Sorting

Sorting

- List: a collection of records
  - Each record has one or more fields.
  - Key: used to distinguish among the records.

	Key	Other fields
Record 1	4	DDD
Record 2	2	BBB
Record 3	1	AAA
Record 4	5	EEE
Record 5	3	CCC

original list

Key	Other fields
1	AAA
2	BBB
3	CCC
4	DDD
5	EEE

sorted list

Motivation of Sorting

- **Sequential search**

  44, 55, 12, 42, 94, 18, 06, 67
- unsuccessful search
  - n
- successful search
  - $\sum_{i=0}^{n-1}(i+1)/n = \frac{n+1}{2}$

Code for Sequential Search

```
template <class E, class K>
int SeqSearch (E *a, const int n, const K& k)
{// Search a[1:n] from left to right. Return least i such that the key of a[i] equals k
 // If ther is no such i , return 0
      int i;
      for (i = 1 ; i <= n && a[i] != k; i++ );
      if (i > n) return 0;

      return i;

}
```

Motivation of Sorting

- A binary search needs $O(\log n)$ time to search a key in a sorted list with *n* records.

```
int BinSearch(T *list , const int length, T num)
{
   int left = 0, right = length - 1;
   while (left <= right){
      int middle = (right + left ) / 2;
      if (list [middle] == num)
         return middle;
      if (list y[middle] > num)
         right = middle - 1;
      else
         left = middle + 1;
   }
   return -1;

}
```

Motivation of Sorting

- **Verification problem**: To check if two lists are equal.
  - 6 3 7 9 5
  - 7 6 5 3 9
- Sequential searching method: $O(mn)$ time, where m and n are the lengths of the two lists.
- Compare after sort: $O(\max\{n \log n, m \log m\})$
  - After sort: 3 5 6 7 9 and 3 5 6 7 9
  - Then, compare one by one.

Categories of Sorting Methods

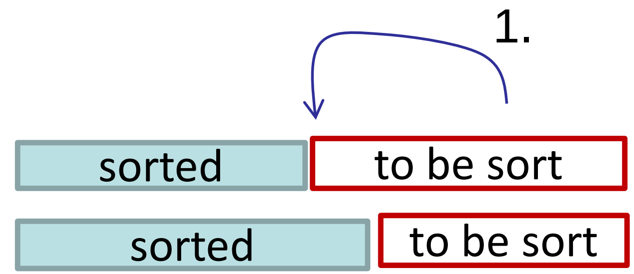
- **Stable Sorting** : the records with the same key have the same relative order as they have before sorting.
  - Example: Before sorting 6 3 7_a 9 5 7_b
  - After sorting 3 5 6 7_a 7_b 9

- **internal sorting**: All data are stored in main memory (more than 20 algorithms).
- **external sorting**: Some of data are stored in auxiliary storage.

Selection Sort

In each iteration

1. find the minimum between item i and n
2. replace the minimum with item i



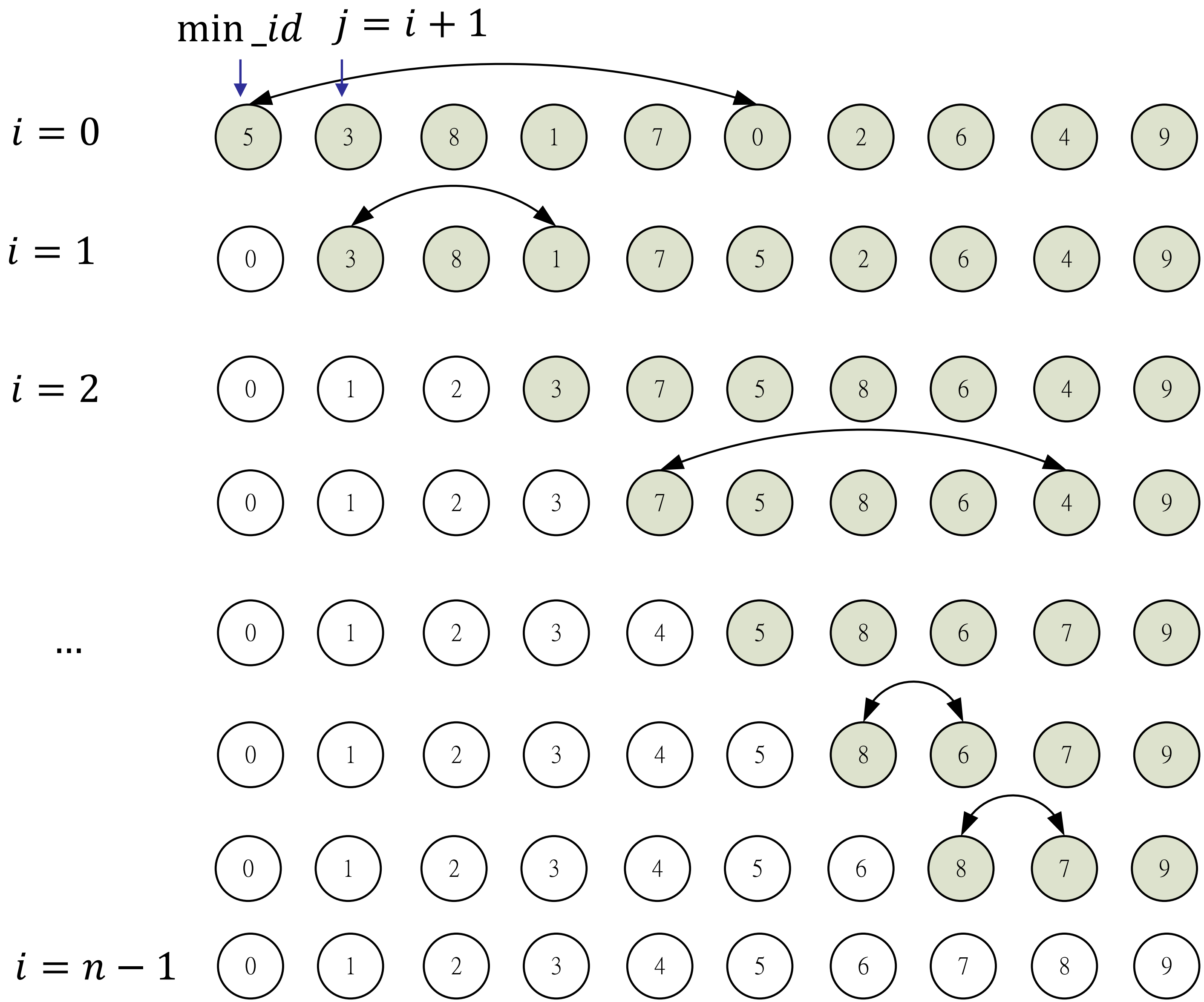

```c
int selection_sort(int list[], int n){
  int i, j, min_id;
  for(i = 0; i<n-1; i++){
    min_id = i;
    for(j=i+1; j<n; j++){
      if (list[j] < list[min_id])
        min_id = j;
    }
    swap(&list[i], &list[min_id]); //送地址過去swap function
  }
};
```

1.

```c
void swap(int *a, int *b){
  int temp;
  temp = *a;
  *a = *b;
  *b= temp;
}
```

- ✓ 需要 n-1 個 pass (run)
- ✓ Complexity:

Selection Sort



Insertion Sort

- 方法: 每次處理一個新的資料時，由右而左insert至其適當的位置才停止。
- 需要 n-1 個 pass
- best case: 未 sort 前，已按順序排好。每個 pass 僅需一次比較, 共需 (n-1) 次比較
- worst case: 未 sort 前, 按相反順序排好。比較次數為：

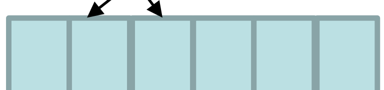
$$1+2+3+\cdots+(n-1)=\frac{n(n-1)}{2}=O(n^2)$$

- Time complexity: O(n^2)

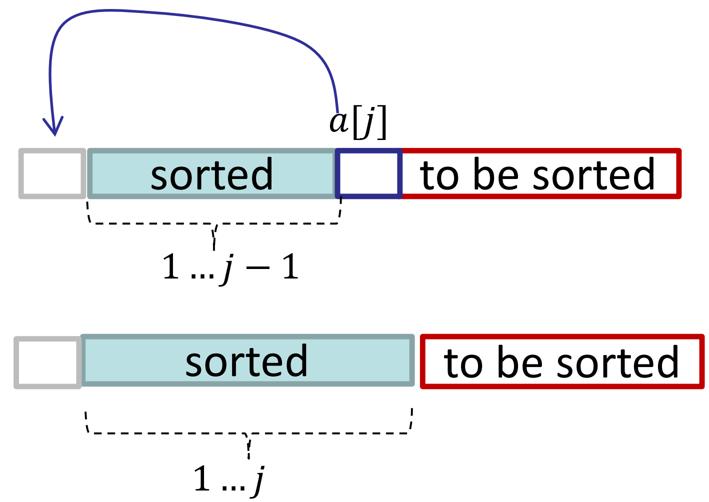
Insertion Sort

```
template <class T>
void InsertionSort(T* a, const int n)
// Sort a[1:n] into nondecreasing order
{
    for (int j = 2; j <= n; j++)
    {
        T temp = a[j];
        Insert(temp, a, j-1);
    }
}
```

a



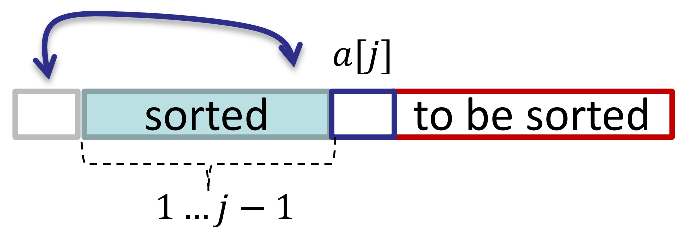
a[0] 當臨時空間用, e=a[j]



Insertion into a Sorted List

```
Insert(temp, a, j-1);
```

```
template <class T>
void Insert ( const T& e, T* a, int i )
// Insert e into the nondecreasing sequence a[1], ..., a[i] such that the resulting
    sequence is also ordered. Array a must have space allocated for at least i+2
    elements
{
  a[0] = e; // Avoid a test for end of list (i<1)
  while (e < a[i])
  {
      a[i+1] = a[i];   //shift right one position
      i--;
  }
  a[i+1] = e;
}
```

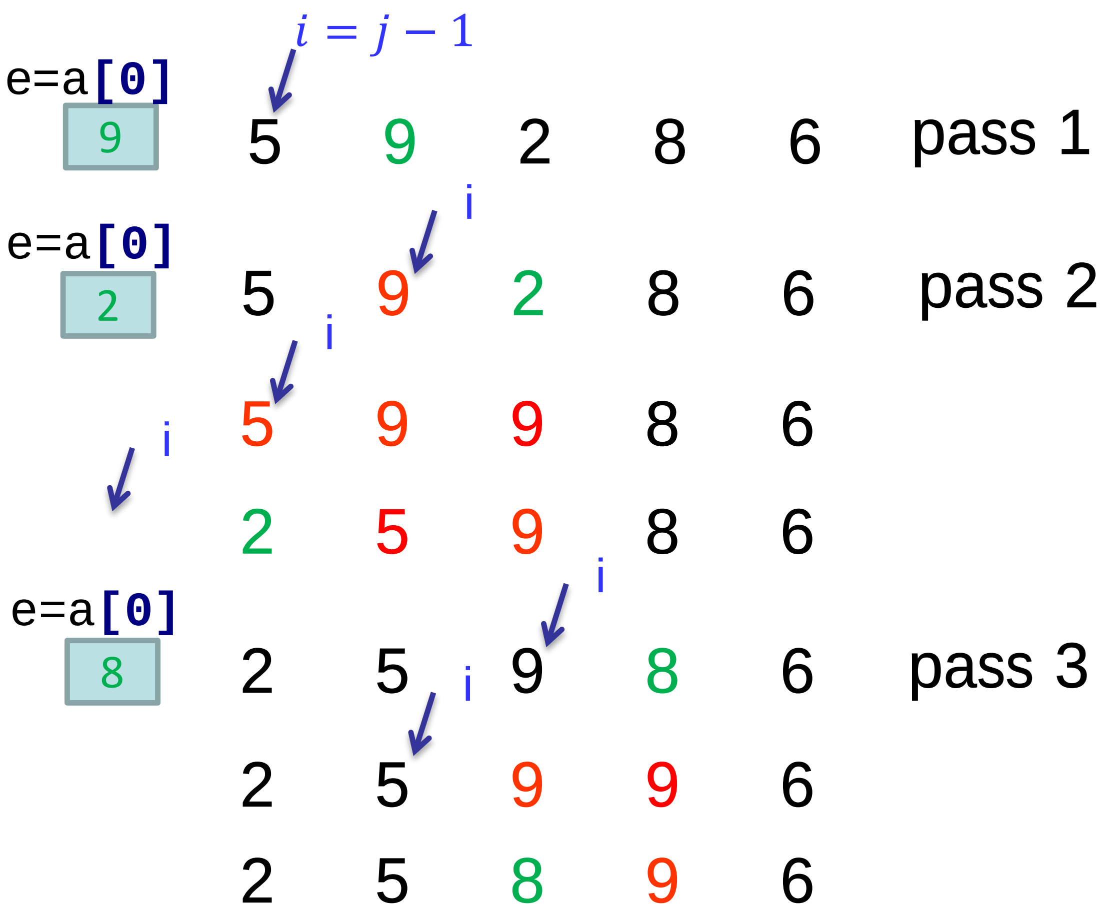


小到大排序

確保從1~i 之element，比e大的都在e的右邊

Insertion Sort

e.g. (nondecreasing order)由小而大 sort




```
while (e < a[i])
{
    a[i+1] = a[i];
    i--;
}
a[i+1] = e;
```

