

資料結構



演算法

排序

Sort

排序 Sort

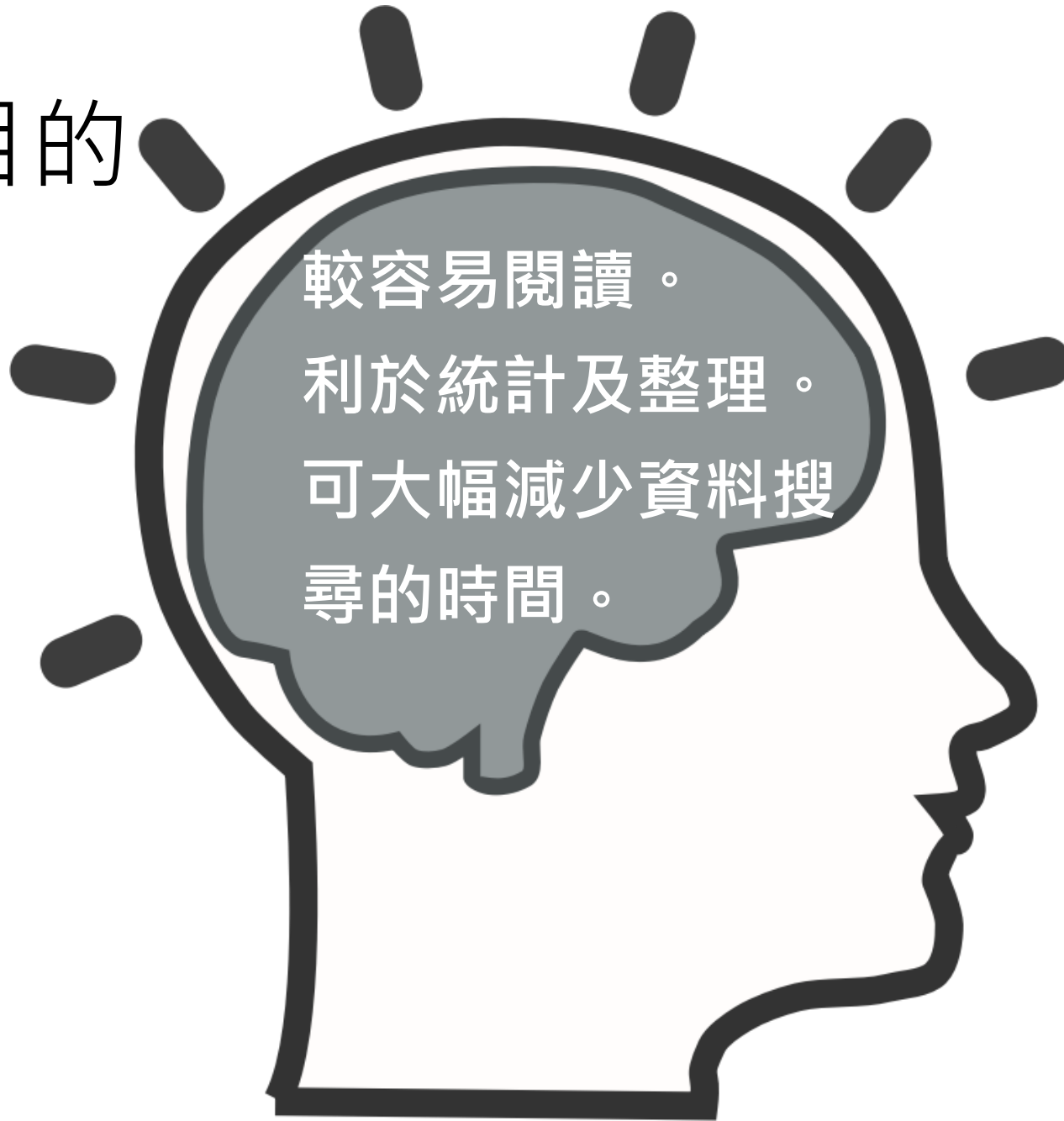
[十八豆教育科技](#)

