Search (Analysis- Average case) ＊找得到的情況：

計有 1	個情況，是找了	1	次即得
2		2	
4		3	
:		:	
$2\lfloor \log n \rfloor$		$\lfloor \log n \rfloor + 1$	



The binary Search (Analysis- Average case) ＊ 找不到的情況：

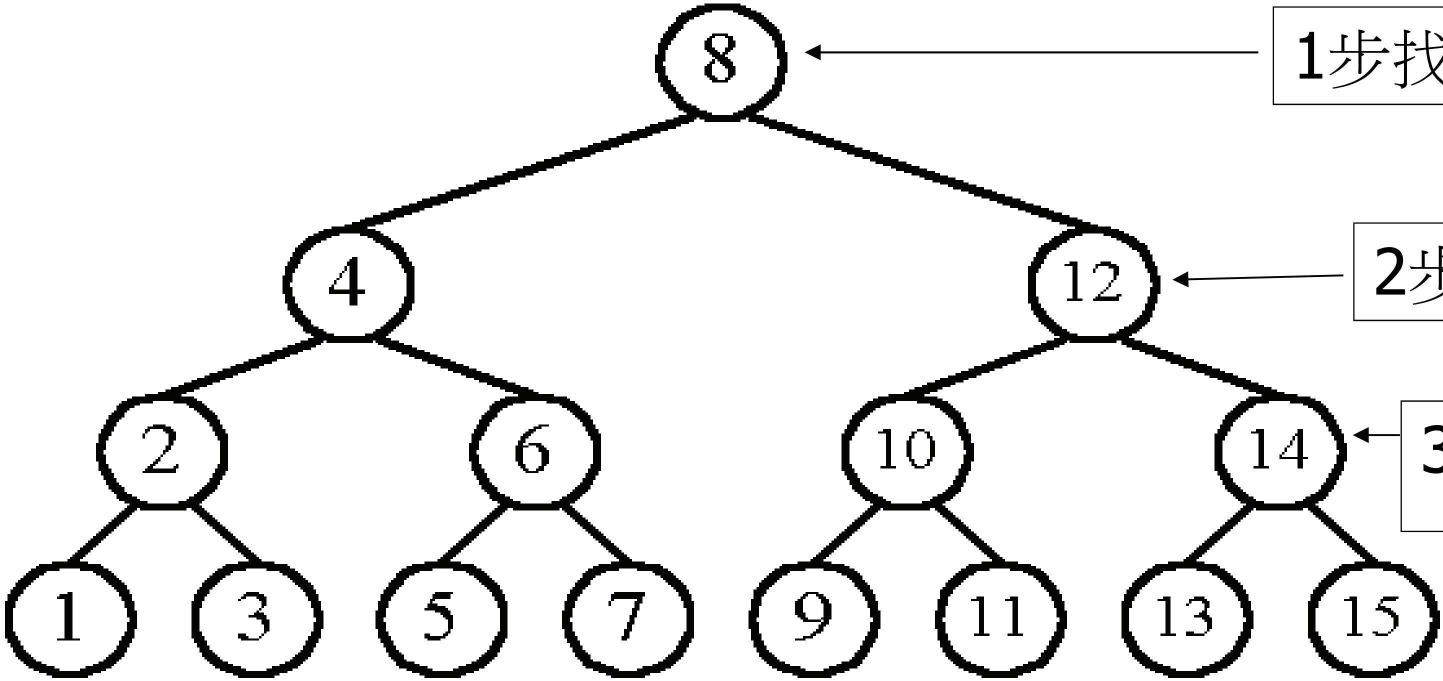
在 $(n+1)$ 種情況裡，每一種都得找 $\lfloor \log n \rfloor + 1$ 次方可確定。

$\therefore$ 平均 “找” 的次數 $A(n) = \dfrac{1}{2n+1}\left(\sum_{i=1}^{k} i \cdot 2^{i-1} + k(n+1)\right)$

（令 $= \lfloor \log n \rfloor + 1$）

利用歸納法 (induction) 可得：

$A(n) < k = O(\lfloor \log n \rfloor)$



1步找到 1個

2步找到 2個

3步找到 4個

i步找到 2^{i-1}個

n cases for successful search

n+1 cases for unsuccessful search

Assume $n=2^k-1$個

Average # of comparisons done in the binary tree:

$$A(n) = \frac{1}{2n+1}\left(\sum_{i=1}^{k} i\,2^{i-1} + k(n+1)\right), \text{ where } k = \lfloor \log n \rfloor + 1$$

K步找不到 (n+1)個

Assume $n=2^k$

$$\sum_{i=1}^{k} i\,2^{i-1} = 2^k(k-1)+1 \qquad (2\text{-}1)$$

proved by induction on k (skip, ref. p.25)

Assume $n=2^k-1$個, $n+1=2^k$

$$A(n)=\frac{1}{2n+1}\left(\sum_{i=1}^{k} i\,2^{i-1}+k(n+1)\right)$$

$$A(n)=\frac{1}{2n+1}((k-1)2^k+1+k(2^k))$$

$$A(n)\approx\frac{1}{2^{k+1}}(2^k(k-1)+1+k2^k)$$

$$=\frac{(k-1)}{2}+\frac{k}{2}=k-\frac{1}{2}$$

$\approx k= \log n= O(\log n)$ as n is very large

Example 2-3 Straight selection sort

- Find the smallest number.
- Let this smallest number occupy a_1 by exchanging a_1 with this smallest number.
- Repeat the above step on the remaining numbers. That is, find the second smallest number and let it occupy a_2.
- Continue the process until the largest number is found.

Ex 2.3 Straight Selection Sort

- Input: $a_1, a_2, \ldots, a_n$.
- Output: The sorted sequence of $a_1, a_2, \ldots, a_n$.

For j :=1 to $n-1$ do

Begin

$\boxed{f := j}$

Flag used to point the Smallest element

For k := j+1 to n do

If $a_k < a_f$ then $f := k$

$a_j \leftrightarrow a_f$

End

Two operations:
(1) comparison
(2) change flag

Straight selection sort

- 7 5 1 4 3
  1 5 7 4 3
  1 3 7 4 5
  1 3 4 7 5
  1 3 4 5 7

  7>5 change
  5>1 change
  1<4 no change
  1<3 no change

- **The number of comparisons** of two elements is a fixed number; namely $n(n-1)/2$. That is, no matter what the input data are, we always have to perform $n(n-1)/2$ comparisons.
- Only consider # of changes in the flag which is used for selecting the smallest number in each iteration.
  - best case: $O(1)$ sorted sequence
  - worst case: $O(n^2)$
  - average case: $O(n \log n)$

The change of flag depends upon the data. Consider $n = 2$. There are only two permutations:

(1, 2)

and (2, 1).

For the first permutation, no change of flag is necessary while for the second permutation, one change of flag is necessary.

Let $f(a_1, a_2, \ldots, a_n)$ denote the number of changing of flags needed to find the smallest number for the permutation $a_1, a_2, \ldots, a_n$. The following table illustrates the case for $n = 3$.

a_1,	a_2,	a_3	$f(a_1, a_2, a_3)$
1,	2,	3	0
1,	3,	2	0
2,	1,	3	1
2,	3,	1	1
3,	1,	2	1
3,	2,	1	2

*$f(a_1, a_2, \ldots, a_n)$ 找出最小數所需改變 **flag** 次數*

Recursive formula

To determine $f(a_1, a_2, \ldots, a_n)$, we note the following:

(1) If $a_n = 1$, then $f(a_1, a_2, \ldots, a_n) = 1 + f(a_1, a_2, \ldots, a_{n-1})$ because there must be a change of flag at the last step.

(2) If $a_n \neq 1$, then $f(a_1, a_2, \ldots, a_n) = f(a_1, a_2, \ldots, a_{n-1})$ because there must not be a change of flag at the last step.

Let $\boxed{P_n(k)}$ denote the probability that a permutation $a_1, a_2, \ldots, a_n$ of $\{1, 2, \ldots, n\}$ needs k changes of flags to find the smallest number. For instance $P_3(0) = \frac{2}{6}$, $P_3(1) = \frac{3}{6}$ and $P_3(2) = \frac{1}{6}$. Then the average number of changes of flags to find the smallest number is

$$\boxed{X_n = \sum_{k=0}^{n-1} k P_n(k).}$$

X_n :n 個數時的平均次數

The average number of changes of flag for the straight selection sort is

$$\boxed{A(n) = X_n + A(n-1).}$$

To find X_n, we shall use the following equations which we discussed before:

$$\begin{aligned} f(a_1, a_2, \ldots, a_n) &= 1 + f(a_1, a_2, \ldots, a_{n-1}) && \text{if } a_n = 1 \\ &= f(a_1, a_2, \ldots, a_{n-1}) && \text{if } a_n \neq 1. \end{aligned}$$

Based upon the above formulas, we have

$$\boxed{P_n(k) = P(a_n = 1)P_{n-1}(k-1) + P(a_n \neq 1)P_{n-1}(k).}$$

But $P(a_n = 1) = 1/n$ and $P(a_n \neq 1) = (n-1)/n$. Therefore, we have

$$\boxed{P_n(k) = \frac{1}{n}P_{n-1}(k-1) + \frac{n-1}{n}P_{n-1}(k).} \qquad (2.2)$$

Furthermore, we have the following formula concerning with the initial conditions:

$P_n(k)$: n個數字的排列，找最小數需要改變flag k次的機率

$$\boxed{\begin{array}{ll} P_1(k) = 1 & \text{if } k=0 \\ \quad = 0 & \text{if } k \neq 0 \\ P_n(k) = 0 & \text{if } k < 0, \text{ and if } k = n. \end{array}} \qquad (2.3)$$

To give the reader some feeling about the formulas, let us note that

$$P_2(0) = \frac{1}{2}$$

and $$P_2(1) = \frac{1}{2};$$

$$P_3(0) = \frac{1}{3}P_2(-1) + \frac{2}{3}P_2(0) = \frac{1}{3}\times 0 + \frac{2}{3}\times\frac{1}{2} = \frac{1}{3}$$

and $$P_3(2) = \frac{1}{3}P_2(1) + \frac{2}{3}P_2(2) = \frac{1}{3}\times\frac{1}{2} + \frac{2}{3}\times 0 = \frac{1}{6}.$$

In the following, we shall prove:

$$\boxed{X_n = \sum_{k=1}^{n-1} kP_n(k) = \frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2} = H_n - 1, \qquad (2.4)}$$

Proof by Induction (see page 30)

Since the average time-complexity of the straight selection sort is:

$$A(n) = X_n + A(n-1),$$

we have

$$\boxed{\begin{aligned} A(n) &= H_n - 1 + A(n-1) \\ &= (H_n - 1) + (H_{n-1} - 1) + \ldots + (H_2 - 1) \\ &= \sum_{i=2}^{n} H_i - (n-1)\,. \qquad (2.5) \end{aligned}}$$

$$\begin{aligned} \sum_{i=1}^{n} H_i &= H_n + H_{n-1} + \ldots + H_1 \\ &= H_n + (H_n - \frac{1}{n}) + \ldots + (H_n - \frac{1}{n} + \frac{1}{n-1} - \ldots - \frac{1}{2}) \\ &= nH_n - (\frac{n-1}{n} + \frac{n-2}{n-1} + \ldots + \frac{1}{2}) \\ &= nH_n - (1 - \frac{1}{n} + 1 - \frac{1}{n-1} + \ldots + 1 - \frac{1}{2}) \\ &= nH_n - (n - 1 - \frac{1}{n} - \frac{1}{n-1} - \ldots - \frac{1}{2}) \\ &= nH_n - n + H_n \\ &= (n+1)H_n - n\,. \end{aligned}$$

Straight selection sort

Therefore

$$\boxed{\sum_{i=2}^{n} H_i = (n+1)H_n - H_1 - n\,.}$$

Substituting (2.6) into (2.5), we have

$$A(n) = (n+1)H_n - H_1 - (n-1) - n$$

$$= (n+1)H_n - 2n\,.$$

Rosen, p255, ex.57, 58

As n is large enough,

$$A(n) \cong n \log_e n = O(n \log n)\,.$$

$$\boxed{\begin{array}{l} 1 + \frac{n}{2} \le H_{2^n} \le 1 + n \\ H_k \le 1 + \log_2 K \end{array}}$$

Example 2-4 QuickSort

- Quicksort is based upon the divide-and-conquer strategy.
- Divide-and-conquer strategy divides a problem into two sub-problems and solves these two subproblems individually and independently. We later merge the results.
- Applying this divide-and-conquer strategy to sort, we have a sorting method, called Quicksort.
- Given a set of numbers $a_1, a_2, \ldots, a_n$ we choose an element X to divide $a_1, a_2, \ldots, a_n$ into two lists.
- After the dividing, we may apply Quicksort to both L_1 and L_2 recursively and the resulting list is a sorted list.

QuickSort

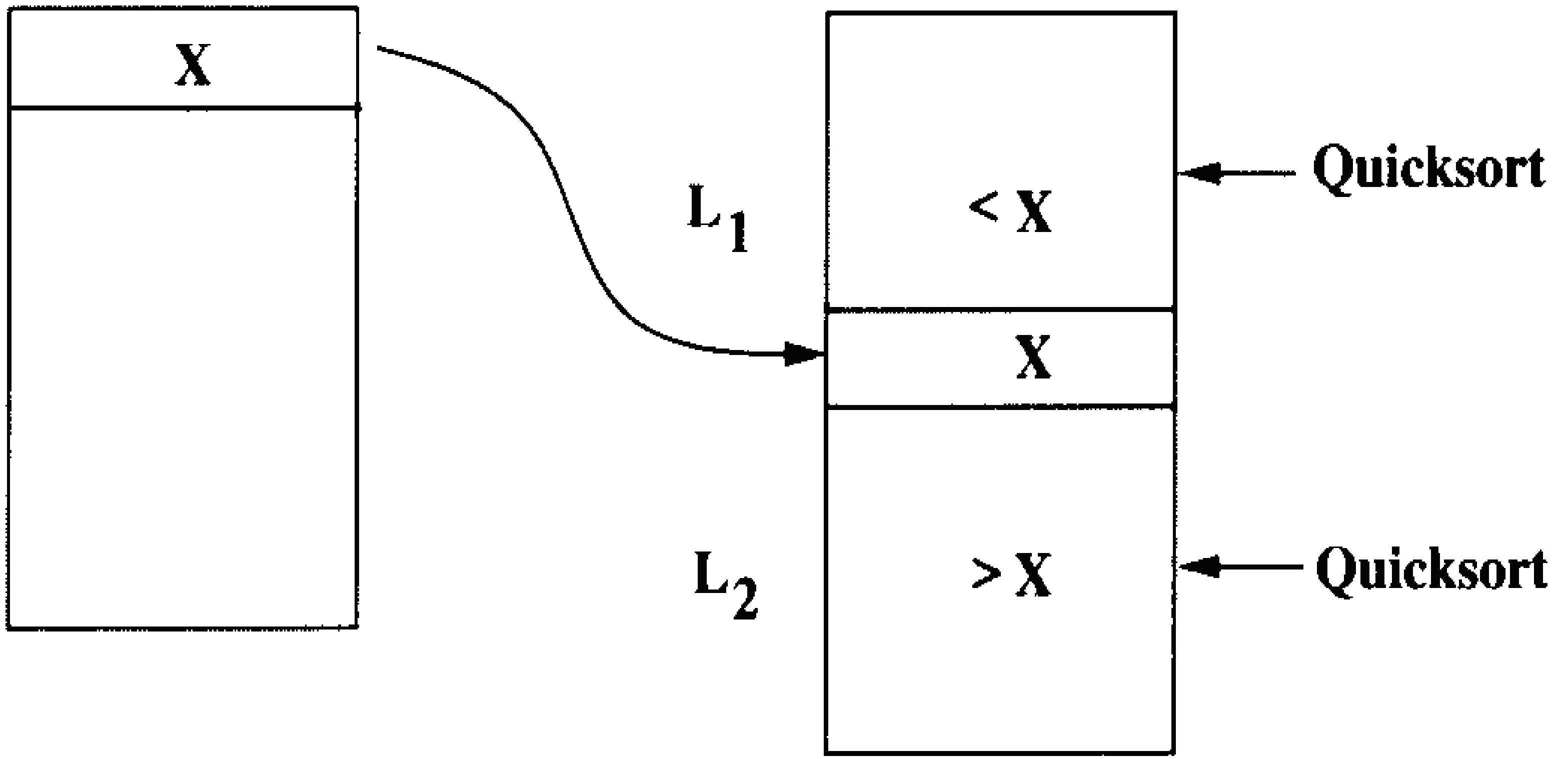


Figure 2-1 Quicksort.