Insertion Sort

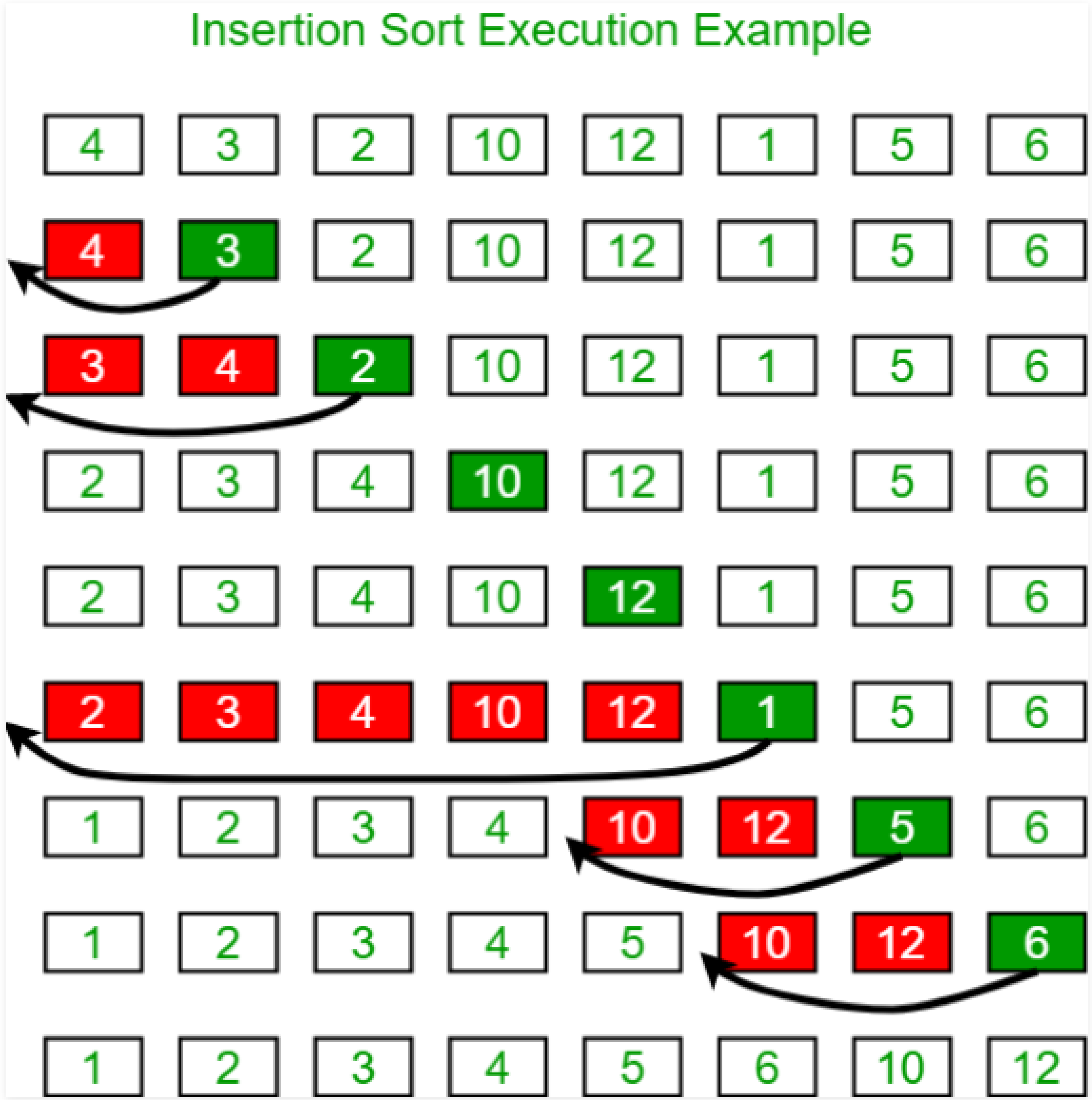
e.g. (nondecreasing order)由小而大 sort

e=a[0]

$\boxed{6}$

2	5	8	9 (i)	6	pass 4
2	5	8 (i)	9	9	
2	5 (i)	8	8	9	
2	5	6	8	9	

```
while (e < a[i])
{
    a[i+1] = a[i];
    i--;
}
a[i+1] = e;
```


Insertion Sort Execution Example
4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12

Quick Sort

- Quick sort 方法： 以每組的第一個資料為基準(pivot)，把比它小的資料放在左邊， 比它大的資料放在右邊，之後以pivot中心， 將這組資料分成兩部份。然後，兩部分資料各自recursively執行相同方法。

- 平均而言，Quick sort 有很好效能。

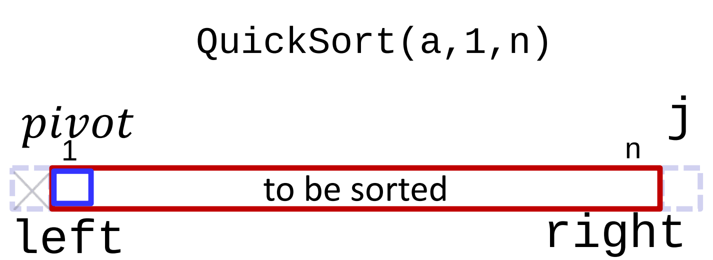
Code for Quick Sort

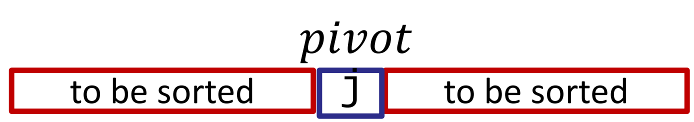
```
void QuickSort(Element* a, const int left, const int right)
// Sort a[left:right] into nondecreasing order.
// Key pivot = a[left].
// i and j are used to partition the subarray so that
// at any time a[m]<= pivot, m < i, and a[m]>= pivot, m > j.
// It is assumed that a[left] <= a[right+1].
{
   if (left < right) {
      int i = left, j = right + 1, pivot = a[left];
      do {
        do i++; while (a[i] < pivot);
        do j--; while (a[j] > pivot);
        if (i<j) swap(a[i], a[j]);
      } while (i < j);

      swap(a[left], a[j]);

      QuickSort(a, left, j-1);
      QuickSort(a, j+1, right);
   }
}
```

$a[i] \geq pivot$
$a[j] \leq pivot$





```
do i++; while(a[i]<pivot);
do j--; while(a[j]>pivot);
  i = left
  j = right + 1
```

QuickSort(a,1,n=10)

pivot j

to be sorted

left right

Quick Sort

pivot

- Input: 26, 5, 37, 1, 61, 11, 59, 15, 48, 19

i = 1
j = 11
i = 3
j = 10
i = 5

QuickSort(a,1,n=5)

i = 1
j = 6

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	Left	Right
[26	5	i = 3 37	1	61	11	59	15	48	j = 10 19]	1	10
[26	5	19	1	i = 5 61	11	59	j = 8 15	48	37]	1	10
[26	5	19	1	15	j = 6 11	i = 7 59	61	48	37]	1	10
[11	5	i = 3 19	j = 4 1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37]	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Time Complexity of Quick Sort

$pivot$每次都落在最左邊

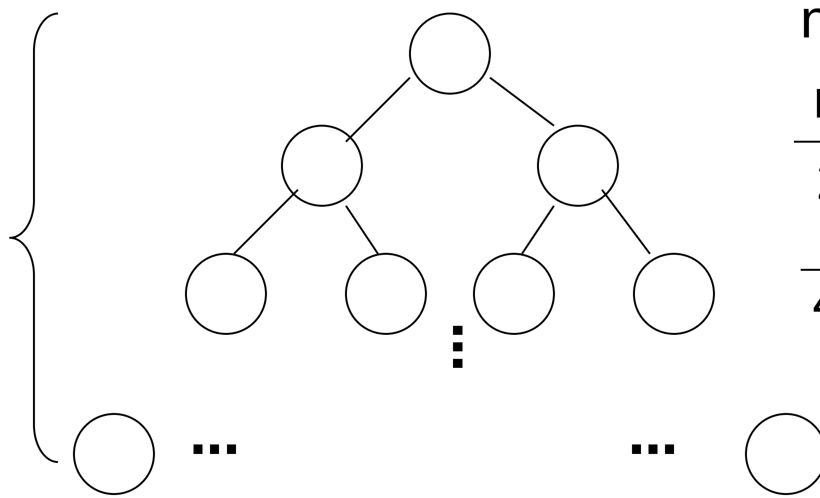
- <u>Worst case time complexity</u>: 每次的基準恰為最大，或最小。所需比較次數：

$$(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}=\mathrm{O}(n^2)$$

$pivot$每次都落在中間

- <u>Best case time complexity</u>：$\mathrm{O}(nlogn)$
  - 每次分割(partition)時, 都分成大約相同數量的兩部份 。

$\log_2 n$



n

$\frac{n}{2} \times 2 = n$

$\frac{n}{4} \times 4 = n$

Mathematical Analysis of Best Case

- $T(n)$: Time required for sorting n data elements.

$$
\begin{aligned}
T(1) &= b, \text{ for some constant } b \\
T(n) &\le cn + 2T(n/2), \text{ for some constant } c \\
&\le cn + 2(cn/2 + 2T(n/4)) \\
&\le 2cn + 4T(n/4) \\
&\quad \vdots \\
&\quad \vdots \\
&\le cn \log_2 n + T(1) \\
&= O(n \log n)
\end{aligned}
$$

Variations of Quick Sort

- Quick sort using a median of three:
  - Pick the median of the first, middle, and last keys in the current sublist as the pivot. Thus, $pivot$ = median $\{K_l, K_{\frac{l+n}{2}}, K_n\}$.

- Use the selection algorithm to get the real median element.
  - Time complexity of the selection algorithm: $O(n)$.

Two-way Merge

- Merge two sorted sequences into a single one.