# 排序的目的

較容易閱讀。

利於統計及整理。

可大幅減少資料搜尋的時間。



5個整數，  
由小到大排列。



## 先想想簡單的

只有兩個數字時，就比較這兩個：

- 前面數字比較大就跟後面的交換
- 前面數字比較小就維持不動



## 複雜一點點呢？

若有三個數字時，維持兩個數字排序的原則，兩個數字比較，前面數字比較大就跟後面的交換，當所有兩個數字的組合都比較過後，結果自然就是排序後的組合



原始資料

• [66, 22, 7]

第 1 回合

- 比較第 1 個跟第 2 個數字
- 需要交換: [66, 22, 7] => [22, 66, 7]

第 2 回合

- 比較第 2 個跟第 3 個數字
- 需要交換: [22 66, 7] => [22, 7, 66]

第 3 回合

- 第2個數字在第2回合有被移動過，重新比較第 1 跟第 2 個數字
- 需要交換: [22, 7, 66] => [7, 22, 66]

第 4 回合

- 第2個數字在第3回合可能有移動過，重新比較第 2 跟第 3 個數字
- 需要交換: [7, 22, 66] => [7, 22, 66]

結果

• [7, 22, 66]

# 我們知道

- 排序的中心原則就是由小到大（或由大到小）
- **兩兩數字**比較，前面比較大就跟後面數字交換
- 只要確定所有兩兩數字的配對組合都有比較過，就可得到想要的排序順序

# 問題是

兩個數字

=> 第 1 個跟第 2 個數字比

三個數字

=> 第 1 個跟第 2 個數字比

=> 第 2 個跟第 3 個數字比

=> 第 1 個跟第 3 個數字比

一堆數字

=> 要怎麼組合阿???



# 換個方式想

- 兩個數字相比，大的數字會被換到後面位置
- N個數字時，從第 1 個位置開始：
  - 第 1 回合：第 1 個位置的數字跟第 2 個位置的數字比，第 1 個位置的數字比較大就跟第 2 個位置的數字交換位置
  - 第 2 回合：第 2 個位置的數字跟第 3 個位置的數字比，第 2 個位置的數字比較大就跟第 3 個位置的數字交換位置
  - 第 3 回合：第 3 個位置的數字跟第 4 個位置的數字比，第 3 個位置的數字比較大就跟第 4 個位置的數字交換位置
  - 第 4 回合、第 5 回合、第 6 回合...
  - 第 N-1 回合：第 N-1 個位置的數字跟第 N 個位置的數字比，第 N-1 個位置的數字比較大就跟第 N 個位置的數字交換位置



# 也就是說

- 經過第 1 次的  $N-1$  回合，第 1 大的數字會被推到最後 1 個位置
- 經過第 2 次的  $N-1$  回合，第 2 大的數字會被推到倒數第 2 個位置
- 經過第 3 次的  $N-1$  回合，第 3 大的數字會被推到倒數第 3 個位置
- .....
- .....
- .....
- 經過第  $N$  次的  $N-1$  回合，第  $N$  大的數字會被推到倒數第  $N$  個位置
  - ⇒ 也就是最小的數字會被放到最前面的方式
  - ⇒ 所以，最後就產生由小到大排列的數字組合了！！！！

# 就這樣比...

原始資料	第一回合				第二回合				第三回合				第四回合			
4	3	3	3	3	2	2	2	2	2	2	2	2	1	繼續往下比...		
3	4	2	2	2	3	3	3	3	1	1	1	1	2			
2	2	4	4	4	4	1	1	1	3	3	3	3	3			
5	5	5	5	1	1	4	4	4	4	4	4	4	4			
1	1	1	1	5	5	5	5	5	5	5	5	5	5			