Quicksort

11	5	24	2	31	7	8	26	10	15
		↑						↑	
11	5	10	2	31	7	8	26	24	15
				↑		↑			
11	5	10	2	8	7	31	26	24	15
△					△				
7	5	10	2	8	11	31	26	24	15
|←		<11		→|		|←		>11	→|

- Recursively apply the same procedure.

Quick-sort Example

40	20	10	80	60	50	7	30	100

- Given a pivot, partition the elements of the array such that the resulting array consists of:
  - One sub-array that contains elements >= pivot
  - Another sub-array that contains elements < pivot

pivot_index = 0

40	20	10	80	60	50	7	30	100
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	60	50	7	80	100

40	20	10	30	60	50	7	80	100

40	20	10	30	7	50	60	80	100

pivot_index = 0

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
40	20	10	30	7	50	60	80	100

too_big_index too_small_index

pivot_index = 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
7	20	10	30	40	50	60	80	100

too_big_index too_small_index

7	20	10	30	40	50	60	80	100

Algorithm 2–4 □ Quick sort (f, l)

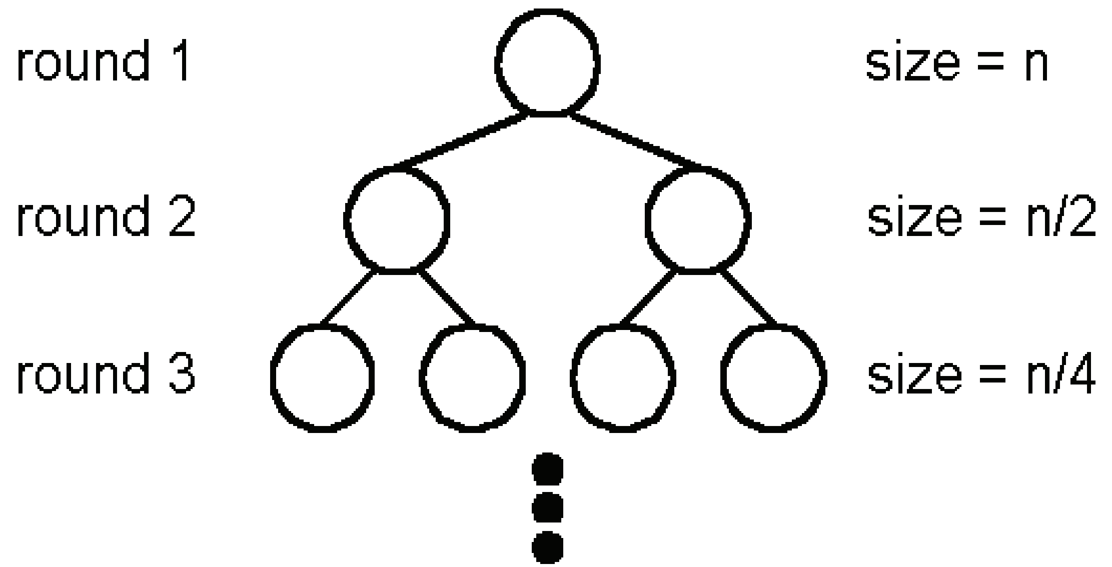
Input: $a_f, a_{f+1}, \ldots, a_l$.

Output: The sorted sequence of $a_f, a_{f+1}, \ldots, a_l$.

```
If f ≥ l then Return
X := a_f
i := f + 1
j := l
While i < j do
Begin
    While a_j ≥ X and j ≥ f + 1 do
        j := j − 1
    While a_i ≤ X and i ≤ l do
        i := i + 1
    if i < j then a_i ↔ a_j
End
a_f ↔ a_j
Quicksort(f, j − 1)
Quicksort(j + 1, l)
```

Best case of Quicksort

- Best case: $O(nlogn)$
- A list is split into two sublists with almost equal size.



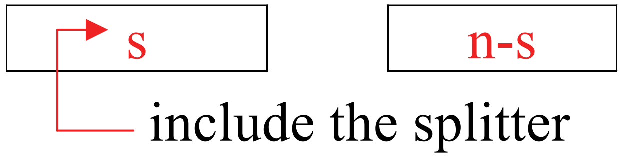
- $log\ n$ rounds are needed
- In each round, n comparisons (ignoring the element used to split) are required.

Worst case of Quicksort

- Worst case: $O(n^2)$
- Sorted or reverse sorted.
- In each round, the number used to split is either the smallest or the largest.
- $$n+(n-1)+\cdots+1=\frac{n(n-1)}{2}=O(n^2)$$

Average case of Quicksort

- Average case: O(n log n)



The number of operations needed for first splitting operation

$$T(n) = \underset{1 \le s \le n}{\mathrm{Avg}}\,(T(s) + T(n-s)) + \boxed{cn}$$

$$= \frac{1}{n}\sum_{s=1}^{n}(T(s) + T(n-s)) + cn$$

$$= \frac{1}{n}(T(1)+T(n-1)+T(2)+T(n-2)+\cdots+T(n)+T(0))+cn,\ T(0)=0$$

$$= \frac{1}{n}(2T(1)+2T(2)+\cdots+2T(n-1)+T(n))+cn$$

$$(n-1)T(n) = 2T(1)+2T(2)+\cdots+2T(n-1) + cn^2 \cdots\cdots (1)$$

$$(n-2)T(n-1)=2T(1)+2T(2)+\cdots+2T(n-2)+c(n-1)^2 \cdots (2)$$

Let n=n-1 to (1)

$$(1) - (2)$$

$$(n-1)T(n) - (n-2)T(n-1) = 2T(n-1)+c(2n-1)$$

$$(n-1)T(n) - nT(n-1) = c(2n-1)$$

部份分式

$$\frac{T(n)}{n} = \frac{T(n-1)}{n-1} + c\left(\frac{1}{n} + \frac{1}{n-1}\right)$$

$$= c\left(\frac{1}{n} + \frac{1}{n-1}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2}\right) + \cdots + c\left(\frac{1}{2} + 1\right) + T(1),\ T(1) = 0$$

$$= c\left(\frac{1}{n} + \frac{1}{n-1} + \ldots + \frac{1}{2}\right) + c\left(\frac{1}{n-1} + \frac{1}{n-2} + \ldots + 1\right)$$

$$= c(H_n - 1) + cH_{n-1}$$

Harmonic number **[Knuth 1986]**

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

$$= \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon, \text{ where } 0 < \varepsilon < \frac{1}{252n^6}$$

$$\gamma = 0.5772156649\cdots.$$

$$H_n = O(\log n)$$

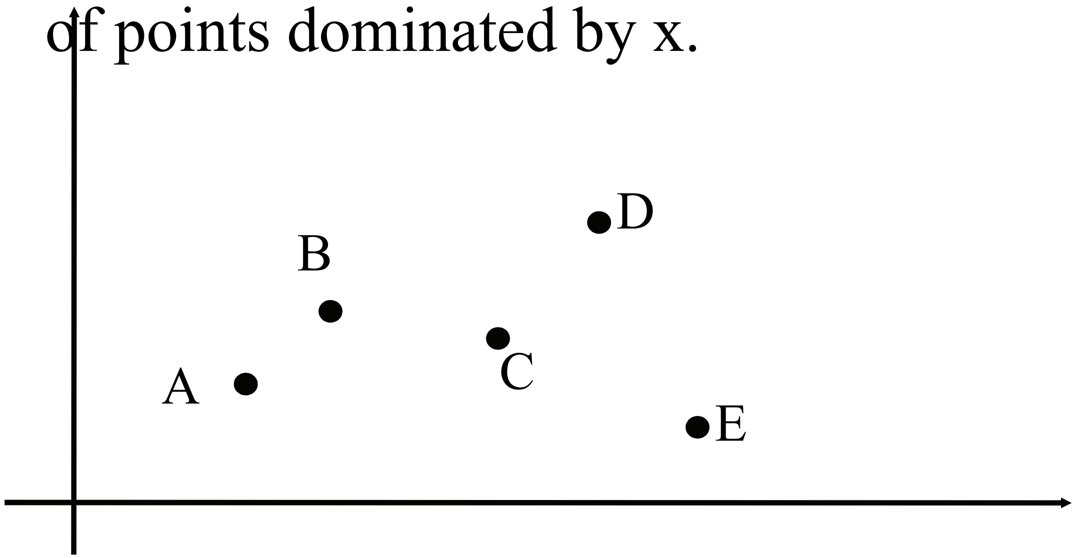
$$T(n)/n = c(H_n - 1) + cH_{n-1}$$

$$\Rightarrow T(n)/n = c(H_n + H_n - 1 - 1/n)$$

$$\Rightarrow T(n) = 2cH_n - c(n+1) = O(n \log n)$$

Example 2-5 2-D ranking finding

- **<u>Def</u>**: Let $A = (a_1,a_2)$, $B = (b_1,b_2)$. A <u>dominates</u> B iff $a_1 > b_1$ and $a_2 > b_2$
- **Def**: If neither A dominates B nor B dominates A, then A and B are incomparable.
- **<u>Def</u>**: Given a set S of n points, the <u>rank</u> of a point x is the number of points dominated by x.



B, C and *D* dominate *A*.
D dominates A, B and C.
All other pairs of points are incomparable.

rank(A)= 0 rank(B) = 1 rank(C) = 1 rank(D) = 3 rank(E) = 0

Rank Finding Problem

- **Find the rank of every points.**
- Straightforward algorithm**:**
  - compare all pairs of points : $O(n^2)$

- **Divide-and-conquer** 2-D ranking finding
  - Step 1: Split the points along the median line L into A and B.
  - Step 2: Find ranks of points in A and ranks of points in B, recursively.
  - Step 3: Sort points in A and B according to their y-values. Update the ranks of points in B.

Local ranks before merge

- Find a straight line L perpendicular to the x-axis which separates the set of points into two subsets and these two subsets are of equal size.
- The rank of any point in A will not be affected by the presence of B. But the rank of a point in B may be affected the presence of A.