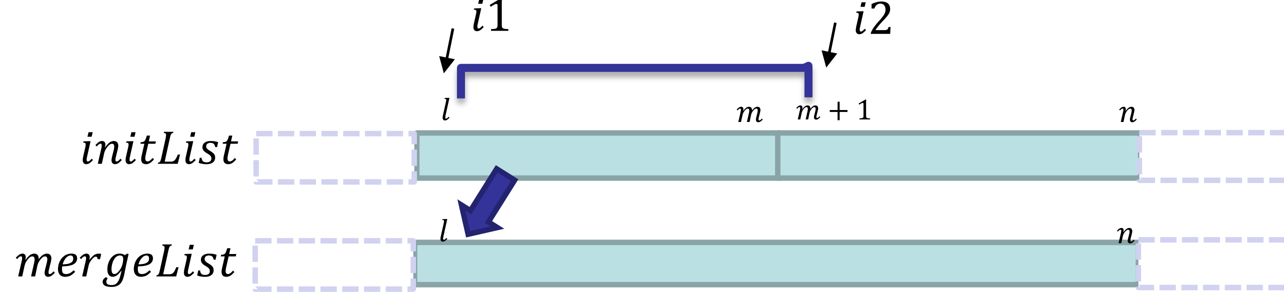
[1 5 26 77] [11 15 59 61]

⇩ merge

[1 5 11 15 26 59 61 77]

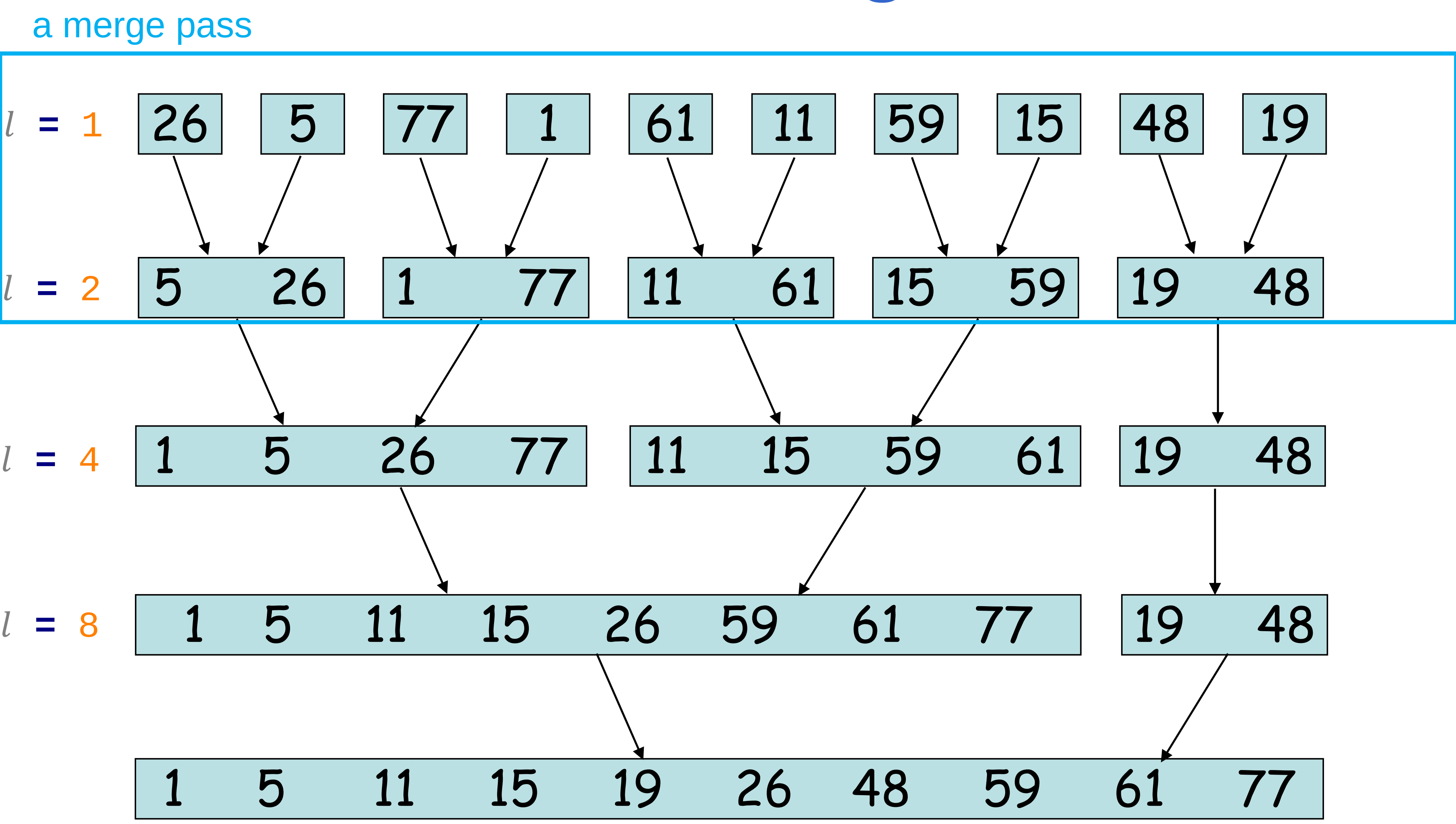
- 設兩個 sorted lists 長度各為 *m*, *n*

  Time complexity: $O(m+n)$



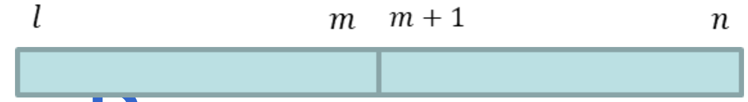

```
template <class T>
void Merge(T *initList, T *mergedList, const int l, const int m, const int n)
{ // initList [l:m] 與 initList [m + 1:n] 是排序好的串列。
  // 我們將它們合併成排序好的串列 mergedList [l:n]。
    for (int i1 = l, iResult = l, i2 = m + 1; // i1, i2, 與 iResult 是串列位置
        i1 <= m && i2 <= n; // 兩個輸入串列都還沒用盡
        iResult++)
        if (initList[i1] <= initList [i2]) {
            mergedList [iResult] = initList [i1];
            i1++;
        }
        else {
            mergedList [iResult] = initList [i2];
            i2++;
        }
    // 如果第一個串列有剩下的記錄，那麼把它複製完
    copy (initList + i1, initList + m + 1, mergedList + iResult);
    // 如果第二個串列有剩下的記錄，那麼把它複製完
    copy (initList + i2, initList + n + 1, mergedList + iResult);
}
```

Iterative Merge Sort



Iterative Merge Sort

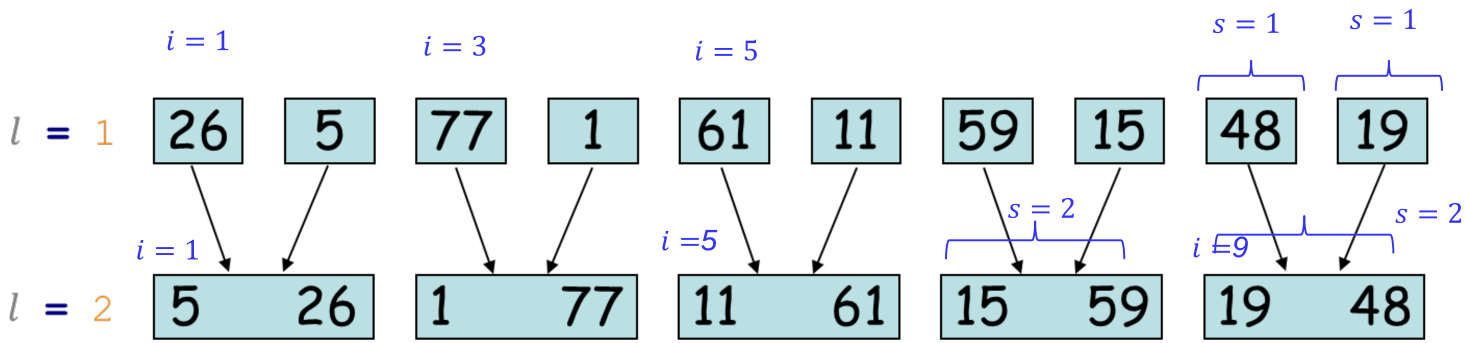
```
template <class T>
void MergeSort(T *a,const int n)
{// 將陣列 a[1:n] 排序成非遞減順序
    T *tempList = new T[n+1];
    // l 是目前合併中的子串列之長度
    for (int l =1; 1 < n; l*= 2)
    {
        MergePass(a, tempList, n, l);
        l*=2;
        MergePass(tempList, a, n, l); // 交換 a 與 tempList 的角色
    }
    delete [] tempList;
}
```



Code for Merge Pass

總長度　小節長度

```
template <class T>
void MergePass(T *initList, T *resultList, const int n, const int s)
{// 將大小為 s 的相鄰子串列對從 initList 合併至 resultList。
 // n 是 initList 裡的記錄個數。
    for (int i = 1; // i 是第一個合併中的子串列的第一個位置
         i <= n-2*s+1; // 元素足夠給兩個長度為 s 的子串列用？
         i+ = 2*s)
                                      l      m          n
           Merge(initList, resultList, i, i + s -1, i + 2 * s -1);
    // 合併其餘大小 < 2 * s 的串列
    if ((i + s -1) < n ) Merge(initList, resultList, i, i + s -1, n);
    else copy(initList + i, initList + n + 1, resultList + i);
}
```



Analysis of Merge Sort

- Merge sort is a <u>stable sorting</u> method.
- Time complexity: $O(n \log n)$
  - $\lceil \log_2 n \rceil$ passes are needed.
  - Each pass takes $O(n)$ time.

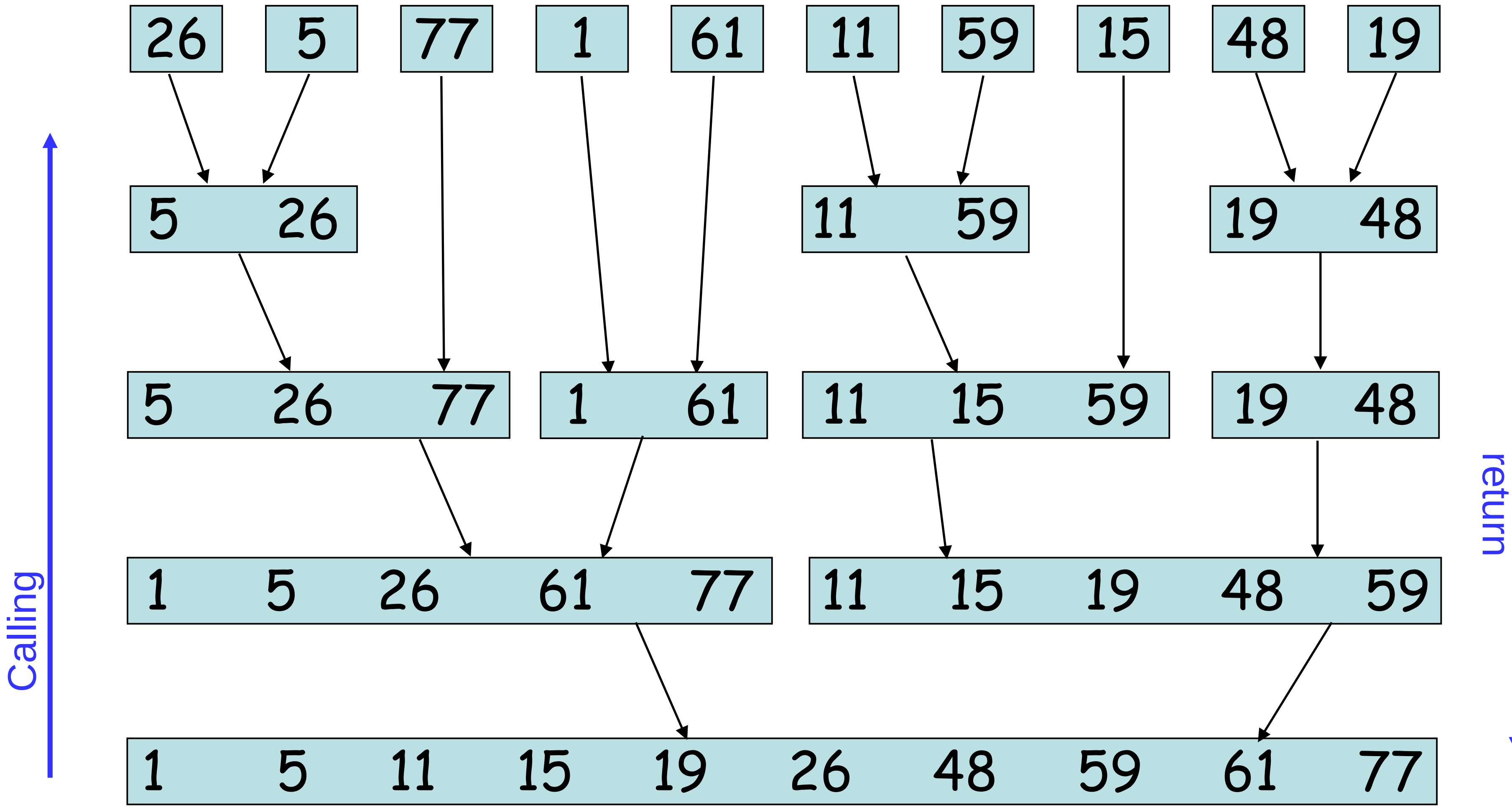
Two way Merger sort: O(m+n)

Recursive Merge Sort

- Dividing the list to be sorted into two roughly equal parts:
  - left sublist $[left : \left\lfloor \frac{left+right}{2} \right\rfloor]$
  - right sublist $[\left\lfloor \frac{left+right}{2} \right\rfloor + 1 : right]$
- These two sublists are sorted recursively.
- Then, the two sorted sublists are merged.

➢ To eliminate the record copying, we associate an integer pointer (instead of real link) with each record.