方法:  
 比4大回合，  
 每一回合又比4小回合  
 兩兩相比，大的往後推



# 將方法變成程式碼

Bubblesort.c

```
#include <stdio.h>
int main() {
    int array[11];
    int i,j,temp;

    for (i=0;i<5;i++)
        scanf("%d",&array[i]);

    for (i=0;i<5-1;i++){ // 比4大回合
        for(j=0;j<5-1-i;j++){ // 比4小回合
            if(array[j]>array[j+1]){
                temp=array[j];
                array[j]=array[j+1];
                array[j+1]=temp;
            }
        }
    }

    for (i=0;i<5;i++)
        printf("%d ",array[i]);

    return 0;
}
```

輸入資料

前一個比後一個大，  
就做交換

輸出資料



第一大回合

排序前:	[4, 3, 2, 5, 1]
i: 0, j: 0	[3, 4, 2, 5, 1]
i: 0, j: 1	[3, 2, 4, 5, 1]
i: 0, j: 2	[3, 2, 4, 5, 1]
i: 0, j: 3	[3, 2, 4, 1, 5]

第一大回合結束後，最大的值會被推到最下面

第二大回合

i: 1, j: 0	[2, 3, 4, 1, 5]
i: 1, j: 1	[2, 3, 4, 1, 5]
i: 1, j: 2	[2, 3, 1, 4, 5]
i: 1, j: 3	[2, 3, 1, 4, 5]

第二大回合結束後，次大的值會被推到倒數第二層

第三大回合

i: 2, j: 0	[2, 3, 1, 4, 5]
i: 2, j: 1	[2, 1, 3, 4, 5]
i: 2, j: 2	[2, 1, 3, 4, 5]
i: 2, j: 3	[2, 1, 3, 4, 5]

第三大回合結束後，第三大的值會被推到倒數第三層

第四大回合

i: 3, j: 0	[1, 2, 3, 4, 5]
i: 3, j: 1	[1, 2, 3, 4, 5]
i: 3, j: 2	[1, 2, 3, 4, 5]
i: 3, j: 3	[1, 2, 3, 4, 5]

第四大回合結束後，第四大的值會被推到倒數第四層

第五大回合

i: 4, j: 0	[1, 2, 3, 4, 5]
i: 4, j: 1	[1, 2, 3, 4, 5]
i: 4, j: 2	[1, 2, 3, 4, 5]
i: 4, j: 3	[1, 2, 3, 4, 5]

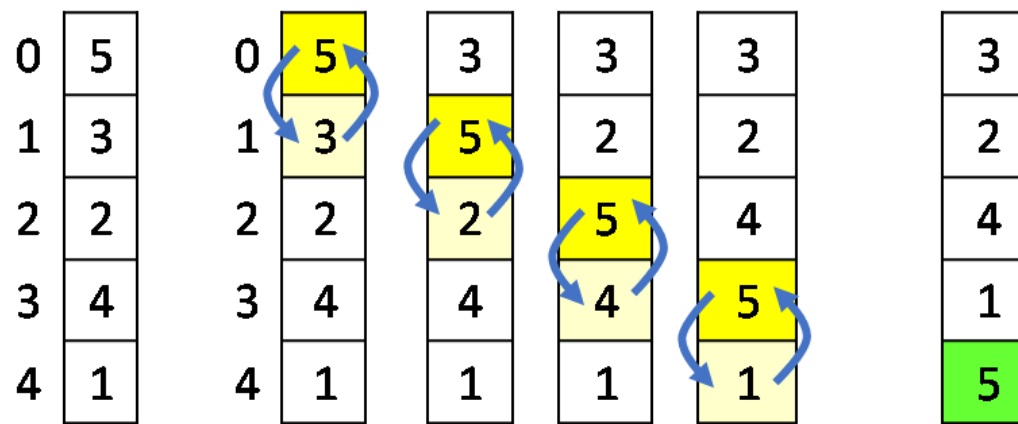
第五大回合結束後，第五大（最小）的值會被推到倒數第五層（最上層）

排序後:	[1, 2, 3, 4, 5]
------	-----------------



# 這是泡沫排序法

- 重複地走訪要排序的數列，一次比較兩個元素，如果他們的順序錯誤就把他們交換過來。
- 走訪數列的工作是重複地進行直到沒有再需要交換，也就是說該數列已經排序完成。
- 越小的元素會經由交換慢慢「浮」到數列的頂端。
- 越大的元素會經由交換慢慢「沉」到數列的底端。