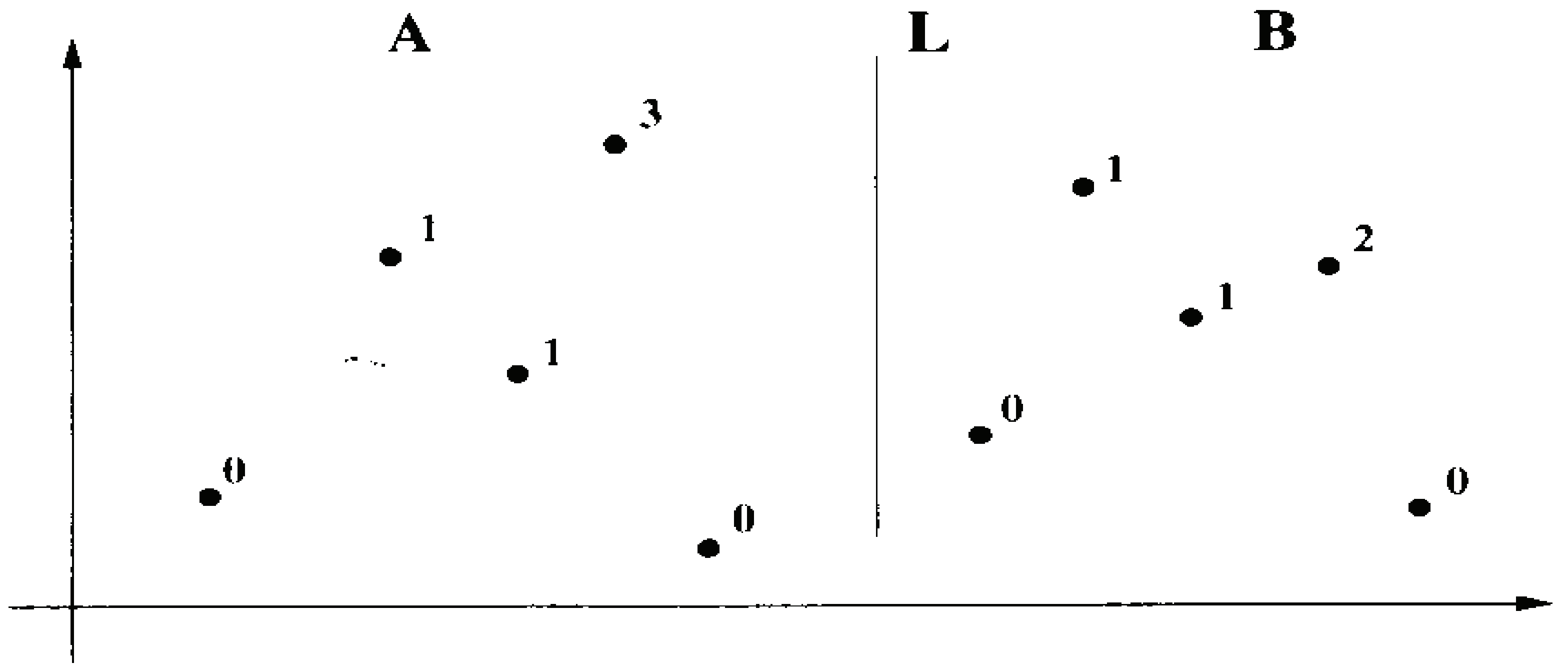
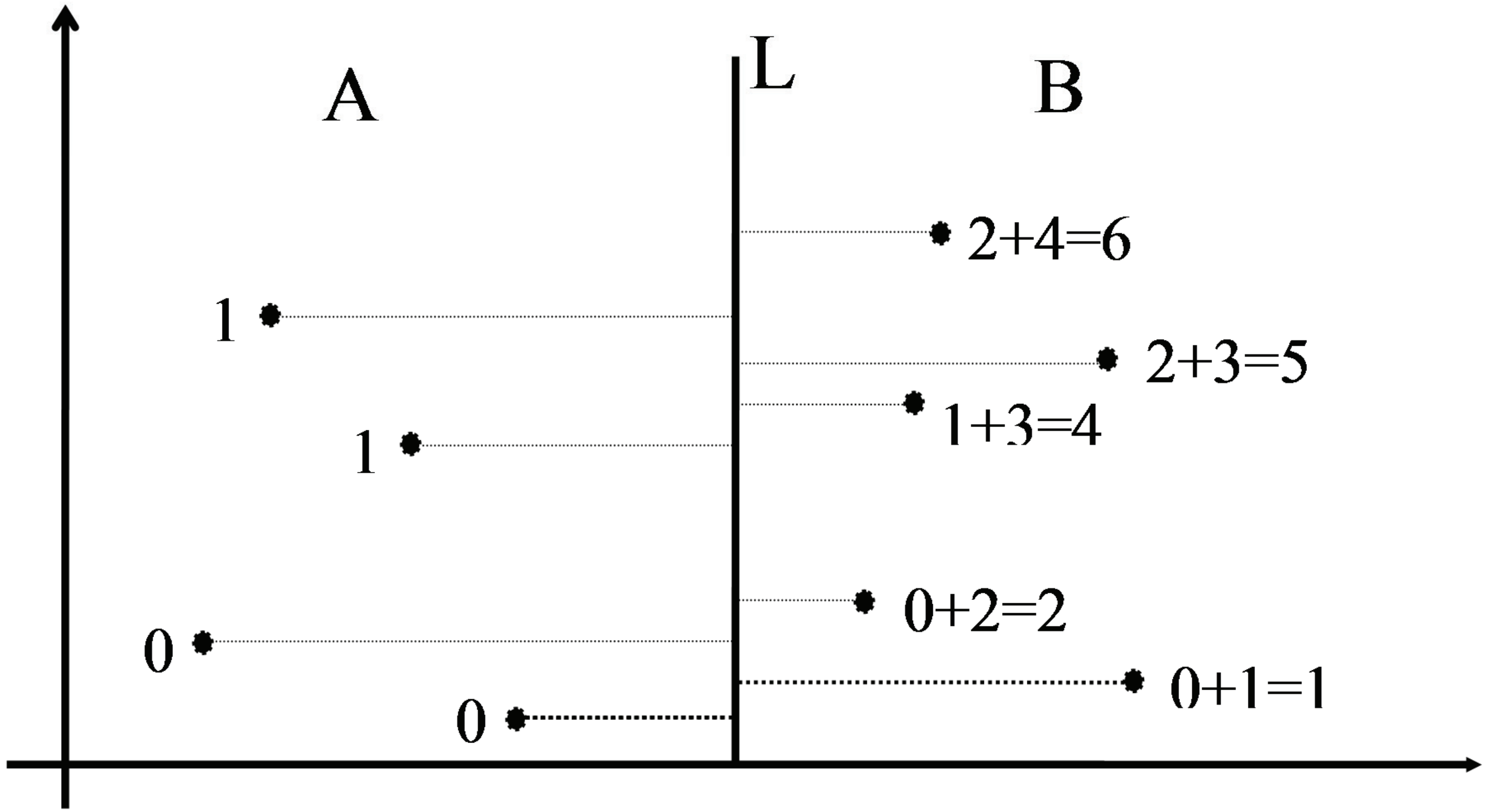


Figure 2-4 The Local Ranks of Point in *A* and *B*.

- More efficient algorithm (divide-and-conquer)



Algorithm 2–5 □ A rank finding algorithm

Input: A set S of planar points $P_1, P_2, \ldots, P_n$.

Output: The rank of every point in S.

Step 1. If S contains only one point, return its rank as 0. Otherwise, choose a cut line L perpendicular to the x-axis such that $n/2$ points of S have X-values less than L (call this set of points A) and the remainder points have X-values greater than L (call this set B). Note that L is a median X-value of this set.

Step 2. Recursively, use this rank finding algorithm to find the ranks of points in A and ranks of points in B.

Step 3. Sort points in A and B according to their y-values. Scan these points sequentially and determine, for each point in B, the number of points in A whose y-values are less than its y-value. The rank of this point is equal to the rank of this point among points in B (found in Step 2), plus the number of points in A whose y-values are less than its y-value.

- time complexity : step 1 : $O(n)$ (finding median)
  step 3 : $O(n \log n)$ (sorting)

- total time complexity :

$$
\begin{aligned}
T(n) &\le 2T(\tfrac{n}{2}) + c_1 n \log n + c_2 n \\
&\le 2T(\tfrac{n}{2}) + c\, n \log n \\
&\le 4T(\tfrac{n}{4}) + c\, n \log \tfrac{n}{2} + c\, n \log n \\
&\le nT(1) + c(n \log n + n \log \tfrac{n}{2} + n \log \tfrac{n}{4} + \cdots + n \log 2) \\
&= nT(1) + \frac{cn \log n(\log n + \log 2)}{2} \\
&= O(n \log^2 n)
\end{aligned}
$$

If $n = 2^p$, $p = \log n$

$$
\begin{aligned}
&\log n + \log \frac{n}{2} + \log \frac{n}{4} + \ldots + \log 2 \\
&= p + (p-1) + (p-2) + \ldots + 1 \\
&= \frac{p(p+1)}{2} = \log n(\log n + 1) = (\log n)^2
\end{aligned}
$$

For average & worst case

2-3 Lower bound

- **How to we measure the difficulty of a problem?**
- **Def** : A lower bound of a problem is the least time complexity required for any algorithm which can be used to solve this problem.
- ☆ **worst case lower bound**

  ☆ **average case lower bound**
- Def : $f(n) = \Omega(g(n))$ "at least", "lower bound"

  $\exists$ c, and n_0, $\ni |f(n)| \geq c|g(n)| \ \forall \ n \geq n_0$

  e. g. $f(n) = 3n^2 + 2 = \Omega(n^2)$ or $\Omega(n)$
- The lower bound for a problem is not unique.
  - e.g. $\Omega(1)$, $\Omega(n)$, $\Omega(n \log n)$ are all lower bounds for sorting.
  - ($\Omega(1)$, $\Omega(n)$ are trivial)

Trivial lower bound

- ex.: sorting

$\Omega(1)$, $\Omega(n)$均為 trivial lower bound，討論它們沒有意義！

$\Omega(n^2)$如何？已有heapsort 其worst case為$\Omega(n\log n)$由 Def. 可知 lower bound 必須是所有 algorithms 中最小者，所以 $\Omega(n^2)$ 也不對！lower bound 至多是$\Omega(n\log n)$。

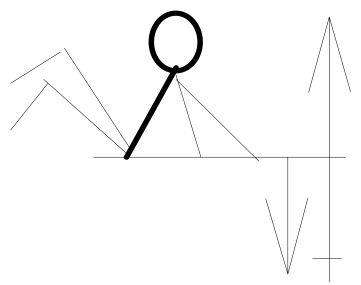
$\Omega(n\log n)$

$\Omega(n)$
每個值都得scan一次!

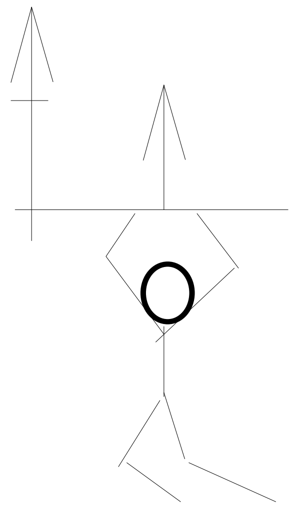
$\Omega(1)$
至少做一次!

- 若目前 problem 之 highest lower bound 為 $\Omega(n \log n)$ 而找到的algorithm 最快的是$O(n^2)$，則：

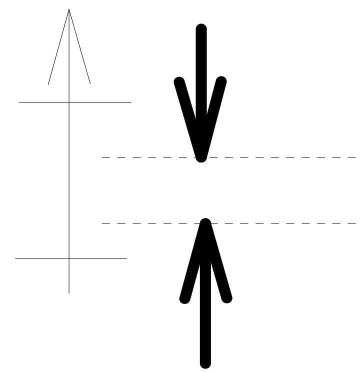
(1)



(2)



(3)



- 若問題的 lower bound 為$\Omega(n\log n)$且找到的 algorithm 的 time-complexity 為 $O(n\log n)$
則 optimal algorithm of this problem 即已找到！
lower bound 與 algorithm 都無法再 improve。

2.4 The worst case lower bound of sorting

- **Execution of an algorithm can be represented as binary trees.**
- **In general, any sorting algorithm whose basic operation is <u>compare</u> and <u>exchange</u> operation can be described by a binary tree.**
- **Straight insertion sort.**

6 permutations for 3 data elements

a_1	a_2	a_3
1	2	3
1	3	2
2	1	3
2	3	1
3	1	2
3	2	1

Straight insertion sort

- input data: (2, 3, 1)

  (1) $a_1:a_2$

  (2) $a_2:a_3$, $a_2 \leftrightarrow a_3$

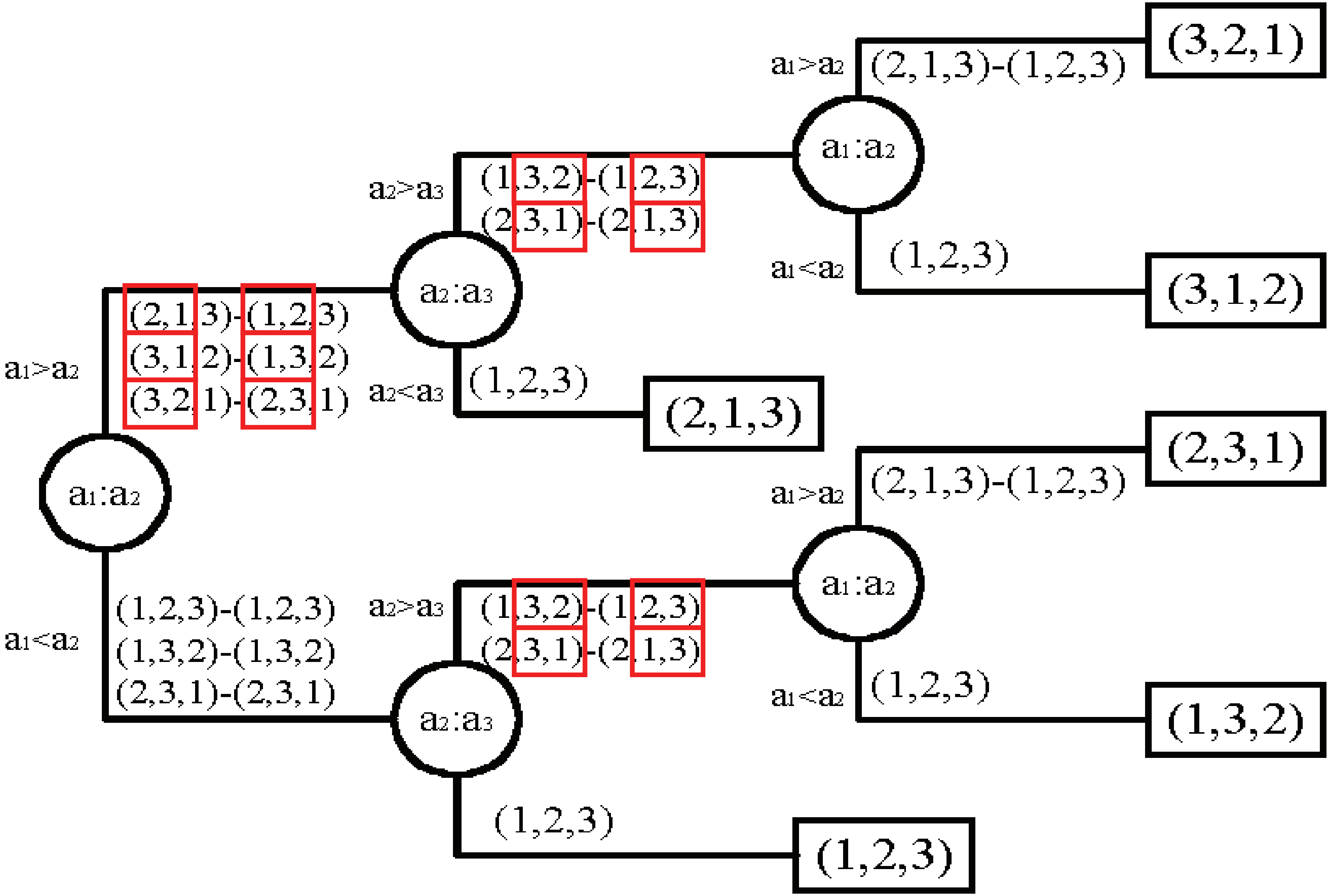
  (3) $a_1:a_2$, $a_1 \leftrightarrow a_2$

- input data: (2, 1, 3)