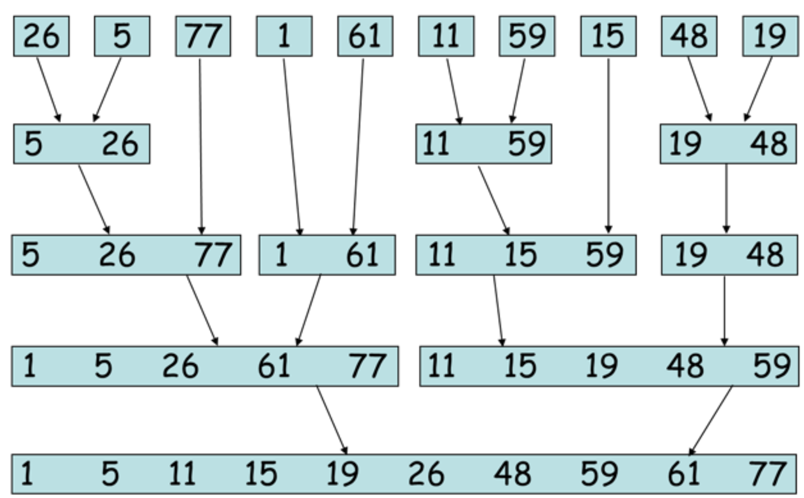
Recursive Merge Sort



Recursive Merge Sort

```
template <class T>
int rMergeSort(T* a, int* link, const int left, const int right)
{// 要排序的是 a[left:right]。對於所有 i，link[i] 初始化為 0。
 // rMerge 回傳排序好的鏈的第一個元素之索引值。
    if (left >= right) return left;
    int mid = (left + right) /2;
    return ListMerge(a, link,
                  rMergeSort(a, link, left, mid),        // 排序左半邊
                  rMergeSort(a, link, mid + 1, right));  // 排序右半邊
}
```

1. rMergeSort 左半
2. rMergeSort 右半
3. ListMerge

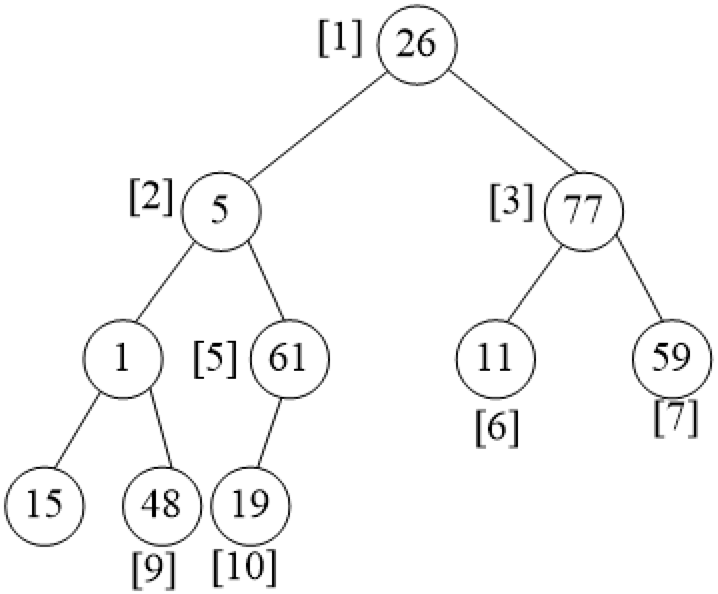


Recursive Merge Sort

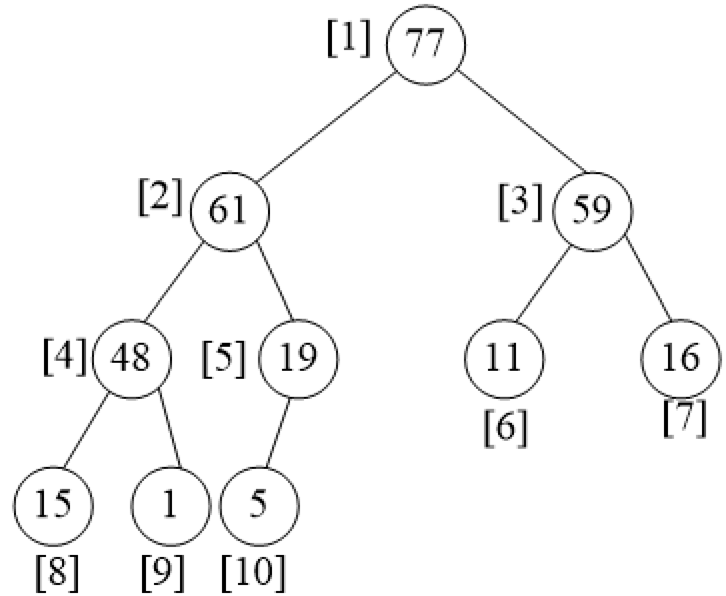
```
tamplate <class T>
int ListMerge(T* a, int* link, const int start1, const int start2)
{// 兩個排序好的鏈分別從 start1 及 start2 開始，將它們合併
 // 將 link[0]當作一個暫時的標頭。回傳合併好的鏈的開頭。
    int iResult = 0; // 結果鏈的最後一筆記錄
    for (int i1 = start1, i2 =start2; i1 && i2; )
        if (a[i1] <= a[i2]) {
            link[iResult] = i1;
            iResult = i1; i1 = link[i1];
        }
        else {
        link[iResult] = i2;
        iResult = i2; i2 =link[i2];
        }
    // 將其餘的記錄附接至結果鏈
    if (i1 = = 0) link[iResult] = i2;
    else link[iResult] = i1;
    return link[0];
}
```

iResult: interger array
link[i] 表示i的下一個

Heap Sort (1)

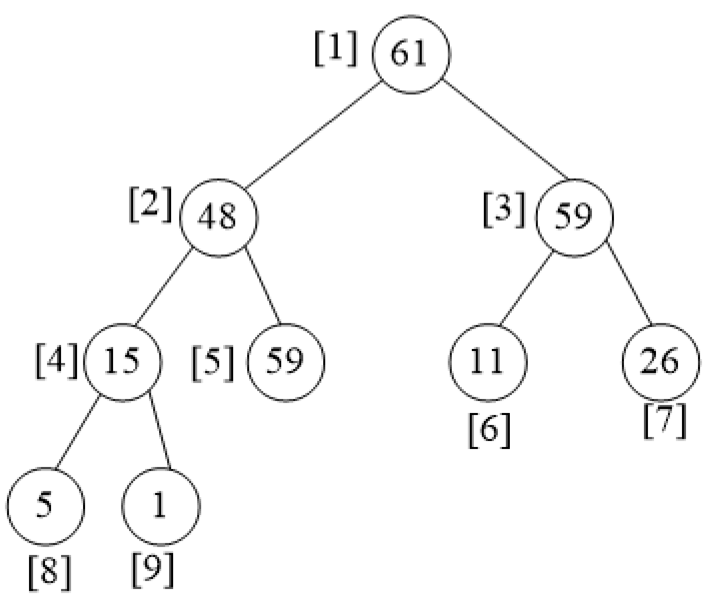


(a) Input array

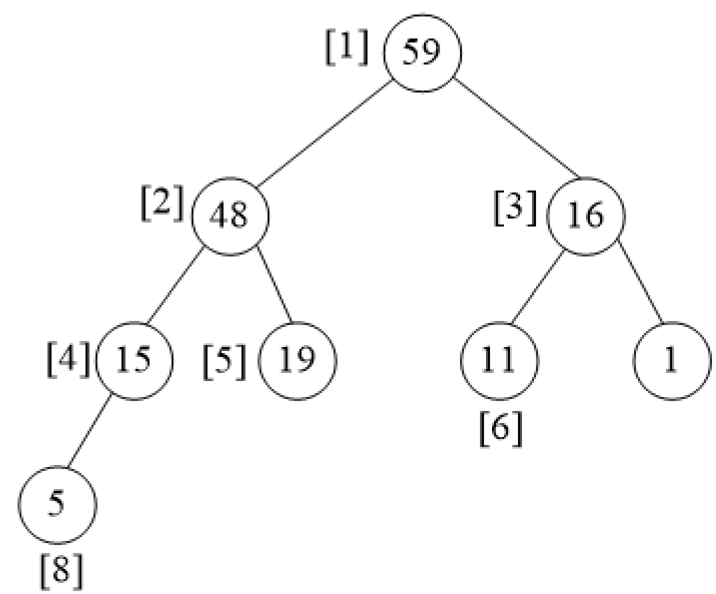


(b) Max heap after constructing

Heap Sort (2)

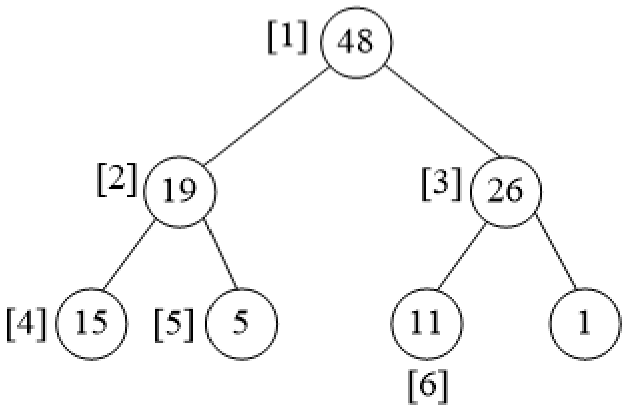


Heap size = 9
a[10] = 77

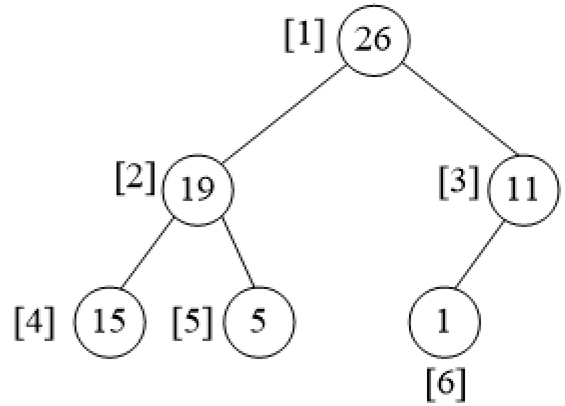


Heap size = 8
a[9]=61, a[10]=77

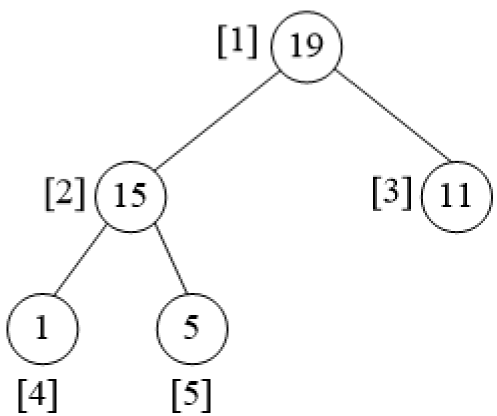
Heap Sort (3)



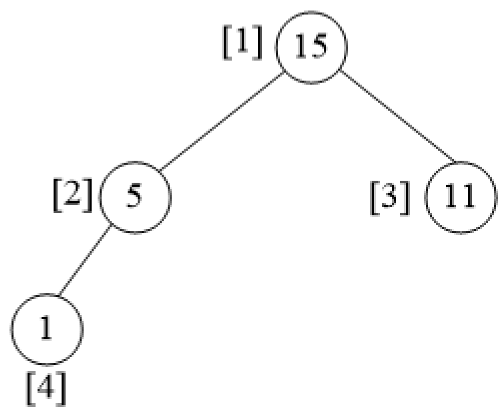
(c) 堆積大小 = 7
已排序 = [59, 61, 77]



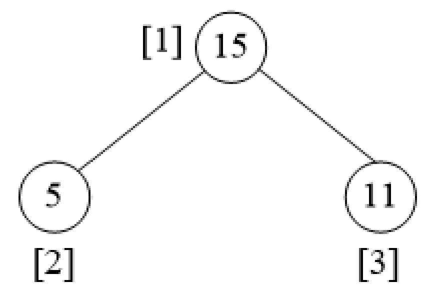
(d) 堆積大小 = 6
已排序 = [48, 59, 61, 77]



(e) 堆積大小 = 5
已排序 = [26, 48, 59, 61, 77]



(f) 堆積大小 = 4
已排序 = [19, 26, 48, 59, 61, 77]