# 泡沫排序法的時間複雜度

```
#include <stdio.h>
int main() {
    int array[11];
    int i,j,temp;
```

**O(N)**

```
for (i=0;i<5;i++)
    scanf("%d",&array[i]);
```

**O(N<sup>2</sup>)**

```
for (i=0;i<5;i++){ // 比4大回合
    for(j=0;j<5-1-i;j++){ // 比4小回合
        if(array[j]>array[j+1]){
            temp=array[j];
            array[j]=array[j+1];
            array[j+1]=temp;
        }
    }
}
```

**O(N)**

```
for (i=0;i<5;i++)
    printf("%d ",array[i]);
```

```
return 0;
}
```

當 N 很大時，泡沫排序法需要執行很久的！

**O(N<sup>2</sup>)**



還有其他方法嗎？



# 也可以這樣比

原始  
資料

• [4, 3, 2, 5, 1]

Step 1

- 找出最小數字所在的位置 → 第 5 個位置
- 最小數字所在的位置 (第 5 個位置) 跟第 1 個位置的數字交換
- 需要交換: [4, 3, 2, 5, 1] ⇒ [1, 3, 2, 5, 4]

Step 2

- 找出第 2 小數字所在的位置 → 第 3 個位置
- 第 2 小數字所在的位置 (第 3 個位置) 跟第 2 個位置的數字交換
- 需要交換: [1, 3, 2, 5, 4] ⇒ [1, 2, 3, 5, 4]

Step 3

- 找出第 3 小數字所在的位置 → 第 3 個位置
- 第 3 小數字已經在第 3 個位置
- 不需要交換: [1, 2, 3, 5, 4]

Step 4

- 找出第 4 小數字所在的位置 → 第 5 個位置
- 第 4 小數字所在的位置 (第 5 個位置) 跟第 4 個位置的數字交換
- 需要交換: [1, 2, 3, 5, 4] ⇒ [1, 2, 3, 4, 5]

Step 5

- 找出第 5 小數字所在的位置 → 第 5 個位置
- 第 5 小數字已經在第 5 個位置
- 不需要交換: [1, 2, 3, 4, 5]

答案

• [1, 2, 3, 4, 5]

方法:  
在每一回合中找出最小值





# 將方法變成程式碼

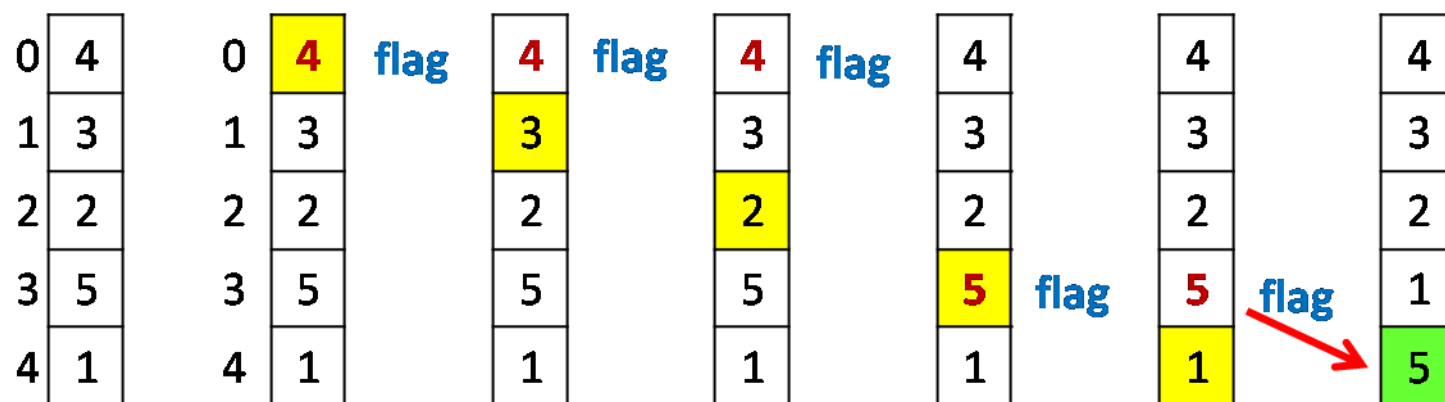
```
#include <stdio.h>
int main()
{
    int num[10], flag;
    int i, j, temp=0;
    for (i=0; i<5; i++) {
        scanf("%d", &num[i]);
    }
    for (i=0; i<5; i++) {
        flag = i;
        for (j=i+1; j<5; j++) {
            if (num[flag]>num[j]){
                flag = j; //找到最小值的位置
            }
        }
        temp = num[flag];
        num[flag] = num[i];
        num[i] = temp;
        for(j=0; j<5; j++) {
            printf("%d ", num[j]);
        }
        printf("\n");
    }
    return 0;
}
```