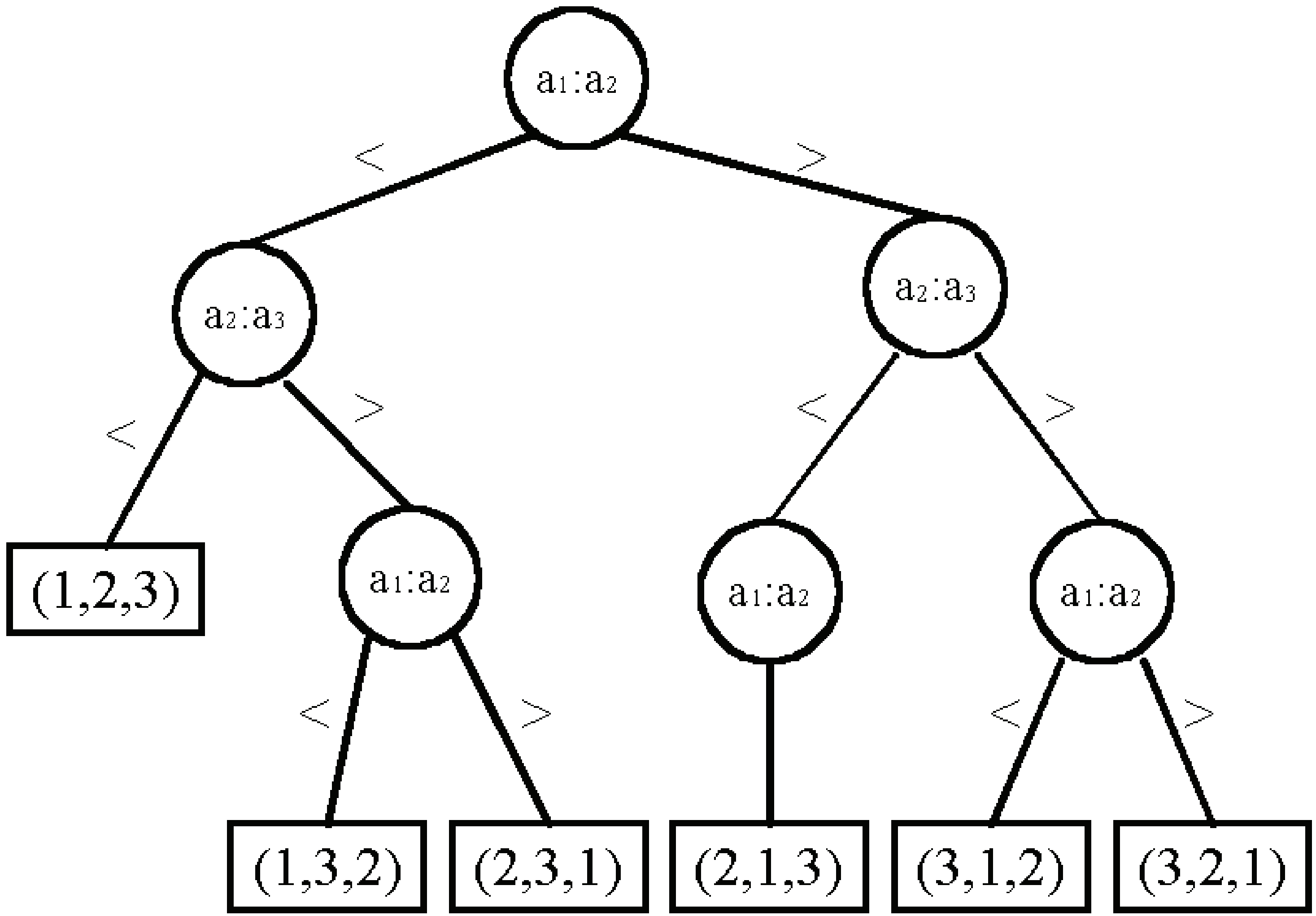
  (1)$a_1:a_2$, $a_1 \leftrightarrow a_2$

  (2)$a_2:a_3$

Decision tree for straight insertion sort



Decision tree for bubble sort



Lower bound of sorting

- The action of a sorting algorithm based upon compare and exchange operations on a particular input data set corresponds to one path from the top of the tree to a leaf node
- Each ***leaf node*** therefore corresponds to a particular permutation.
- *The longest path from the top of the tree to a leaf node, which is called the depth of the tree, represents the worst case time-complexity o this algorithm.*
- To find the lower bound of the sorting problem, we have to find the smallest depth of some tree, among all possible binary decision trees modeling sorting algorithms.

Lower bound of sorting

- To find the lower bound, we have to find the depth of a binary tree with the smallest depth.
- *n!* distinct permutations

  n! leaf nodes in the binary decision tree.
- balanced tree has the smallest depth:

  $\lceil \log(n!) \rceil = \Omega(n \log n)$

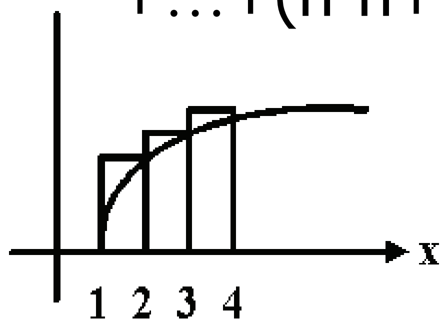
  lower bound for sorting: $\Omega(n \log n)$
-

(See the next page.)

Method 1:

$$\begin{aligned}
\log(n!) &= \log(n(n-1)\cdots 1) \\
&= \log 2 + \log 3 + \cdots + \log n \\
&> \int_1^n \log x\,dx \\
&= \log e \int_1^n \ln x\,dx \\
&= \log e[x \ln x - x]_1^n \\
&= \log e(n \ln n - n + 1) \\
&= n \log n - n \log e + 1.44 \\
&\geq n \log n - 1.44n \\
&= \Omega(n \log n)
\end{aligned}$$

=(2-1)log2+(3-2)log3 +…+(n-n+1)logn



$$\boxed{\log_a b = \frac{\ln b}{\ln a}}$$

$$\boxed{\int \ln x\,dx = x \ln x - x + C}$$

Method 2:

- Stirling approximation:

$$\boxed{n! \approx S_n = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n}$$

$$\log n! \approx \log\sqrt{2\pi} + \frac{1}{2}\log n + n\log\frac{n}{e} \approx n\log n = \Omega(n\log n)$$

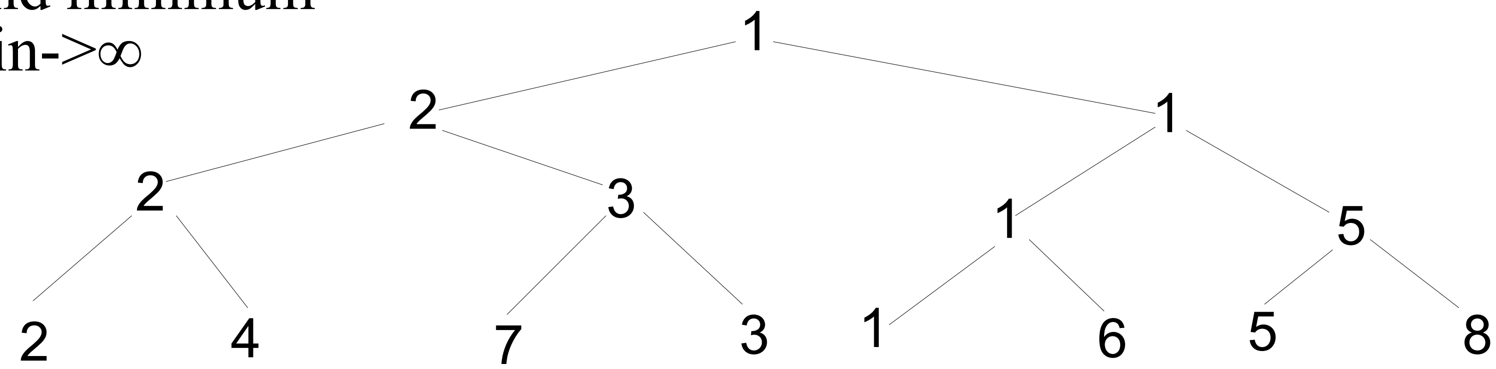
n	n!	S_n
1	1	0.922
2	2	1.919
3	6	5.825
4	24	23.447
5	120	118.02
6	720	707.39
10	3,628,800	3,598,600
20	2.433×10^{18}	2.423×10^{18}
100	9.333×10^{157}	9.328×10^{157}

2.5 knockout sort

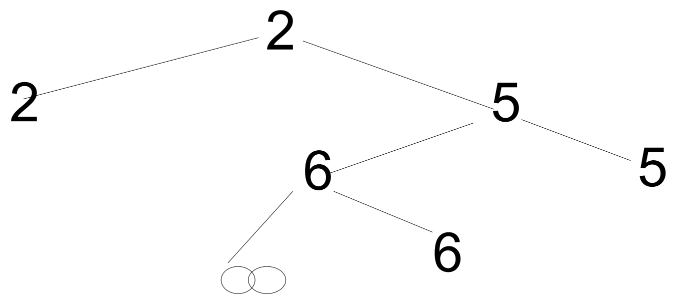
- Note that when we try to find the second smallest number, the information we may have extracted by finding the first smallest number is not used at all.
- This is why the straight insertion sort behaves so clumsily.
- It keeps some information after it finds the first smallest number so that it is quite efficient to find the <u>second smallest number</u>.

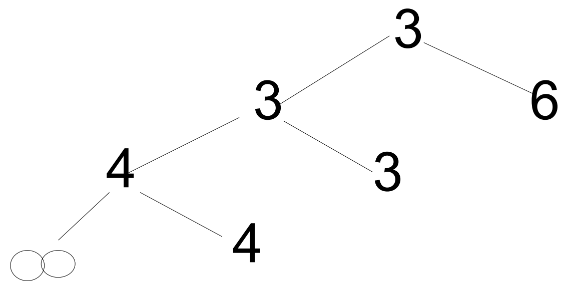
Knockout sort (example)

Input: 2, 4, 7, 3, 1, 6, 5,8
Construct Knockout tree
Find minimum
Min->∞



(n-1) comparisons





. . . . .

($\lceil \log n \rceil$ -1)

Time complexity of Knockout(淘汰) sort

- The first smallest number is found after $(n-1)$ comparisons.
- For all of the other selections, only $\lceil \log n \rceil - 1$ comparisons are needed. Therefore the total number of comparisons is

  $$(n-1)+(n-1)(\lceil \log n \rceil - 1).$$

- Thus the time-complexity of knockout sort is $O(n\log n)$ which is equal to the lower.
- Knockout sort is therefore <u>an optimal sorting</u> algorithm.
- We must note that the time-complexity $O(n\log n)$ is valid for best, average and worst cases.
- Drawbacks: **space 2n**.

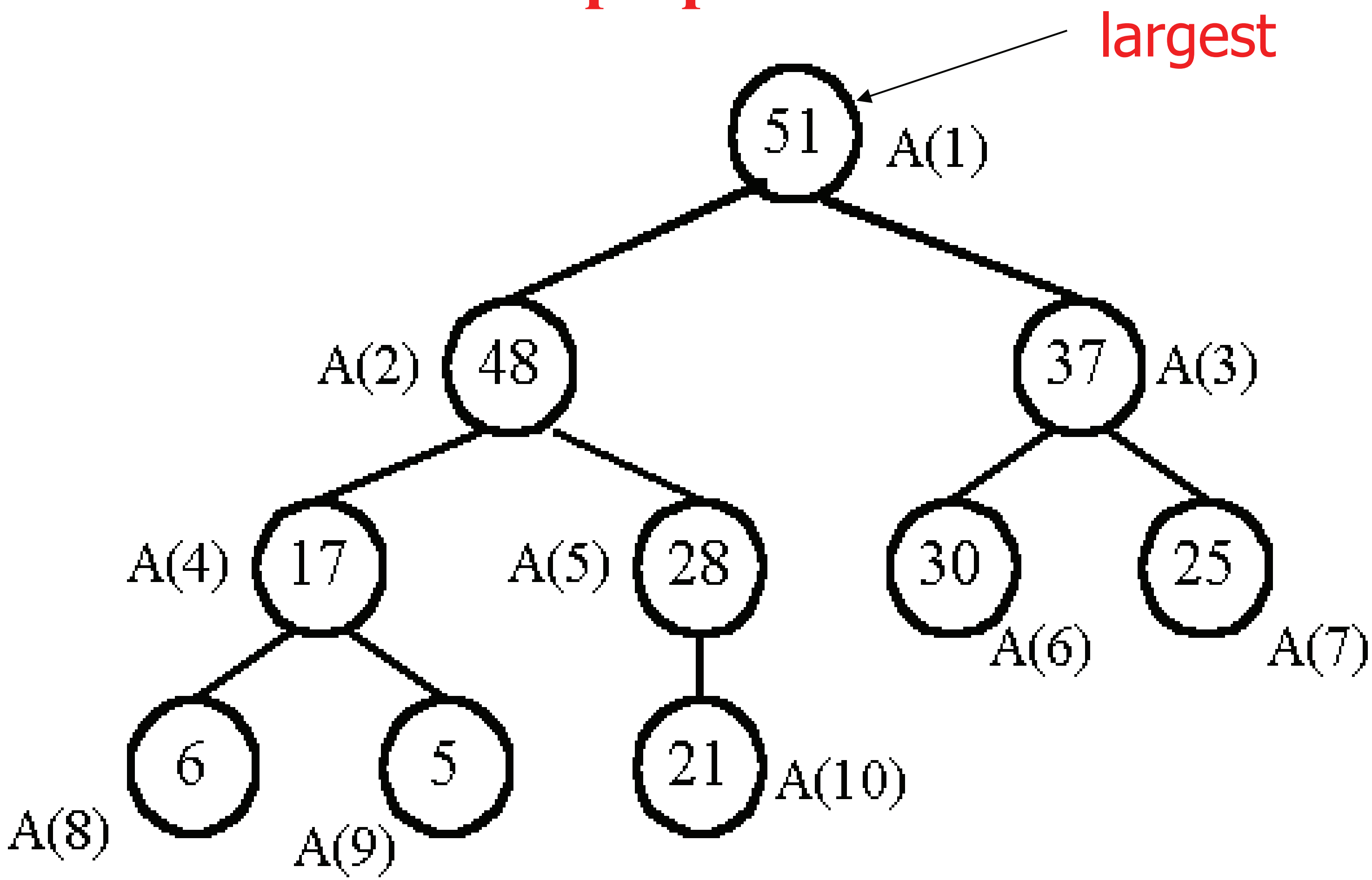
Heap

- A heap is a binary tree satisfying the following conditions:
  - This tree is completely balanced.
  - If the height of this binary tree is h, then leaves can be at level h or level h-1.
  - All leaves at level h are as far to the left as possible.
  - The data associated with all descendants of a node are smaller than the datum associated with this node.