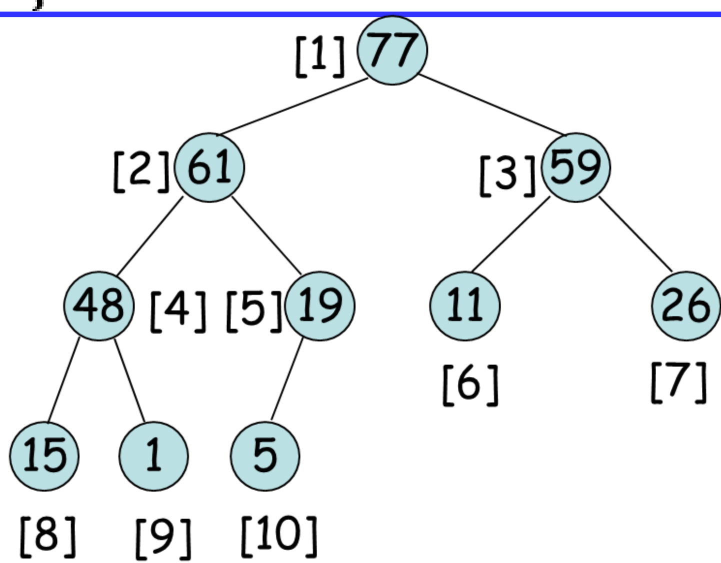
(g) 堆積大小 = 3
已排序 = [15, 19, 26, 48, 59, 61, 77]

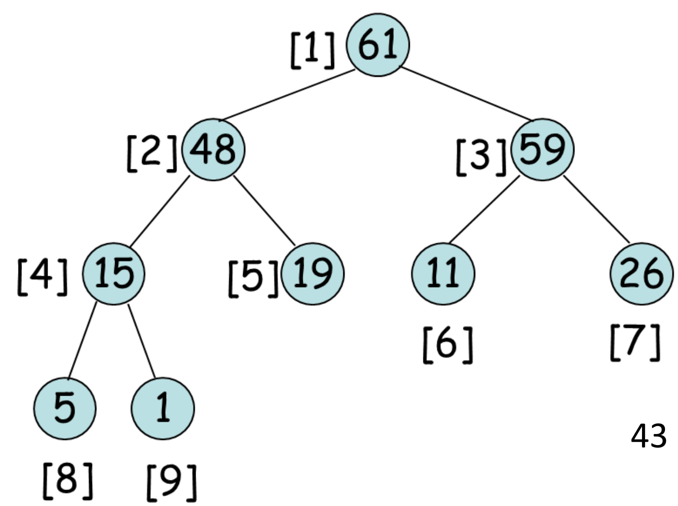
Heap Sort

```
template <class T>
void HeapSort(T *a, const int n)
{// 將 a[1:n] 排序成非遞減的順序
    for (int i = n/2; i >= 1; i--) // 建立堆積
        Adjust(a, i, n);
    for (i = n-1; i >= 1; i--)    // 排序
    {
        swap(a[1], a[i+1]);       // 對調目前堆疊中的第一個與最後一個
        Adjust(a, 1, i);   // 建立堆疊
    }
}
```

建max heap

逐一輸出

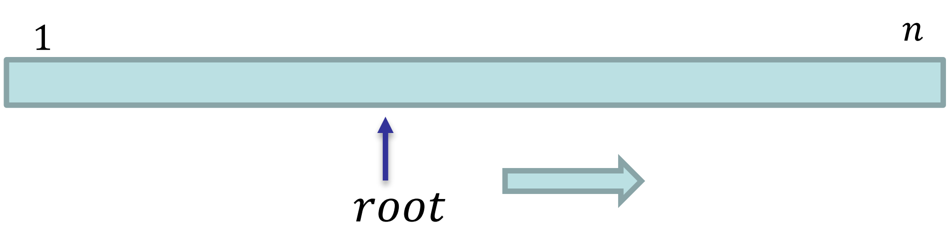


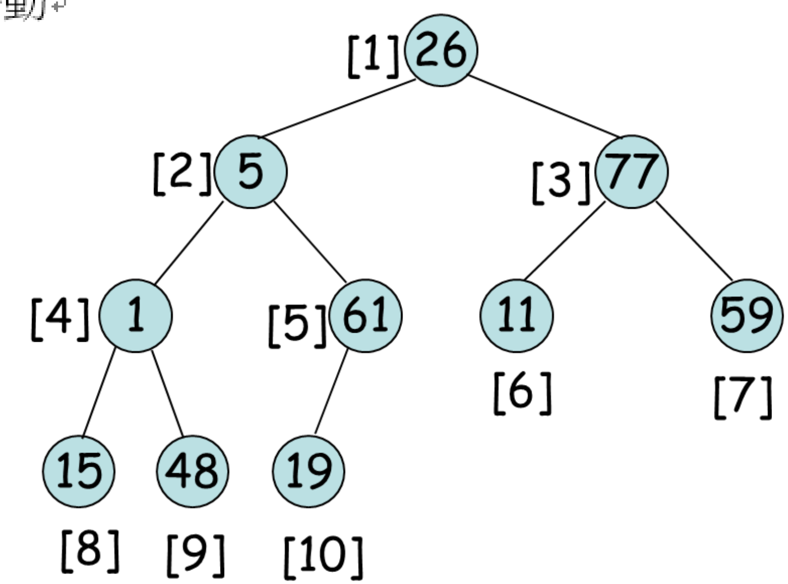


Adjusting a Max Heap

```
template <class T>
void Adjust(T *a, const int root, const int n)
{// 調整一棵樹根為 root 的二元樹使其符合堆積的性質。root 的左、右子樹都已經符合
 // 堆積的性質。沒有一個節點的索引值是 > n 的
    T e = a[root];
    // 找到 e 的適當位置
    for (int j =2*root; j <= n; j *=2) {
        if (j < n && a[j] < a[j+1]) j++; // j 是它父親的最大兒子
        if (e >= a[j]) break; // e 可以插入成為 j 的父親
        a[j / 2] = a[j]; // 把第 j 筆記錄往樹的上方移動
    }
    a[j / 2] = e;
}
```

從root位置開始，一路往下找最大的兒子





Time Complexity

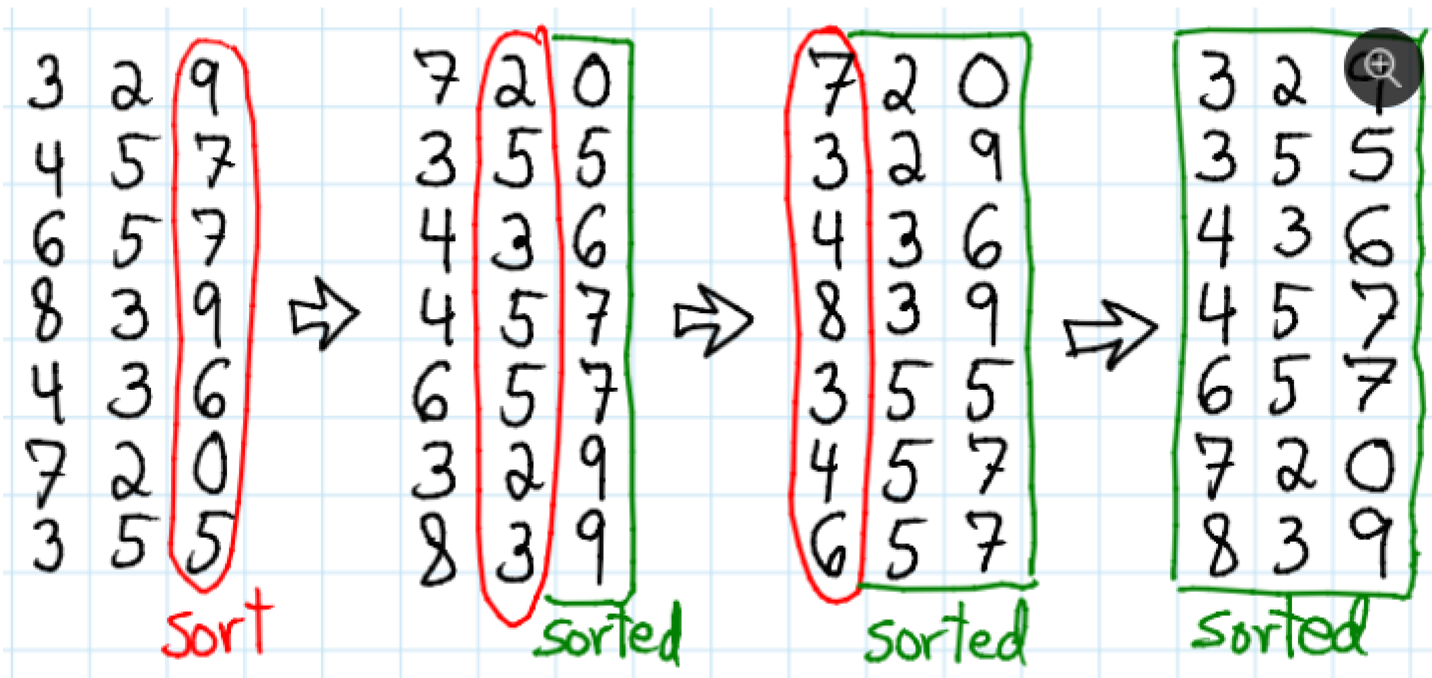
Algorithm	Average complexity	Best complexity	Worst complexity
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Modified Bubble sort	$O(n^2)$	$O(n)$	$O(n^2)$
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n)$	$O(n^2)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

Radix Sort

- 方法：least significant digit first (LSD)
  - 每個資料不與其它資料比較，根據key分佈來排序
  1) pass 1 ：從個位數開始處理。若是個位數為 1，則放在 bucket 1，以此類推…
  2) pass 2 ：處理十位數,
  3) pass 3：處理百位數…
- 好處：若以array處理，速度快
- Time complexity: $O((n+r)\log_r k)$
  - k: input data 之最大數
  - r: 以 r 為基數(radix)， $\log_r k$: 位數之長度
- 缺點: 若以array處理需要較多記憶體。使用 linked list，可減少所需記憶體，但會增加時間

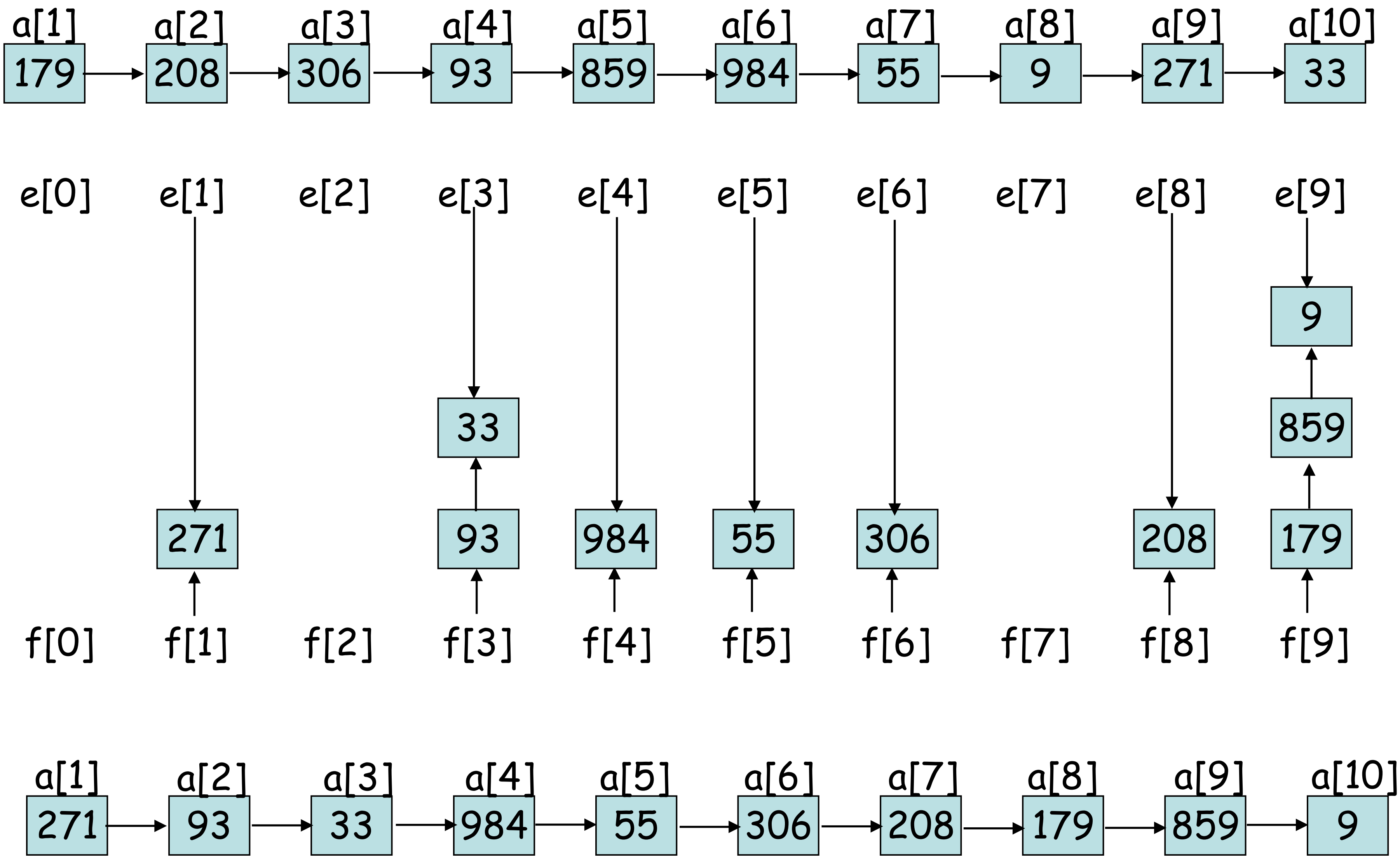
Radix Sort

- Least significant digit (LSD)：從最低有效鍵值開始排序（最小位數排到大）。
- Most significant digit (MSD)：從最高有效鍵值開始排序（最大位數排到小）。

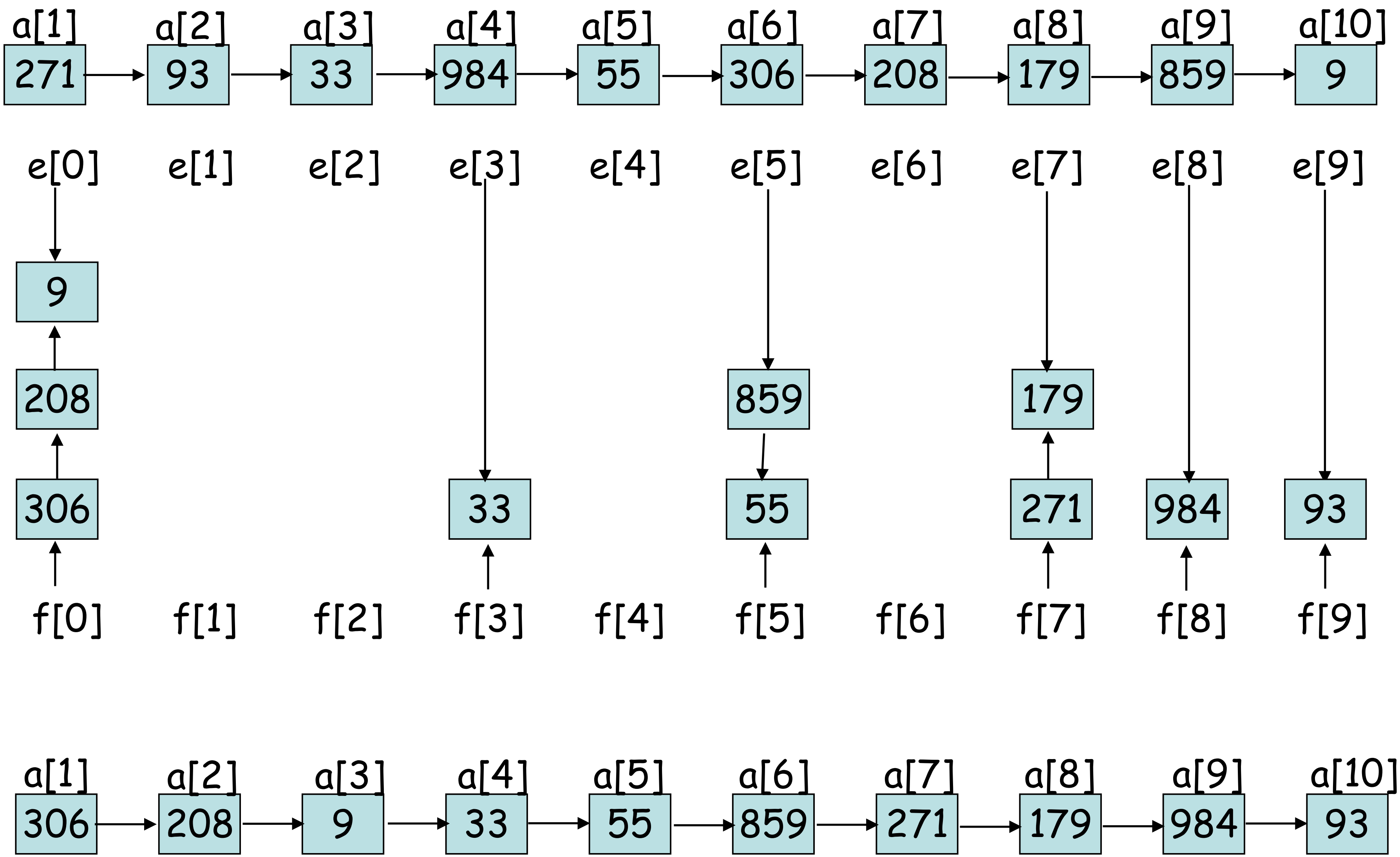


Radix Sort[1]

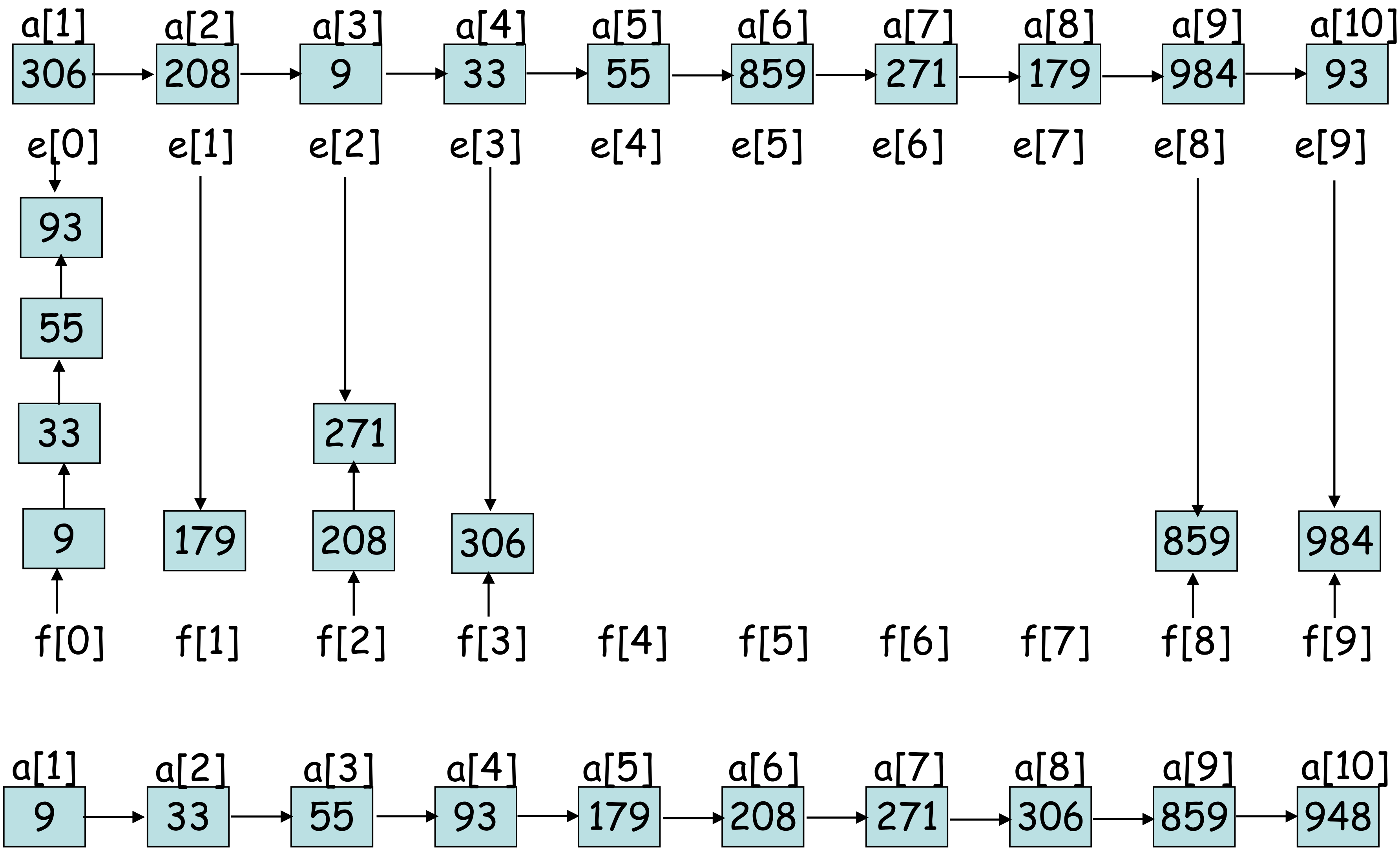
Radix Sort基數排序: Pass 1 (nondecreasing)



Radix Sort: Pass 2



Radix Sort: Pass 3



```
template <class T>
int RadixSort(T *a, int *lin
{// 使用一個 d 位元、基數
 // digit(a[i], j, r) 回傳 a[i
 // 每一個數字的範圍都是
    int e[r], f[r]; // 佇列的
    // 產生一個從 first 開
    int first = 1;
    for (int i =1; i < n; i+
    link[n] = 0;
```

```
    for (i = d-1 ; i >=0; i--){// 根據數字 i 來排序
        fill(f, f+r, 0); // 將容器初始化為空的佇列
        for (int current = first; current; current = link[current])
        {// 把記錄放到佇列/容器中
            int k = digit(a[current], i , r);
            if (f[k]== 0) f[k] = current;
            else link[e[k]] = current;
            e[k] =current;
        }
        for (j = 0; !f[j]; j++); // 找出第一個非空的佇列/容器
        first = f [j];
        int last = e[j];
        for (int k = j + 1; k < r; k++) // 連接其餘的佇列
            if (f[k]) {
                link[last] = f[k];
                last = e[k];
            }
            link[last] = 0;
    }
    return first;
}
```

d: 位數

r: 基數

123_{10}

List Sort

- All sorting methods require **excessive** data movement.
- The **physical data movement** tends to slow down the sorting process.

➢ Using **linked list to** minimize the physical data movement.

- insertion sort or merge sort

➢ Physically rearranging the records in place after sorting

Rearranging Sorted Linked List (1)

Sorted linked list, $first = 4$

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
$linka$	9	6	0	2	3	8	5	10	7	1

Add backward links to become a doubly linked list, $first = 4$

doubly linked list

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	26	5	77	1	61	11	59	15	48	19
$linka$	9	6	0	2	3	8	5	10	7	1
$linkb$	10	4	5	0	7	2	9	6	1	8

Rearranging Sorted Linked List (2)

R_1 is in place. $first = 2$

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	**1**	5	77	**26**	61	11	59	15	48	19
linka	**2**	6	0	**9**	3	8	5	10	7	**4**
linkb	**0**	4	5	**10**	7	2	9	6	**4**	8

R_1, R_2 are in place. $first = 6$

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	**5**	77	26	61	11	59	15	48	19
linka	2	**6**	0	9	3	8	5	10	7	1
linkb	0	**4**	5	10	7	2	9	6	1	8

Rearranging Sorted Linked List (3)

R_1, R_2, R_3 are in place. $first = 8$

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	**11**	26	61	**77**	59	15	48	19
linka	2	6	**8**	9	**6**	**0**	5	10	7	4
linkb	0	4	**2**	10	7	**5**	9	6	4	8

R_1, R_2, R_3, R_4 are in place. $first = 10$

i	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
key	1	5	11	**15**	61	77	59	**26**	48	19
linkb	2	6	8	**10**	6	0	5	**9**	7	**8**
linkb	0	4	2	**6**	7	5	9	**10**	**8**	8

```
template <class T>
void List1(T *a, int *linka, const int n, int first)
{ // 重新排列從 first 開始的排序好的鏈，使得記錄 a[1:n] 排序好
    int *linkb = new int[n]; // 後向鏈結陣列
    int prev = 0;
    for (int current = first; current; current = linka[current])
    { // 把鏈轉換成雙鏈結串列
        linkb[current] = prev;
        prev = current;
    }

    for (int i = 1; i < n; i++) // 移動 a[first]到位置 i
    {
        if (first != i) {
            if (linka[i]) linkb[linka[i]] = first;
            linka[linkb[i]] = first;
            swap(a[first], a[i]);
            swap(linka[first], linka[i]);
            swap(linkb[first], linkb[i]);
        }
        first = linka[i];
    }
}
```

Doubly linked list

Rearrange the list

Table Sort

- The list-sort technique is not well suited for quick sort and heap sort.
- One can maintain an auxiliary table, t, with one entry per record, an indirect reference to the record.
- Initially, $t[i] = i$. When a swap are required, only the table entries are exchanged.
- After sorting, the list a[t[1]], a[t[2]], a[t[3]]…are sorted.

- Table sort is suitable for all sorting methods.