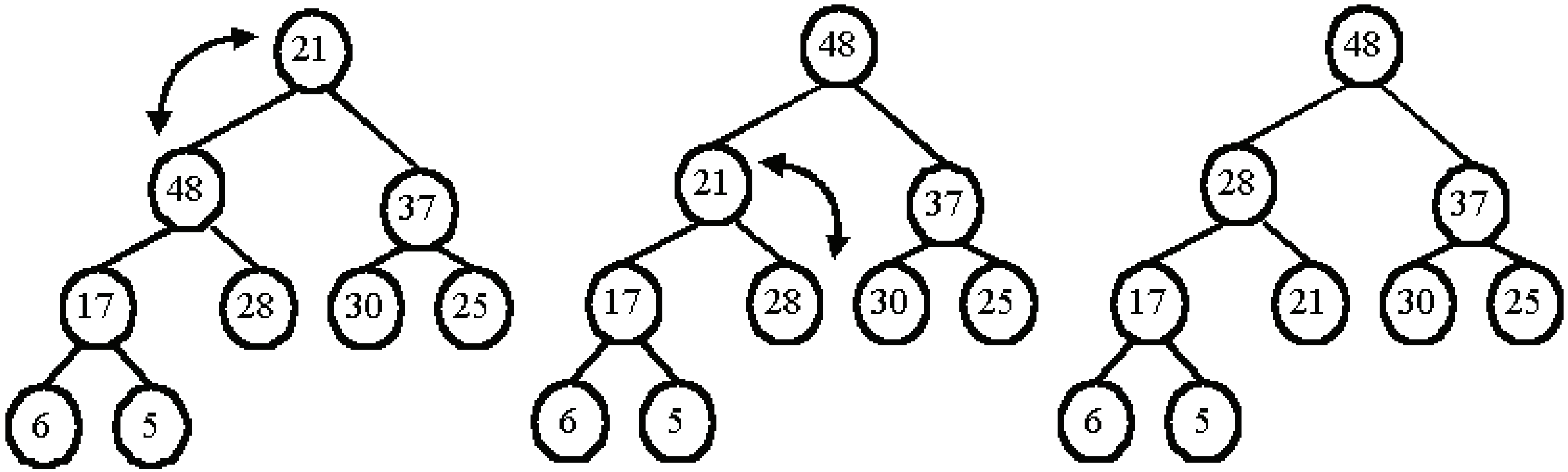
Max-heap or min-heap

Heapsort—An optimal sorting algorithm

- A **maximal heap : parent ≥ son**



- output the maximum and restore:

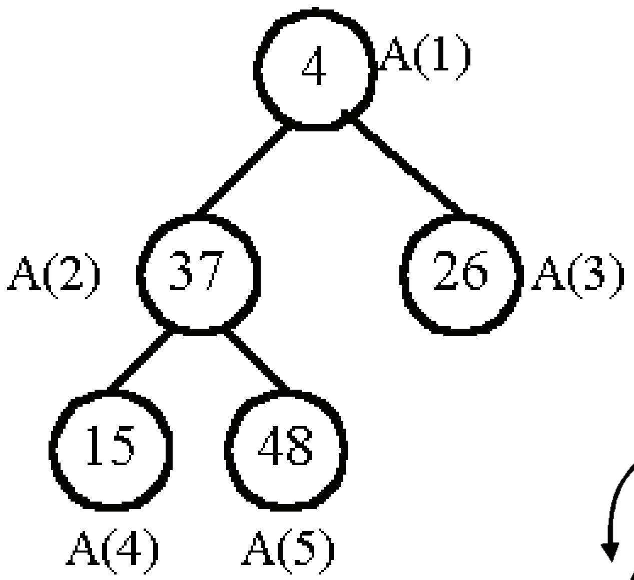


(a) (b) (c)

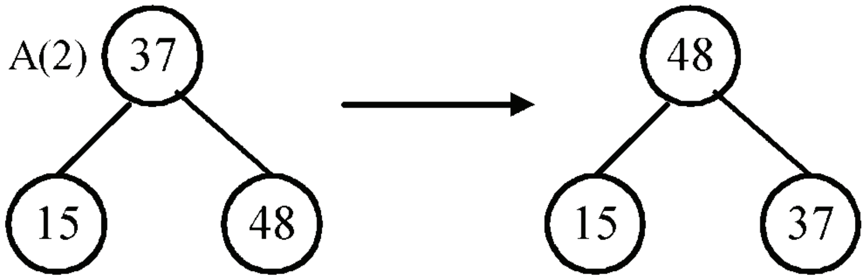
- Heapsort:
  - Phase 1: Construction
  - Phase 2: Output

Phase 1: construction

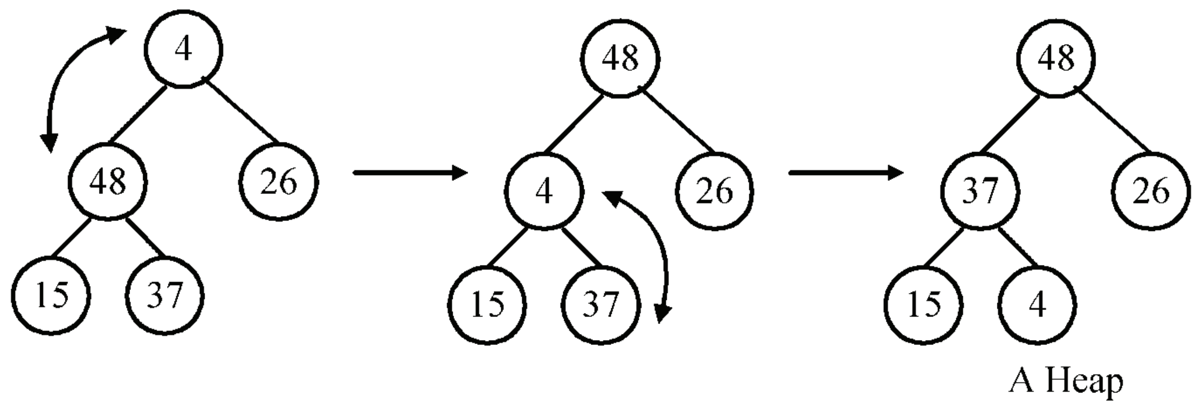
- input data: 4, 37, 26, 15, 48



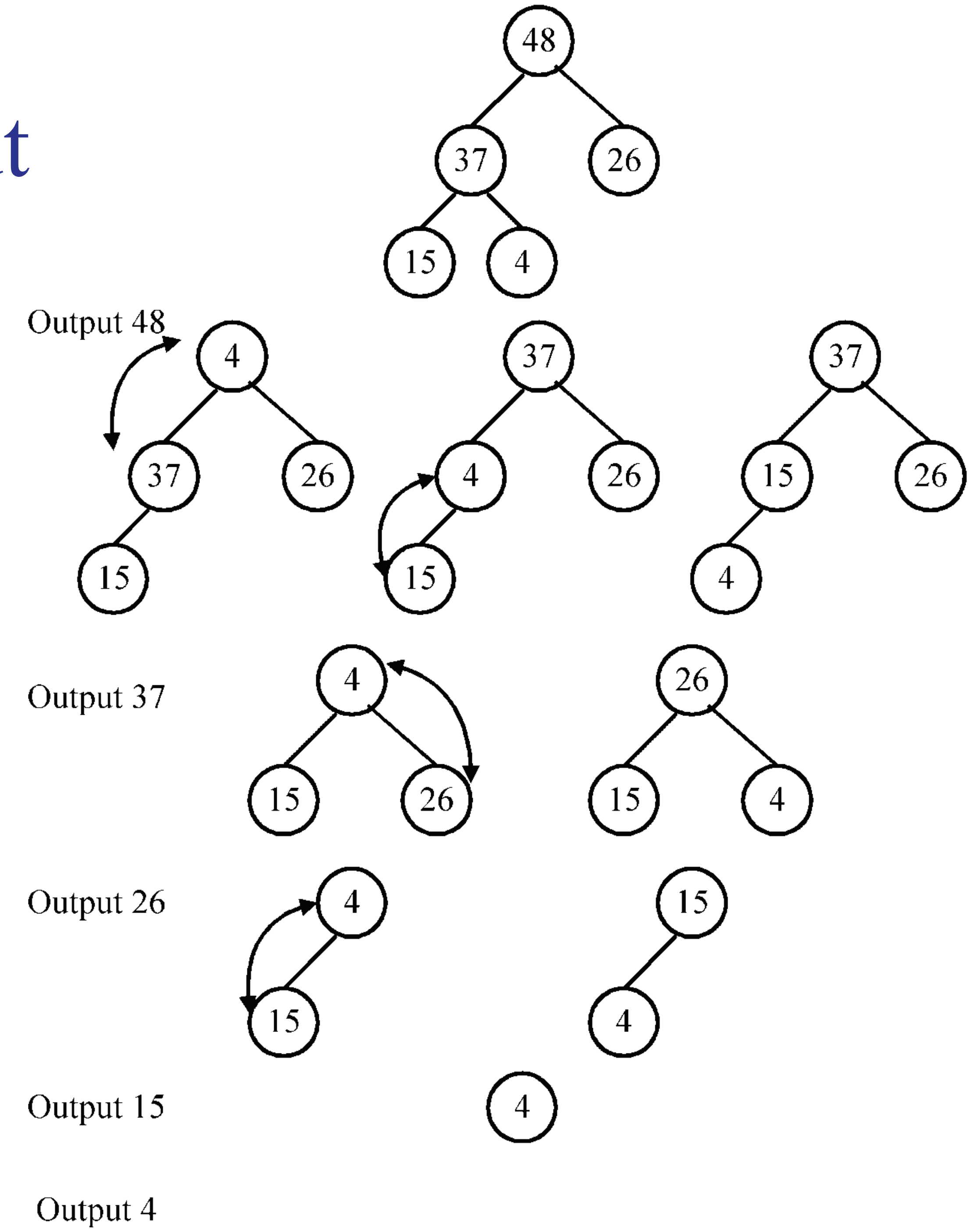
- restore the subtree rooted at A(2):



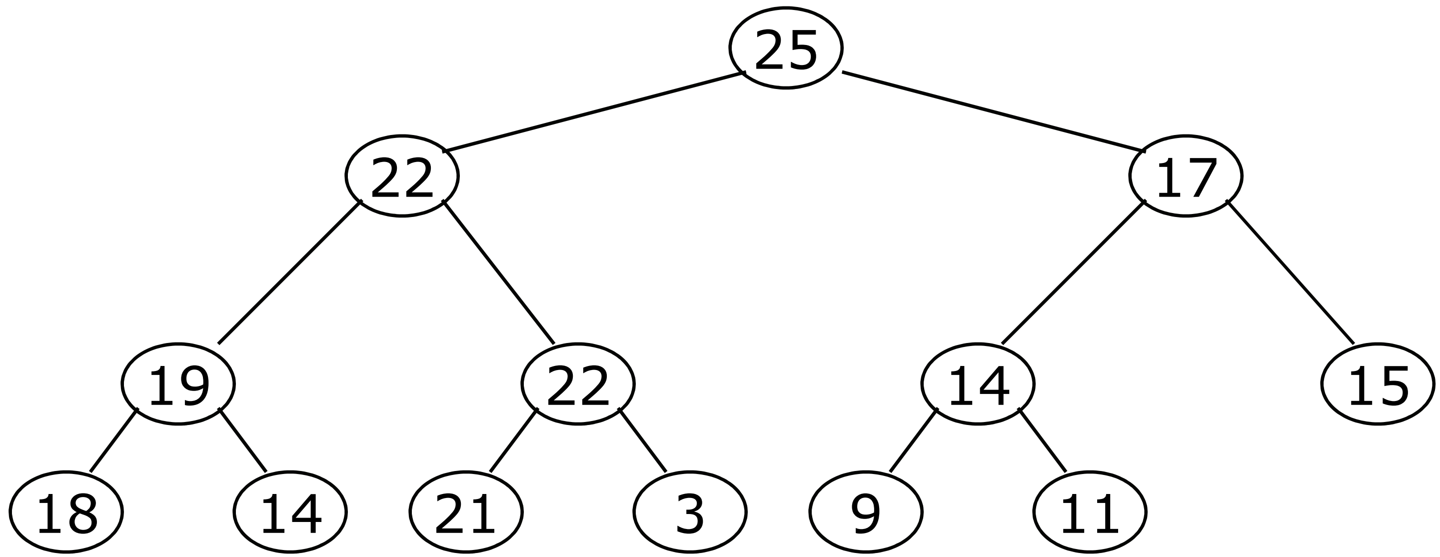
- restore the tree rooted at A(1):



Phase 2: output



Implementation



0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

- Notice:
  - The left child of index i is at index 2*i+1
  - The right child of index i is at index 2*i+2
  - Example: the children of node 3 (19) are 7 (18) and 8 (14)

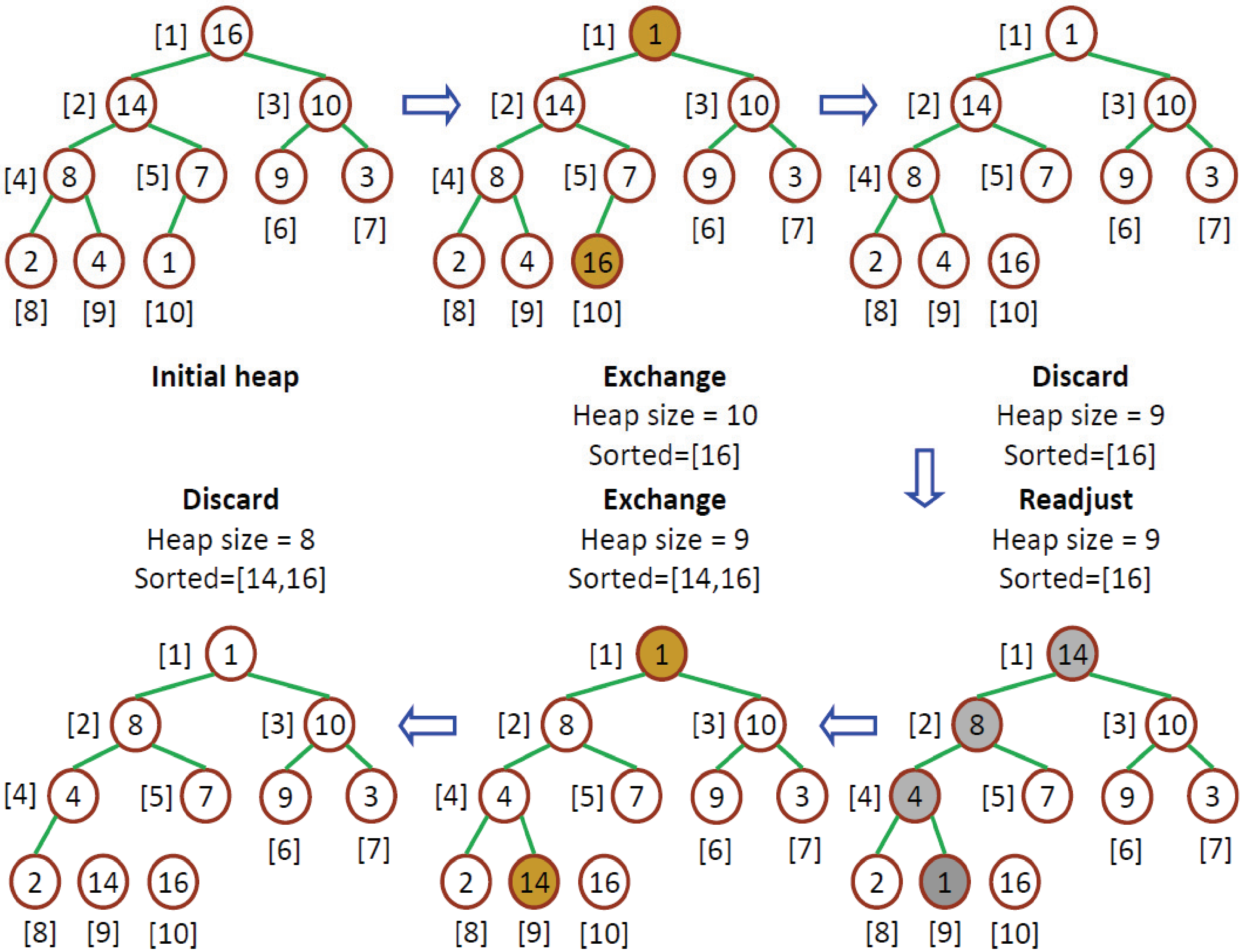
Removing and replacing the root

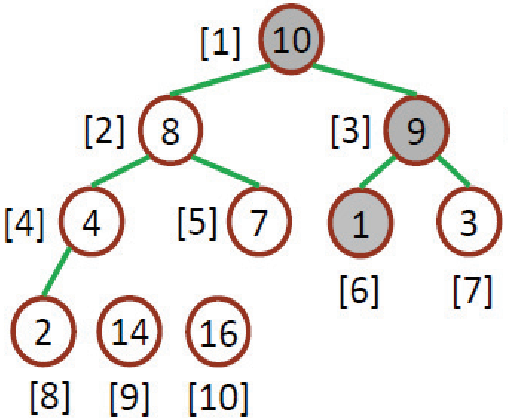
- The "root" is the first element in the array
- The "rightmost node at the deepest level" is the last element
- Swap them...