Permutation Cycle

- After sorting:

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

- Permutation [3 2 8 5 7 1 4 6]
- Every permutation is made up of disjoint permutation cycles:
  - (1, 3, 8, 6) nontrivial cycle
    - R1 now is in position 3, R3 in position 8, R8 in position 6, R6 in position 1.
  - (4, 5, 7) nontrivial cycle
  - (2) trivial cycle

Table Sort Example

Initial configuration

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8
key	35	14	12	42	26	50	31	18
t	3	2	8	5	7	1	4	6

after rearrangement of first cycle

key	12	14	18	42	26	35	31	50
t	1	2	3	5	7	6	4	8

after rearrangement of second cycle

key	12	14	18	26	31	35	42	50
t	1	2	3	4	5	6	7	8

Code for Table Sort

```
template <class T>
void Table(T* a, const int n, int *t)
{
    for (int i = 1; i < n; i++) {
        if (t[i] != i) {// nontrivial cycle starting at i
            T p = a[i];
            int j = i;
            do {
                int k = t[j]; a[j] = a[k]; t[j] = j;
                j = k;
            } while (t[j] != i)
            a[j] = p;  // j is the position for record p
            t[j] = j;
        }
    }
}
```

Summary of Internal Sorting

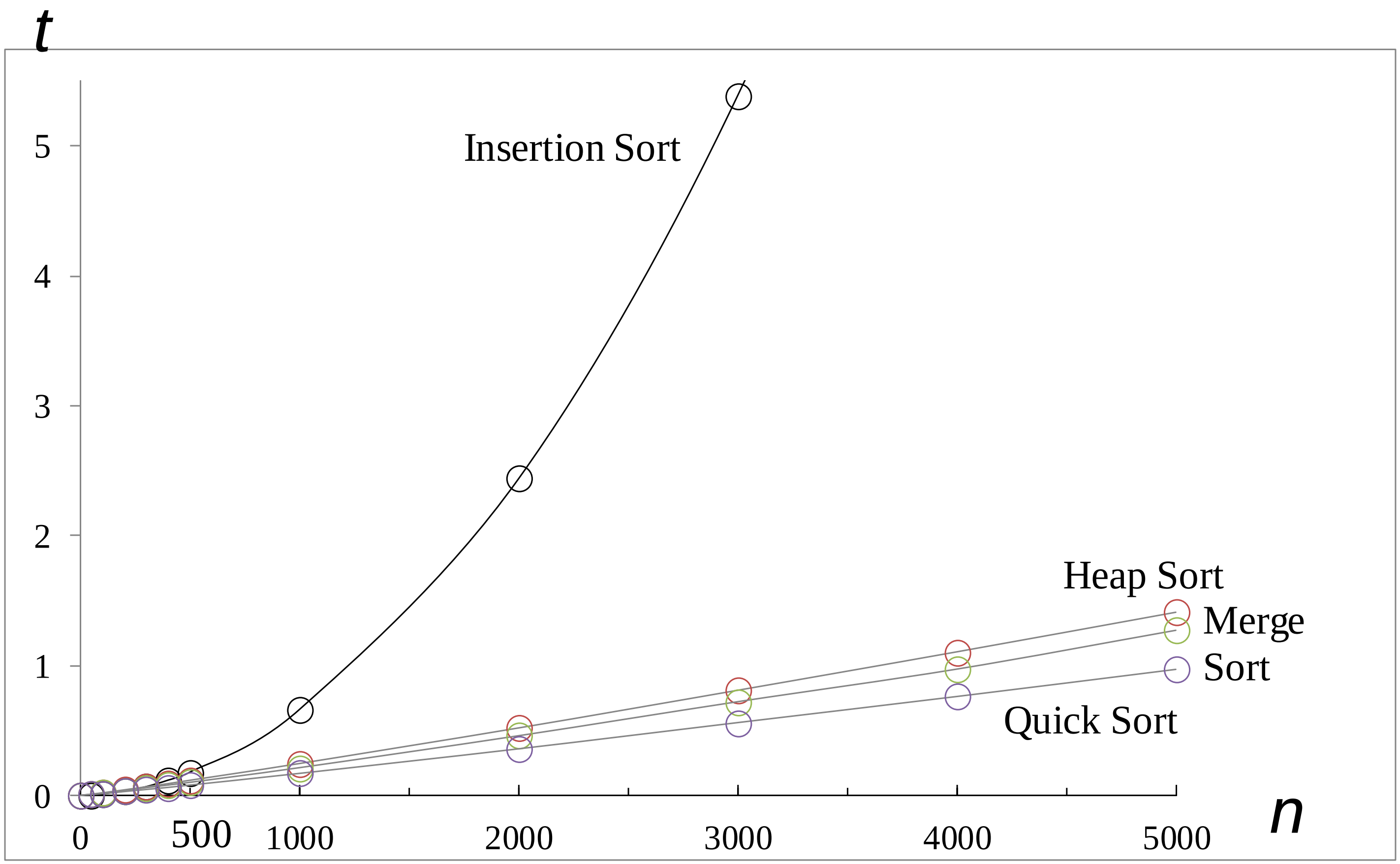
- No one method is best under all circumstances.
  - Insertion sort is good when the list is already partially ordered. And it is the best for small *n*.
  - Merge sort has the best worst-case behavior but needs more storage than heap sort.
  - Quick sort has the best average behavior, but its worst-case behavior is $O(n^2)$.
  - The behavior of radix sort depends on the size of the keys and the choice of *r*.

Complexity Comparison of Sort Methods

Method	Worst	Average
Insertion Sort	n^2	n^2
Heap Sort	$n \log n$	$n \log n$
Merge Sort	$n \log n$	$n \log n$
Quick Sort	n^2	$n \log n$
Radix Sort	$(n+r)\log_r k$	$(n+r)\log_r k$

k: input data 之最大數　r: 以 r 為基數(radix)

Average Execution Time



Average execution time, *n* = # of elements, *t*=milliseconds

External Sorting

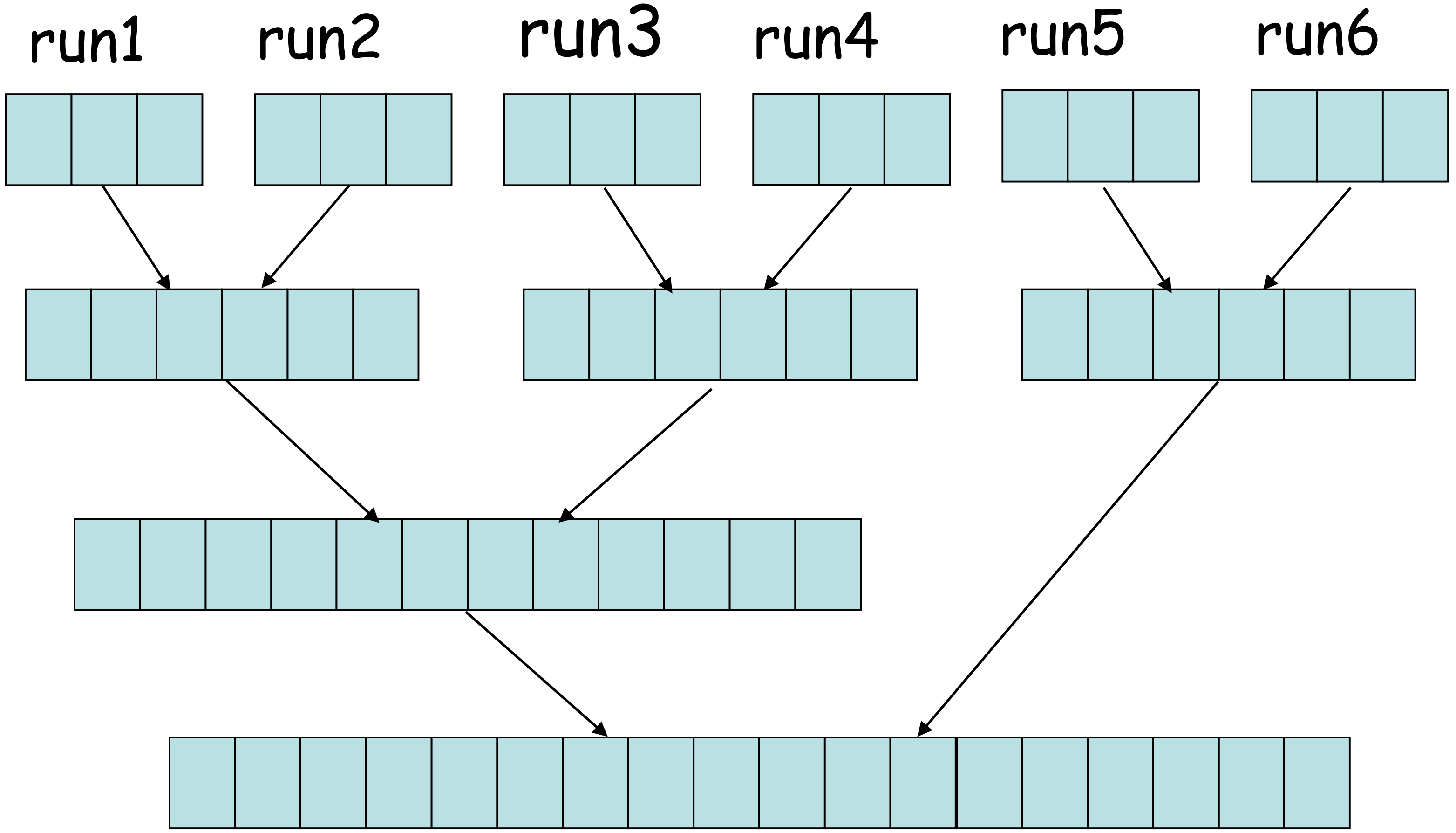
- The lists to be sorted are too large to be contained totally in the internal memory. So internal sorting is impossible.
- The list (or file) to be sorted resides on a disk.
- Block: unit of data read from or written to a disk at one time. A block generally consists of several records.
- read/write time of disks:
  - seek time 搜尋時間：把讀寫頭移到正確磁軌 (track, cylinder)
  - latency time 延遲時間：把正確的磁區(sector)轉到讀寫頭下
  - transmission time 傳輸時間：把資料區塊傳入/讀出磁碟

Merge Sort as External Sorting

- The most popular method for sorting on external storage devices is merge sort.

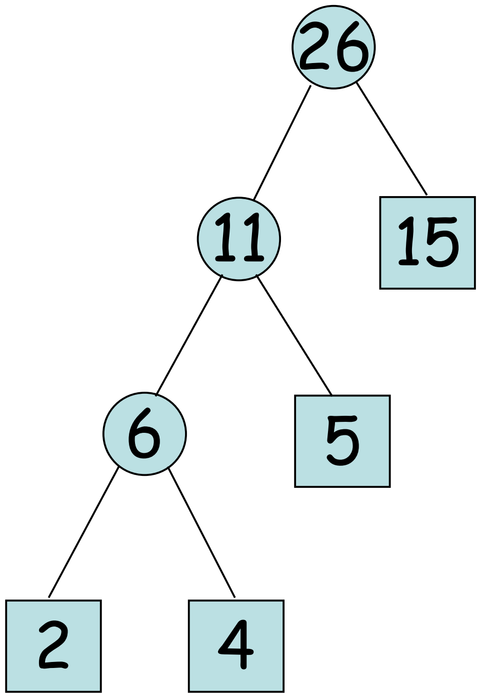
- Phase 1: Obtain sorted runs (segments) by internal sorting methods, such as heap sort, merge sort, quick sort or radix sort. These sorted runs are stored in external storage.
- Phase 2: Merge the sorted runs into one run with the merge sort method.