0	1	2	3	4	5	6	7	8	9	10	11	12
25	22	17	19	22	14	15	18	14	21	3	9	11

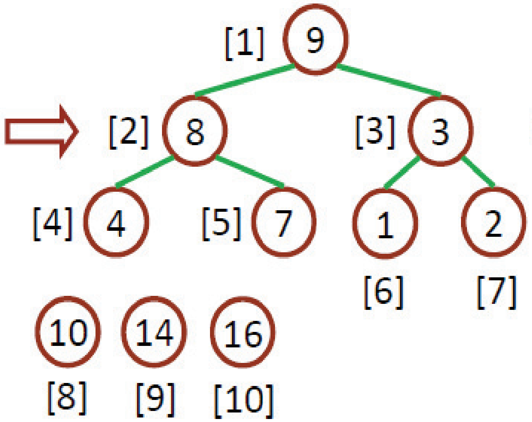
0	1	2	3	4	5	6	7	8	9	10	11	12
11	22	17	19	22	14	15	18	14	21	3	9	25

- ...And pretend that the last element in the array no longer exists—that is, the "last index" is 11 (9)

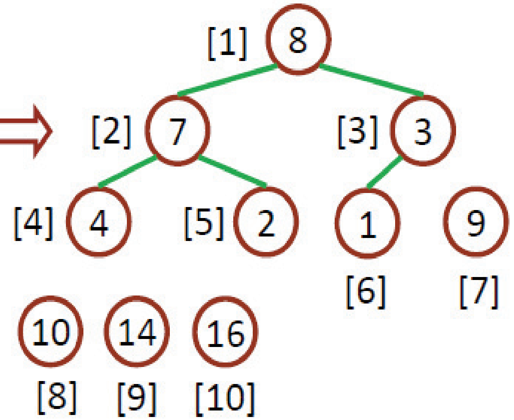

Initial heap
Exchange
Heap size = 10
Sorted=[16]
Discard
Heap size = 9
Sorted=[16]
Readjust
Heap size = 9
Sorted=[16]
Exchange
Heap size = 9
Sorted=[14,16]
Discard
Heap size = 8
Sorted=[14,16]



Readjust
Heap size = 8
Sorted=[14,16]

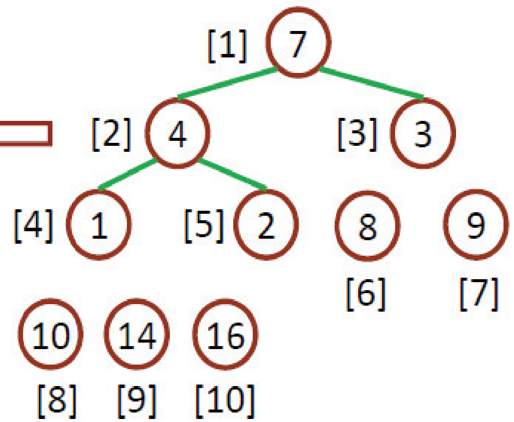


Heap size = 7
Sorted=[10,14,16]

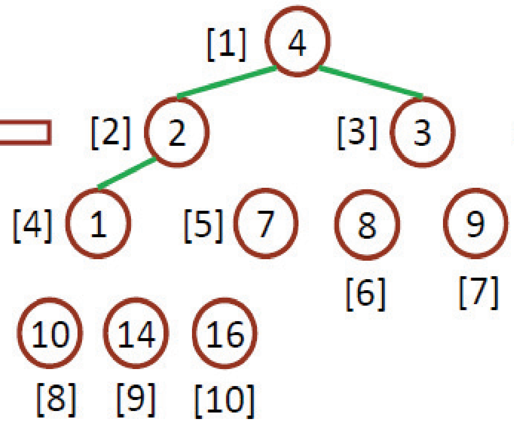


Heap size = 6
Sorted=[9,10,14,16]

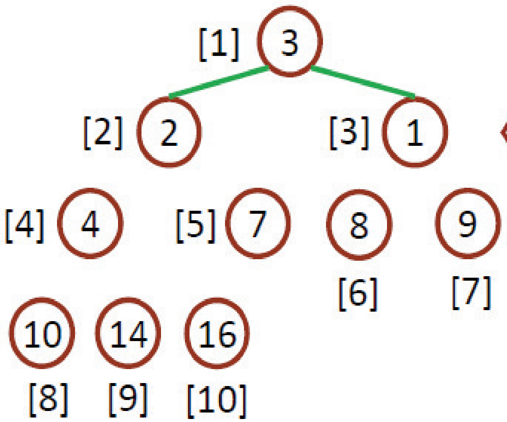
Heap size = 5
Sorted=[8,9,10,14,16]



Heap size = 4
Sorted=[7,8,9,10,14,16]



Heap size = 3
Sorted=[4,7,8,9,10,14,16]



Time complexity Phase 1: construction

$\boxed{d = \lfloor \log n \rfloor : \text{depth}}$

\# of comparisons is at most:

$$\sum_{L=0}^{d-1} 2(d-L)2^L$$

$$=2d \sum_{L=0}^{d-1} 2^L - 4 \sum_{L=0}^{d-1} L2^{L-1}$$

$$\boxed{\left(\sum_{L=0}^{k} L2^{L-1} = 2^k(k-1)+1 \right)}$$

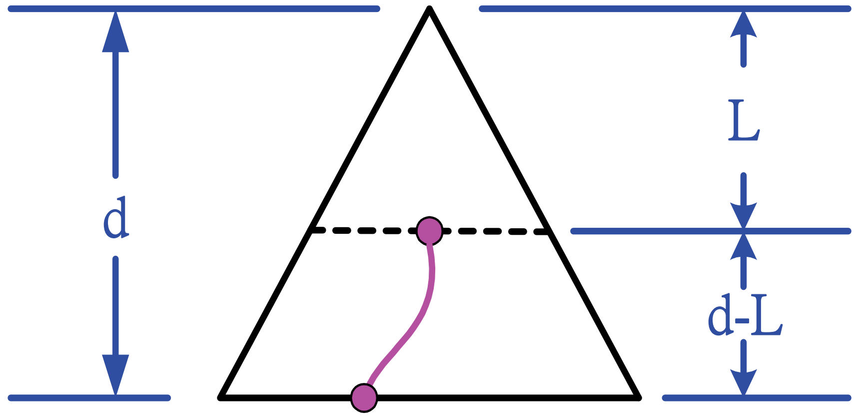
$$=2d(2^d-1) - 4(2^{d-1}(d-1-1)+1)$$

$$\vdots$$

$$= cn - 2\lfloor \log n \rfloor - 4, \quad 2 \le c \le 4$$

Let the level of an internal node be L. The worst case **2(d-L)** comparisons Have to be made to perform the restore.

2^L: number of nodes in level L



Time complexity

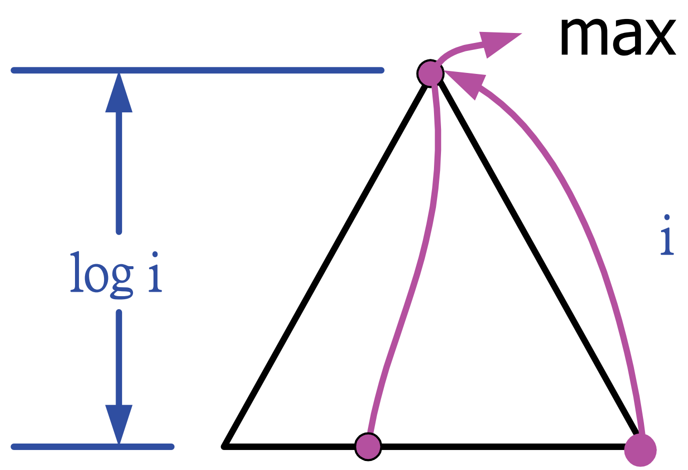
Phase 2: output (delete element from heap)

$$2\sum_{i=1}^{n-1} \lfloor \log i \rfloor$$

$$= \quad \vdots$$

$$= 2n\lfloor \log n \rfloor - 4cn + 4, \quad 2 \le c \le 4$$

$$= O(n \log n)$$



$$2 \sum_{i=1}^{n-1} \lfloor \log i \rfloor .$$

To evaluate this formula, let us consider the case of $n = 10$.

$$\lfloor \log 1 \rfloor = 0$$

$$\lfloor \log 2 \rfloor = \lfloor \log 3 \rfloor = 1$$

$$\lfloor \log 4 \rfloor = \lfloor \log 5 \rfloor = \lfloor \log 6 \rfloor = \lfloor \log 7 \rfloor = 2$$

$$\lfloor \log 8 \rfloor = \lfloor \log 9 \rfloor = 3 .$$

We observe that there are

2^1 numbers equal to $\lfloor \log 2^1 \rfloor = 1$

2^2 numbers equal to $\lfloor \log 2^2 \rfloor = 2$

and $\boxed{10 - 2^{\lfloor \log 10 \rfloor} = 10 - 2^3 = 2 \text{ numbers equal to } \lfloor \log n \rfloor.}$

In general,

$$2\sum_{i=1}^{n-1}\lfloor \log i \rfloor$$

$$= \boxed{2\sum_{i=1}^{\lfloor \log n \rfloor -1} i2^{i}} + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor$$

$$= 4\sum_{i=1}^{\lfloor \log n \rfloor -1} i2^{i-1} + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor.$$

Using $\boxed{\sum_{i=1}^{k} i2^{i-1} = 2^{k}(k-1)+1}$ (Eq. 2.1 in section 2-2)

$$2\sum_{i=1}^{n-1}\lfloor \log i \rfloor$$

$$= 4\sum_{i-1}^{\lfloor \log n \rfloor -1} i2^{i-1} + 2(n - 2^{\lfloor \log n \rfloor})\lfloor \log n \rfloor$$

$$= 4(2^{\lfloor \log n \rfloor -1}(\lfloor \log n \rfloor - 1 - 1) + 1) + 2n\lfloor \log n \rfloor - 2\lfloor \log n \rfloor 2^{\lfloor \log n \rfloor}$$

$$= 2 \cdot 2^{\lfloor \log n \rfloor}\lfloor \log n \rfloor - 8 \cdot 2^{\lfloor \log n \rfloor -1} + 4 + 2n\lfloor \log n \rfloor - 2 \cdot 2^{\lfloor \log n \rfloor}\lfloor \log n \rfloor$$

$$= 2 \cdot n\lfloor \log n \rfloor - 4 \cdot 2^{\lfloor \log n \rfloor} + 4$$

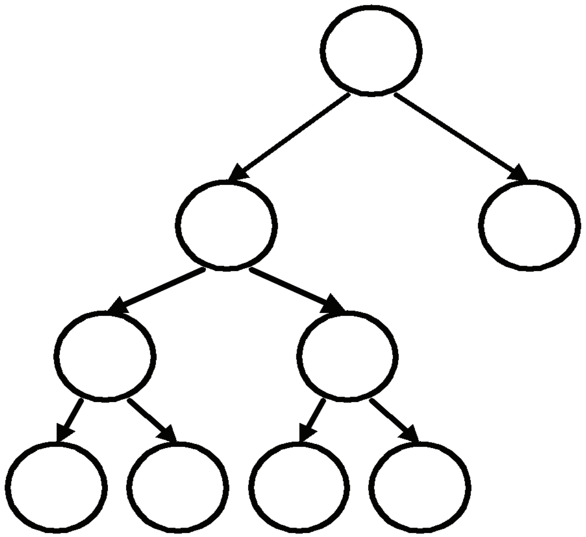
$$= 2n\lfloor \log n \rfloor - 4cn + 4 \qquad \text{where} \quad 2 \le c \le 4$$

$$= O(n \log n).$$

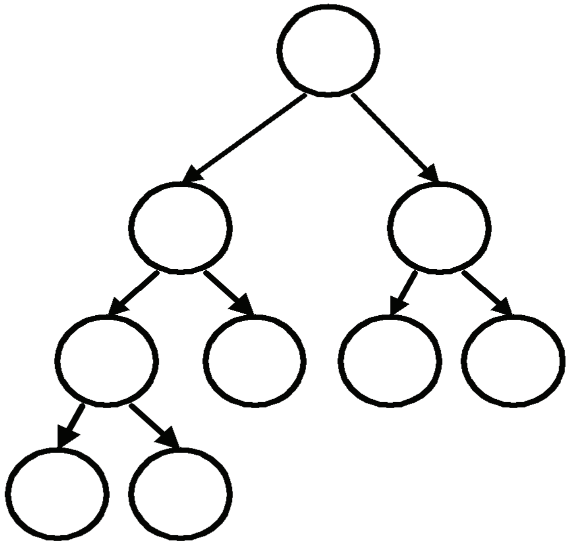
2-6 Average case lower bound of sorting

- By binary decision tree
- The length of this path is equal to the number of comparisons executed for this input data set.
- The average time complexity of a sorting algorithm:
  - the **external path length** of the binary tree is the sum of the lengths of paths from root to each leaf node.
  - Leaf number： n!
- The external path length is minimized if the tree is balanced.

  (all leaf nodes on level d or level d–1)



unbalanced
external path length
$= 4 \cdot 3 + 1 = 13$



balanced
external path length
$= 2 \cdot 3 + 3 \cdot 2 = 12$

Tree Modification

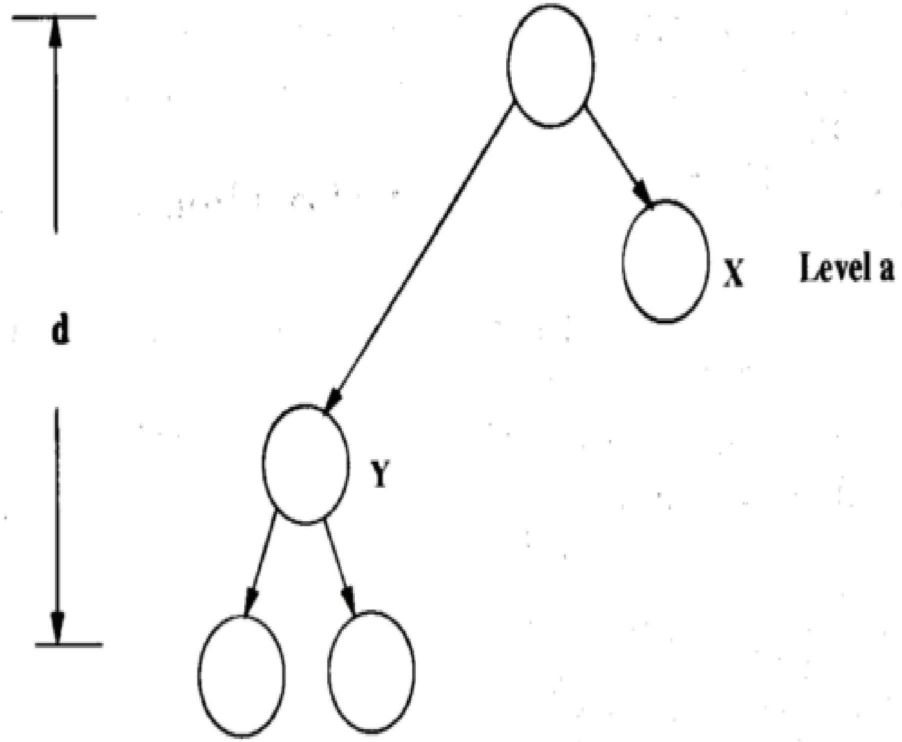


Figure 2-16 An Unbalanced Binary Tree.

- Modify the tree such the external path length is decreased without changing the # of leaf nodes.

The tree can be modified such that the external path length is decreased without changing the number of leaf node.

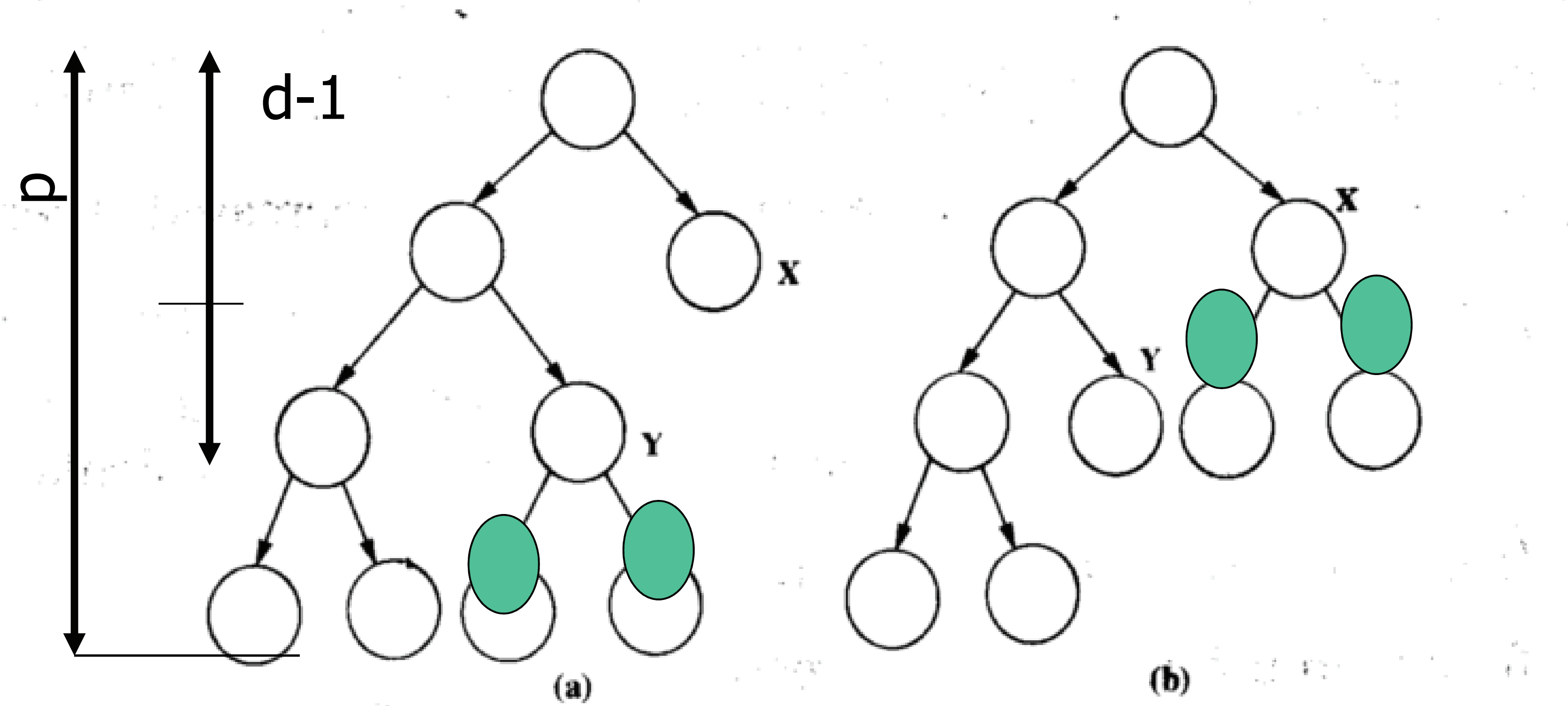


Figure 2-15 The Modification of an Unbalanced Binary Tree.

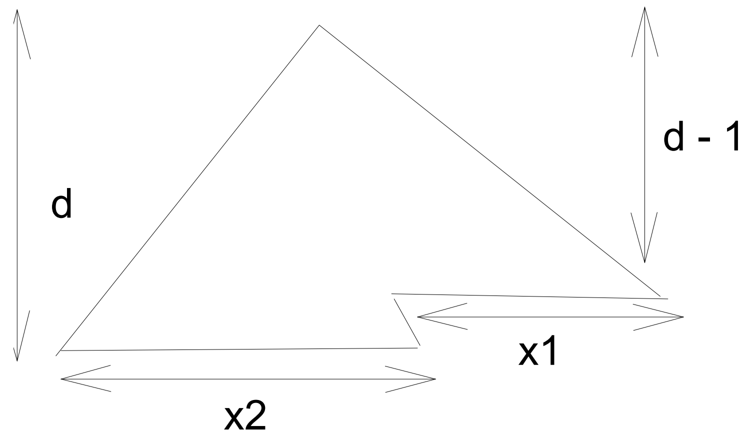
The external path length of a binary tree is minimized if and only if the tree is balanced.

Compute the min external path length

1. Depth of balanced binary tree with c leaf nodes:
   depth d = $\lceil \log c \rceil$
   Leaf nodes can appear only on level d or d–1(balanced).
2. x_1 leaf nodes on level d–1
   x_2 leaf nodes on level d
   - $x_1 + x_2 = c$
   - $x_1 + \frac{x_2}{2} = 2^{d-1}$

$\Rightarrow$ $x_1 = 2^d - c$
$x_2 = 2(c - 2^{d-1})$

- Assume x_2 is even.
- Two leave in level d has a parent in
- level d-1



The external path length of a balanced binary tree is the lower bound of the sorting(in average case).

3. External path length:

$$M = x_1(d-1) + x_2 d$$
$$= (2^d - c)(d-1) + 2(c - 2^{d-1})d$$
$$= c + cd - 2^d, \qquad \log c \le d < \log c + 1$$
$$\ge c + c\log c - 2 * 2^{\log c}$$
$$= \mathbf{c\log c - c}$$

4. $c = n!$

$$M = n!\log n! - n!$$
$$M/n! = \log n! - 1$$
$$= \Omega(n\log n)$$

Average case lower bound of sorting: $\Omega(n\log n)$

Quicksort & Heapsort

- Quicksort is optimal in the average case.

  ($O(n \log n)$ in average)
- (i) worst case time complexity of heapsort is

  $O(n \log n)$

  (ii) average case lower bound: $\Omega(n \log n)$
  - average case time complexity of heapsort is $O(n \log n)$
  - Heapsort is optimal in the average case.

2-7 Improving a lower bound through oracles

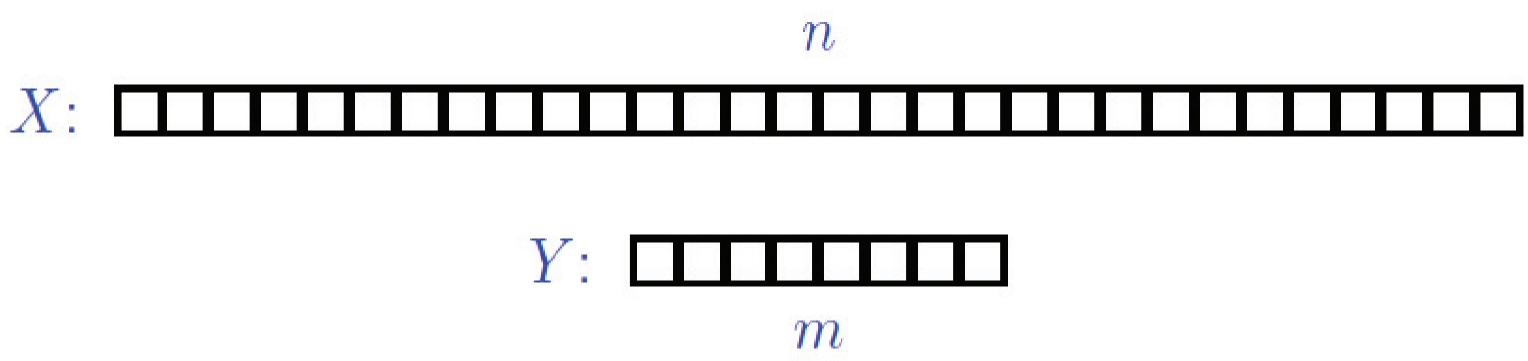
- In some cases, the binary decision tree model does not produce a very meaningful LB. (can be improved)
- Problem P: merge two sorted sequences A and B with lengths m and n.
- Conventional 2-way merging:

  2 3 5 6

  1 4 7 8
- Complexity: at most m+n-1 comparisons

Input: Two sorted lists X and Y of length n and m.

We may assume $n \geq m$.



Standard Merge:

$$\Theta(n + m)$$

Binary Insertion of Y in X:

$$\Theta(m \log n))$$

For "large" m ($m = \Theta(n)$):

$$\Theta(n+m) = \Theta(m(\log(n/m)+1))$$

For "small" m (e.g. $m = O(\sqrt{n})$):

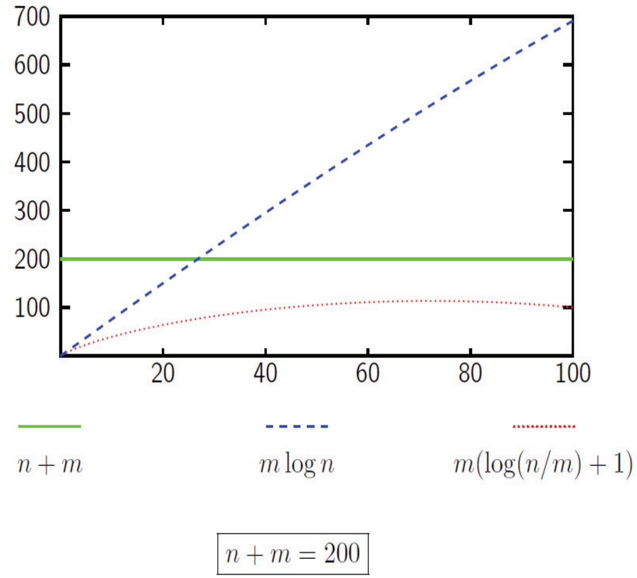
elements from b are evenly spread along a each insertion will take O(log (n/m)) and the overall complexity will be O(m log(n/m)).

$$\Theta(m\log n) = \Theta(m(\log(n/m)+1))$$

E.g. for $m = \Theta(n/\log n)$:

$$\Theta(n+m) = \Theta(n)$$

$$\Theta(m\log n) = \Theta(n)$$



$$\Theta(m(\log(n/m)+1)) = \Theta\left(n\frac{\log\log n}{\log n}\right) = o(n)$$

(1) Binary decision tree:

- **How many possible different merged sequence are there?**
- **Assume (m+n) elements are distinct.**

  There are $\binom{m+n}{n}$ ways to merge n elements into m elements without disturbing the original order. (why?)

$\binom{m+n}{n}$ leaf nodes in the decision tree.

$\Rightarrow$ The lower bound for merging:

$$\left\lceil \log \binom{m+n}{n} \right\rceil \le m + n - 1$$

(conventional merging)

- When m = n

$$\log\binom{m+n}{n} = \log\frac{(2m)!}{(m!)^2} = \log((2m)!) - 2\log m!$$

Using Stirling approximation

$$n! \approx \sqrt{2\pi n}\,(\frac{n}{e})^n$$

$$\log\binom{m+n}{n} \approx (\log\sqrt{2\pi} + \log\sqrt{2m} + 2m\log\frac{2m}{e}) -$$

$$-2\,(\log\sqrt{2\pi} + \log\sqrt{m} + m\log\frac{m}{e})$$

$$\approx 2m - \frac{1}{2}\log m + O(1) \;<\; 2m-1$$

- Optimal algorithm: conventional merging needs 2m-1 comparisons

(2) Oracle (聖賢;哲人):

- The oracle tries its best to cause the algorithm to work as hard as it might. (to give a very hard data set)->to find worst case.
- Two sorted sequences:
  - A: $a_1 < a_2 < \ldots < a_m$
  - B: $b_1 < b_2 < \ldots < b_m$
- The very hard case:
  - **$a_1 < b_1 < a_2 < b_2 < \ldots < a_m < b_m$**

- We must compare:

$$a_1 : b_1$$
$$b_1 : a_2$$
$$a_2 : b_2$$
$$:$$
$$b_{m-1} : a_{m-1}$$
$$a_m : b_m$$

- Otherwise, we may get a wrong result for some input data.

  e.g. If b_1 and a_2 are not compared, we can not distinguish

  $a_1 < b_1 < a_2 < b_2 < \ldots < a_m < b_m$ and

  $a_1 < a_2 < b_1 < b_2 < \ldots < a_m < b_m$
- Thus, at least $2m-1$ comparisons are required.
- The conventional merging algorithm is optimal for $m = n$.

Finding lower bound by problem transformation

Problem A reduces to problem B ($A \propto B$)

iff A can be solved by using any algorithm which solves B.

If $A \propto B$, B is more difficult.

$$
\begin{tikzcd}
\text{instance of A} \arrow[r, "\text{transformation}", "T(tr_1)"'] \arrow[d, "T(A)"'] & \text{instance of B} \arrow[d, "\text{solver of B}", "T(B)"'] \\
\text{answer of A} & \text{answer of B} \arrow[l, "\text{transformation}"', "T(tr_2)"]
\end{tikzcd}
$$

Note: $T(tr1) + T(tr2) < T(B)$

$T(A) \le T(tr1) + T(tr2) + T(B) \sim O(T(B))$

- **Problem Convex Hull(S)**
  - Input: S is a sequence of points (x_i,y_i) in the plane.
  - Output: permute S and return k such that $S_1, \ldots S_k$ is the convex hull of S.