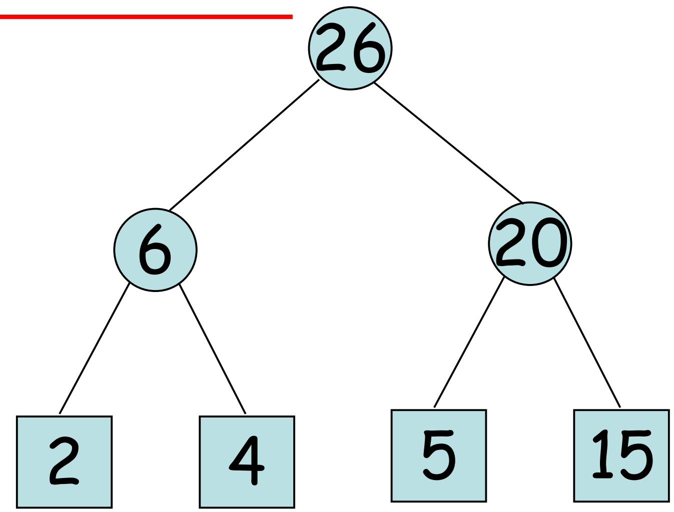
Merging the Sorted Runs



Optimal Merging of Runs

- In the external merge sort, the sorted runs may have different lengths. <u>If shorter runs are merged first, the required time is reduced.</u>





weighted external path length
= 2*3 + 4*3 + 5*2 + 15*1
= 43

weighted external path length
= 2*2 + 4*2 + 5*2 + 15*2
= 52

Huffman Algorithm

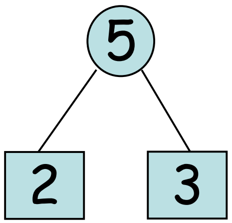
- External path length: sum of the distances of all external nodes from the root.
- Weighted external path length:

$$\sum_{1 \le i \le n+1} q_i d_i, \text{ where } d_i \text{ is the distance from root to node } i$$

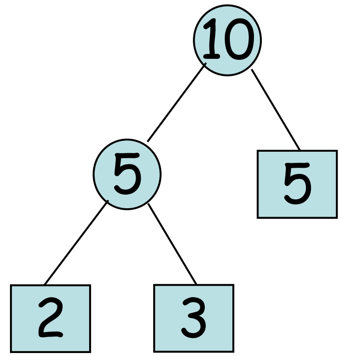
$$q_i \text{ is the weight of node } i.$$

- Huffman algorithm: to solve the problem of finding a binary tree with minimum weighted external path length.
- Huffman tree:
  - Solve the 2-way merging problem
  - Generate Huffman codes for data compression

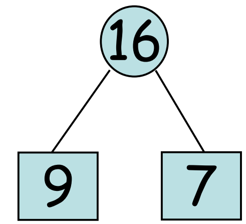
Construction of Huffman Tree



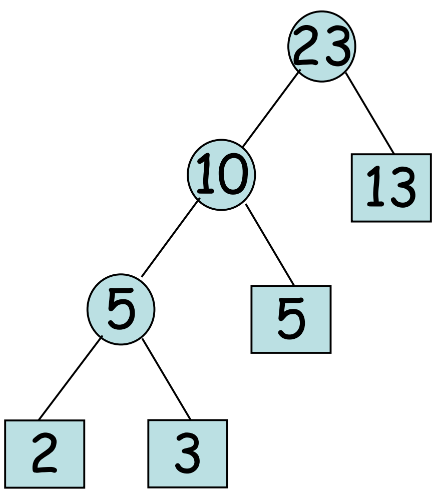
(a) [2,3,5,7,9,13]



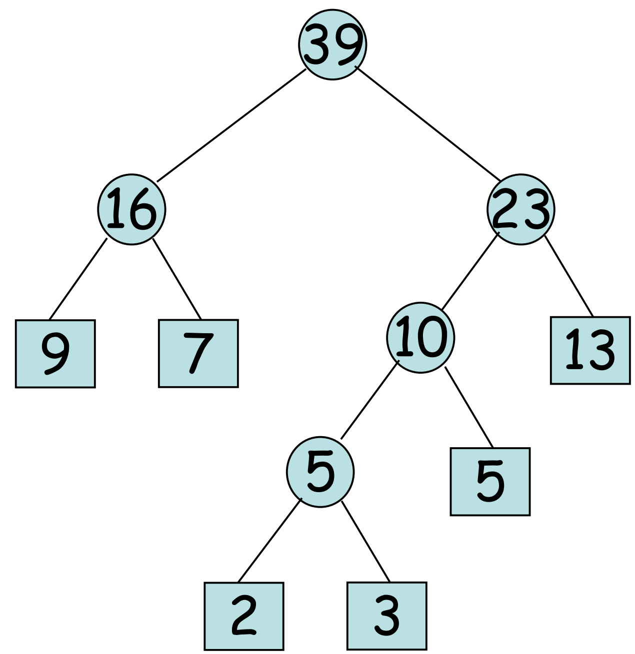
(b) [5,5,7,9,13]



(c) [7, 9, 10,13]



(d) [10,13,16]



(e) [16, 23]

Min heap is used.
Time: O(n log n)

Huffman Code (1)

- Each symbol is encoded by 2 bits (fixed length)

symbol	code
A	00
B	01
C	10
D	11

- Message A B A C C D A would be encoded by 14 bits:

`00 01 00 10 10 11 00`

Huffman Code (2)

- Huffman codes (variable-length codes)

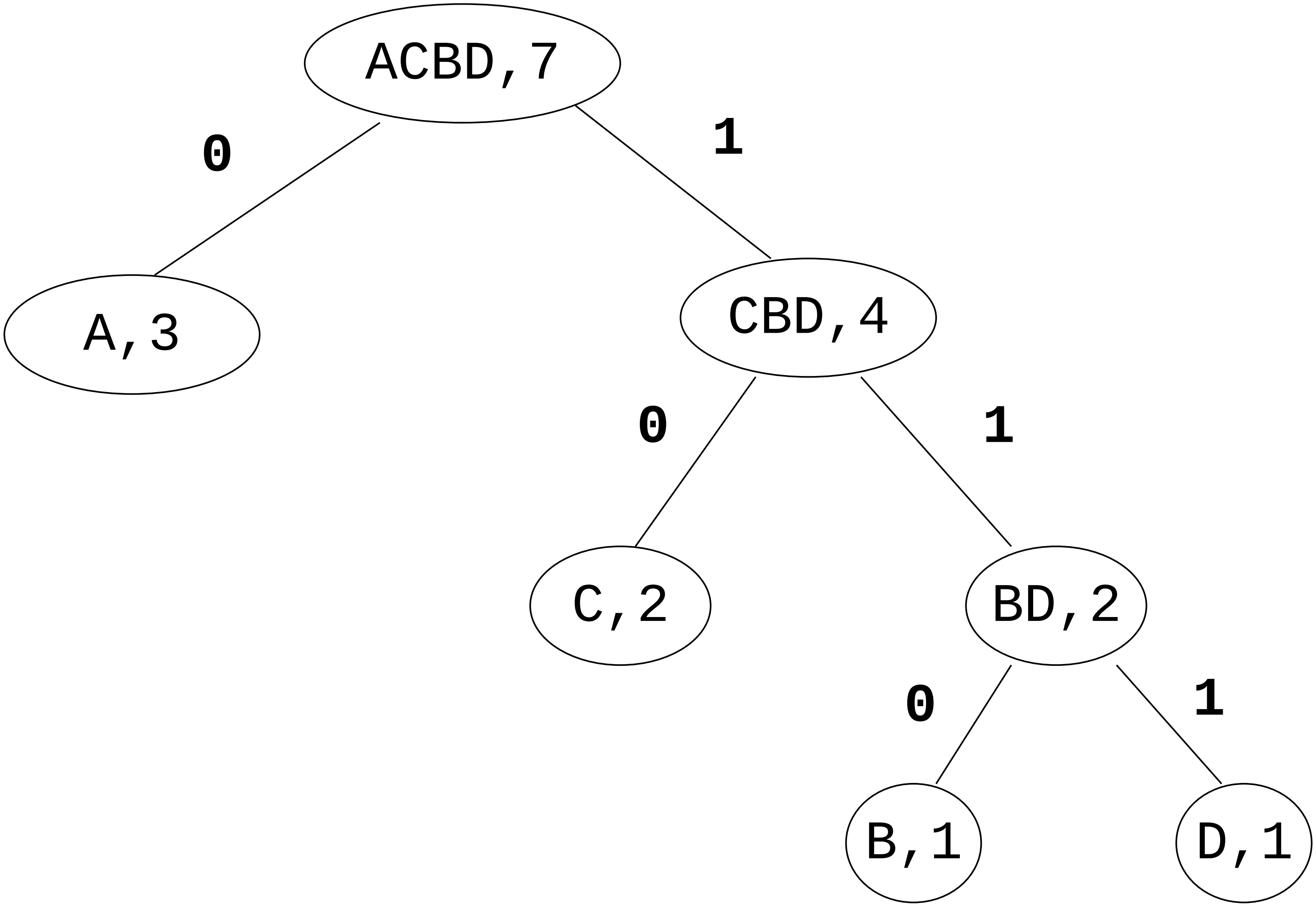
symbol	code
A	0
B	110
C	10
D	111

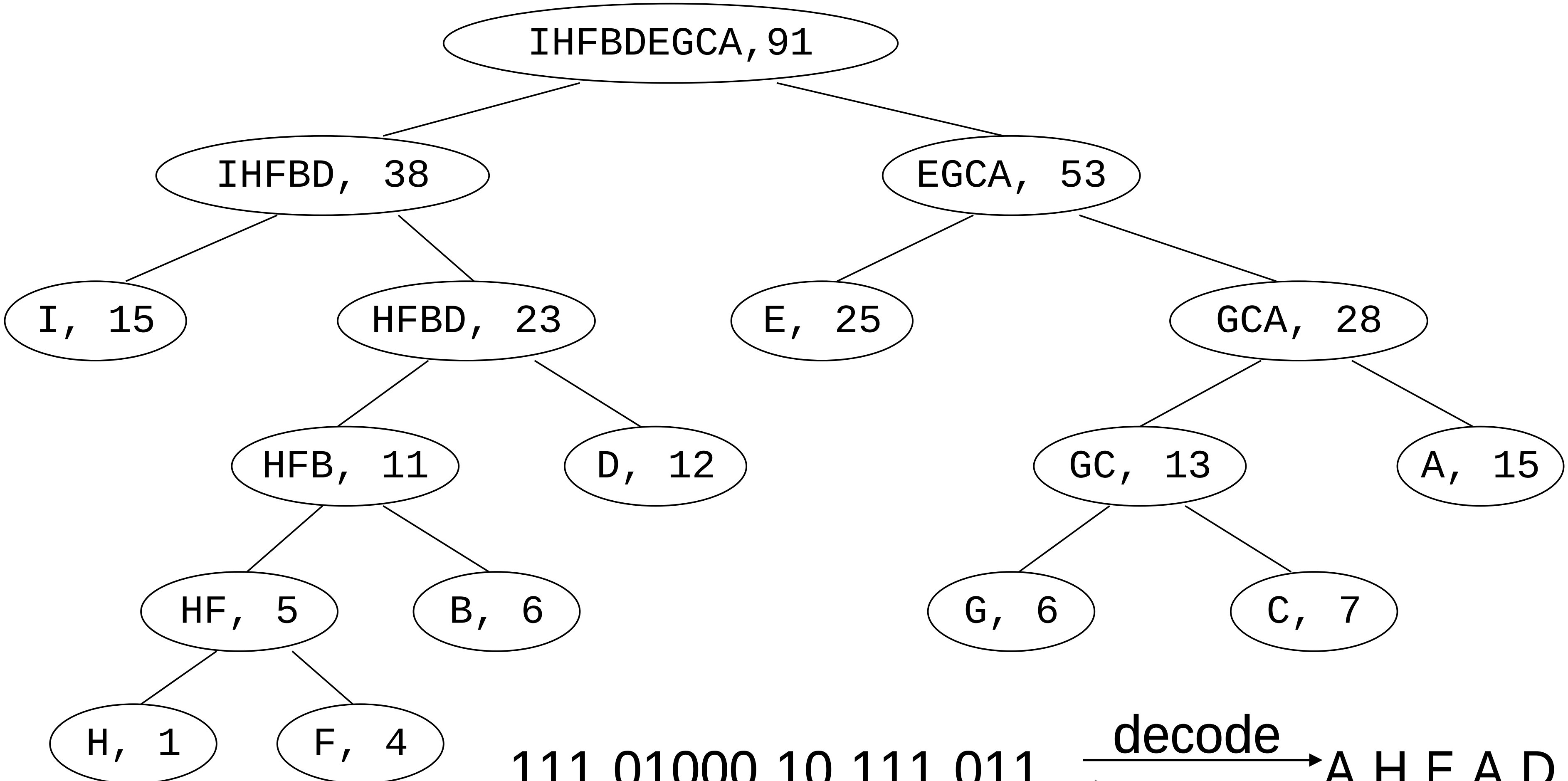
- Message A B A C C D A would be encoded by 13 bits:

  0 110 0 10 10 111 0

- A frequently used symbol is encoded by a short bit string.

Huffman Tree





111 01000 10 111 011 $\xrightarrow{\text{decode}}$ A H E A D
111 01000 10 111 011 $\xleftarrow[\text{encode}]{}$ A H E A D

Sym	Freq	Code	Sym	Freq	Code	Sym	Freq	Code
A	15	111	D	12	011	G	6	1100
B	6	0101	E	25	10	H	1	01000
C	7	1101	F	4	01001	I	15	00