- The reduction of Sorting problem to Convex Hull problem:
  - Reduction **sortByConvexHull(S)**
  - {// S is a sequence of numbers.
    - 1. for i in 1..n, set P[i] = point(S[i], $S[i]^2$);

    /* in other words, set P = { (x, x^2) | x in S } */
    - 2. k = **convexHull(P)**;

    /* We know in advance that k will be size(P).*/
    - 3. for i in 1..n, set S[i] = **P[i].first**;

    /* first = the x of a (x, x^2) pair. */
  - }

2.8 The lower bound of the convex hull problem

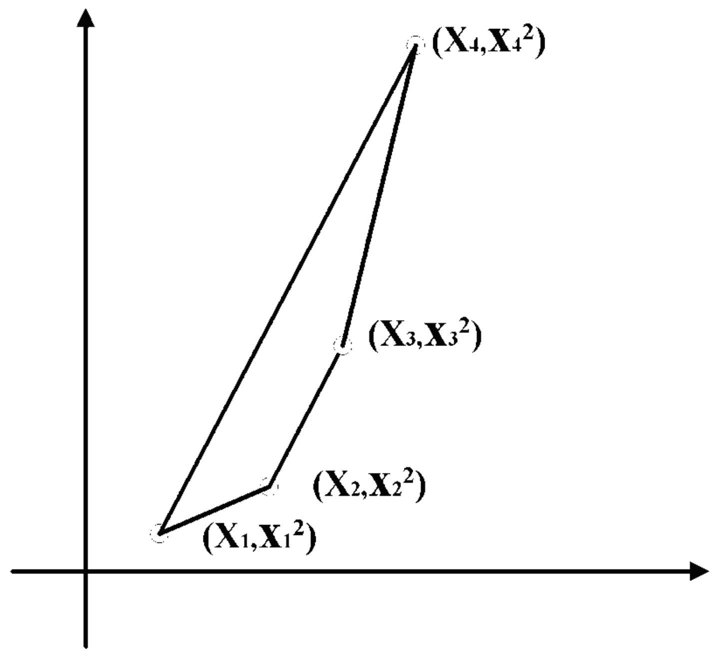
- sorting $\propto$ convex hull

  A B
- an instance of A: $(x_1, x_2, \ldots, x_n)$

  $\downarrow$transformation

  an instance of B: $\{(x_1, x_1^2), (x_2, x_2^2), \ldots, (x_n, x_n^2)\}$

  assume: $x_1 < x_2 < \ldots < x_n$



Solve A by transform A to B, and solve B, the result of B can be Easily transformed to the solution of A.

- If the convex hull problem can be solved, we can also solve the sorting problem.
  - The lower bound of sorting: $\Omega(n \log n)$
- The lower bound of the convex hull problem: $\Omega(n \log n)$

The lower bound of the Euclidean minimal spanning tree (MST) problem

- sorting $\propto$ Euclidean MST

  A B

- an instance of A: $(x_1, x_2, \ldots, x_n)$

  $\downarrow$transformation

  an instance of B: $\{(x_1, 0), (x_2, 0), \ldots, (x_n, 0)\}$

  - Assume $x_1 < x_2 < x_3 < \ldots < x_n$
  - $\Leftrightarrow$there is an edge between $(x_i, 0)$ and $(x_{i+1}, 0)$ in the MST, where $1 \leq i \leq n-1$

- If the Euclidean MST problem can be solved, we can also solve the sorting problem.
  - The lower bound of sorting: $\Omega(n \log n)$
- The lower bound of the Euclidean MST problem: $\Omega(n \log n)$

Sorting In Linear Time

- Counting sort
  - No comparisons between elements!
  - ***But***...depends on assumption about the numbers being sorted
    - We assume numbers are in the range ***1.. k***
  - The algorithm:
    - Input: A[1..n], where A[j] $\in$ $\{1, 2, 3, \ldots, k\}$
    - Output: B[1..n], sorted (notice: not sorting in place)
    - Also: Array C[1..k] for auxiliary storage

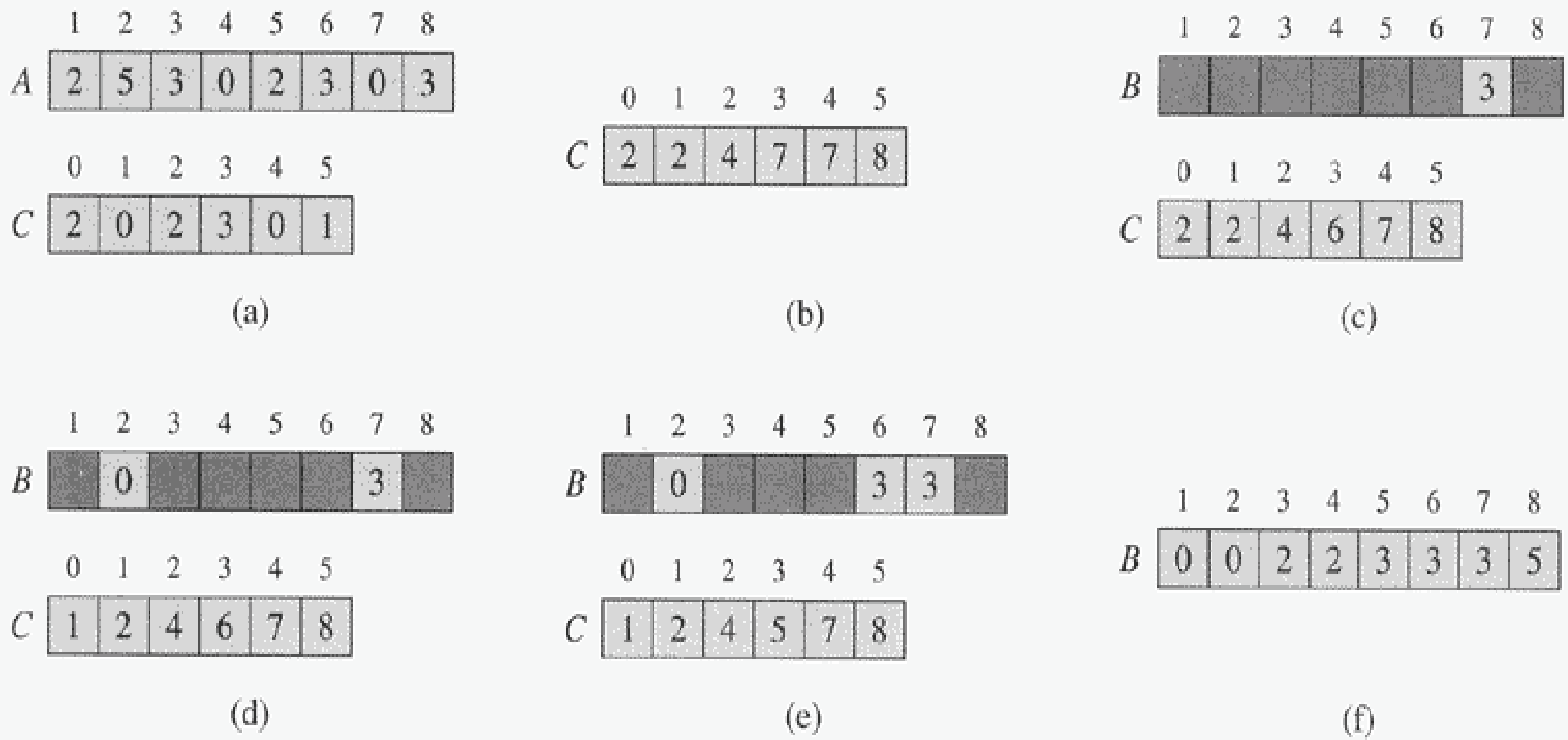


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1\,..\,8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 4. **(b)** The array C after line 7. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 9–11, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

Counting Sort

```
CountingSort(A, B, k)
        for i=1 to k
                C[i]= 0;
        for j=1 to n
                C[A[j]] += 1;
        for i=2 to k
                C[i] = C[i] + C[i-1];
        for j=n downto 1
                B[C[A[j]]] = A[j];
                C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

What will be the running time?

Counting Sort

- Total time: O(n + k)
  - Usually, k = O(n)
  - Thus counting sort runs in O(n) time
- But sorting is Ω(n log n)!
  - No contradiction--this is not a comparison sort (in fact, there are *no* comparisons at all!)
  - Notice that this algorithm is *stable*

穩定排序法(stable sorting)，如果鍵值相同之資料，在排序後相對位置與排序前**相同**時，稱**穩定排序**。

【例如】

排序前：3,5,19,1,3*,10

排序後：1,3,3*,5,10,19

(因為兩個3, 3*的相對位置在排序前與後皆相同。)

Counting Sort

- Cool! *Why don't we always use counting sort?*
- Because it depends on range k of elements
- *Could we use counting sort to sort 32 bit integers? Why or why not?*
- Answer: no, k too large ($2^{32} = 4{,}294{,}967{,}296$)

Counting Sort

- *How did IBM get rich originally?*
- Answer: punched card readers for census tabulation in early 1900's.
  - In particular, a *card sorter* that could sort cards into different bins
    - Each column can be punched in 12 places
    - Decimal digits use 10 places
  - Problem: only one column can be sorted on at a time

Radix sort

329		720		720		329
457		355		329		355
657		436		436		436
839	→	457	→	839	→	457
436		657		355		657
720		329		457		720
355		839		657		839

Radix Sort

- Intuitively, you might sort on the most significant digit (MSD) , then the second MSD, etc.
- Problem: lots of intermediate piles of cards (read: scratch arrays) to keep track of
- Key idea: sort the *least* significant digit first

  RadixSort(A, d)

  for i=1 to d

  StableSort(A) on digit i

  - Example: Fig 9.3

Radix Sort

- *Can we prove it will work?*
- Sketch of an inductive argument (induction on the number of passes):
  - Assume lower-order digits $\{j: j<i\}$ are sorted
  - Show that sorting next digit i leaves array correctly sorted
    - If two digits at position i are different, ordering numbers by that digit is correct (lower-order digits irrelevant)
    - If they are the same, numbers are already sorted on the lower-order digits. Since we use a stable sort, the numbers stay in the right order

Radix Sort

- *What sort will we use to sort on digits?*
- Counting sort is obvious choice:
  - Sort n numbers on digits that range from $1..k$
  - Time: $O(n + k)$
- Each pass over n numbers with d digits takes time $O(n+k)$, so total time $O(dn+dk)$
  - When d is constant and $k=O(n)$, takes $O(n)$ time
- *How many bits in a computer word?*

Radix Sort

- Problem: sort 1 million 64-bit numbers
  - Treat as four-digit radix 2^{16} numbers
  - Can sort in just four passes with radix sort!
- Compares well with typical $O(n \log n)$ comparison sort
  - Requires approx. $\log n = 20$ operations per number being sorted
- *So why would we ever use anything but radix sort?*

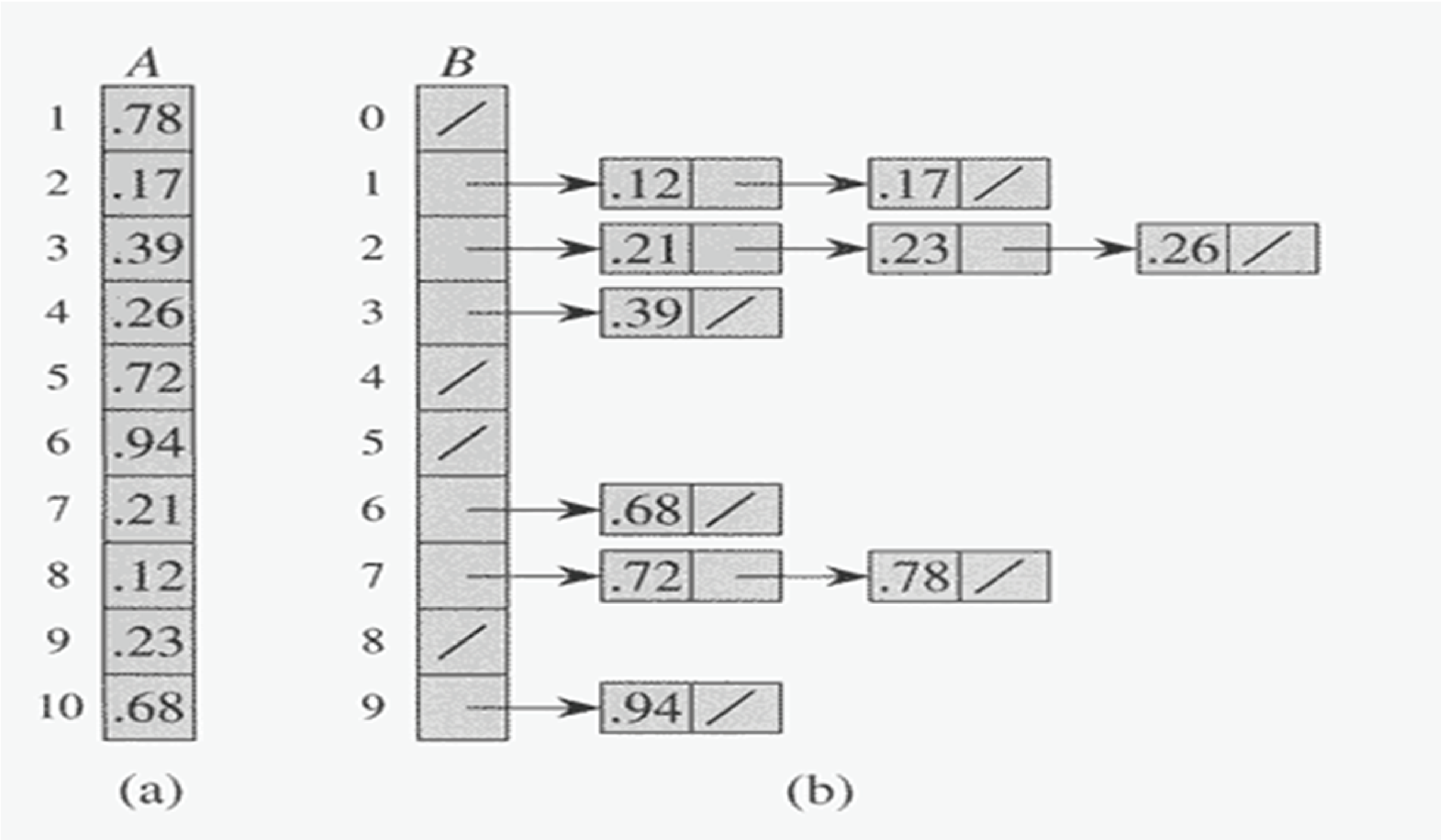
Radix Sort

- In general, radix sort based on counting sort is
  - Fast
  - Asymptotically fast (i.e., $O(n)$)
  - Simple to code
  - A good choice
- To think about: *Can radix sort be used on floating-point numbers?*

Bucket Sort

- Bucket sort
  - Assumption: input is n reals from [0, 1)
  - Basic idea:
    - Create n linked lists (*buckets*) to divide interval [0,1) into subintervals of size $1/n$
    - Add each input element to appropriate bucket and sort buckets with insertion sort
  - Uniform input distribution → O(1) bucket size
    - Therefore the expected total time is O(n)
  - These ideas will return when we study *hash tables*

Bucket Sort



A
1
.78
2
.17
3
.39
4
.26
5
.72
6
.94
7
.21
8
.12
9
.23
10
.68
(a)
B
0
1
2
3
4
5
6
7
8
9
.12
.17
.21
.23
.26
.39
.68
.72
.78
.94
(b)

Bucket Sort

```
BUCKET-SORT(A)
n ← length[A]
for i ← 1 to n
        do insert A[i] into list B[⌊n A[i]⌋]
for i ← 0 to n − 1
        do sort list B[i] with insertion sort
concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```