執行結果

```
dice - sort $ ./selectionsort.exe
5
4
3
2
1
1 4 3 2 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

# 這是選擇排序法

- 重複地走訪要排序的數列，每次走訪會找出該階段最小的元素。
- 每階段的走訪數列元素會比上一輪少一項。
- 走訪數列的工作是重複地進行直到所有階段的最小元素值完全找到。



# 選擇排序法的時間複雜度

```
#include <stdio.h>
int main()
{
    int num[10], flag;
    int i, j, temp=0;
    for (i=0; i<5; i++) {
        scanf("%d", &num[i]);
    }

    for (i=0; i<5; i++) {
        flag = i;
        for (j=i+1; j<5; j++) {
            if (num[flag]>num[j]){
                flag = j; //找到最小值的位置
            }
        }
        temp = num[flag];
        num[flag] = num[i];
        num[i] = temp;
        for(j=0; j<5; j++) {
            printf("%d ", num[j]);
        }
        printf("\n");
    }

    return 0;
}
```

$O(N)$

$O(N^2)$

當  $N$  很大時，選擇排序法跟泡沫排序法一樣需要執行很久時間！

$O(N^2)$

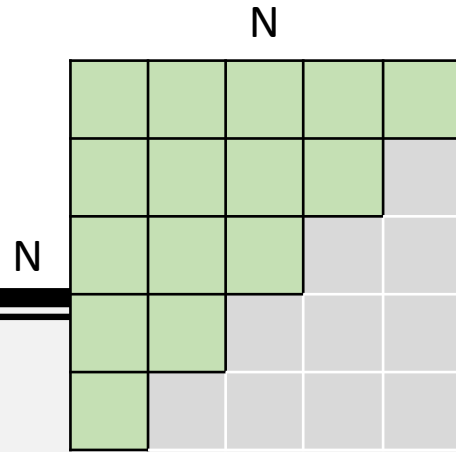
好像都很慢？



$O(n^2)$

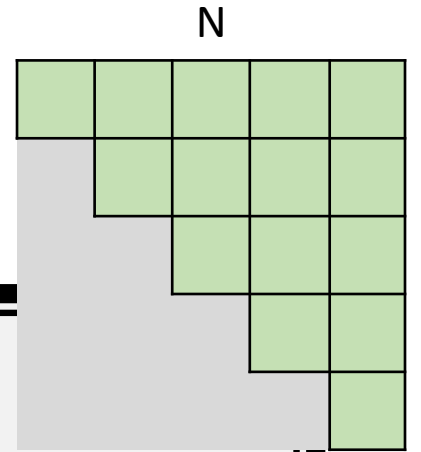
泡沫排序法

```
for (i=0;i<5-1;i++){
  for(j=0;j<5-1-i;j++){
    if(num[j]>num[j+1]){
      temp=num[j];
      num[j]=num[j+1];
      num[j+1]=temp;
    }
  }
}
```



選擇排序法

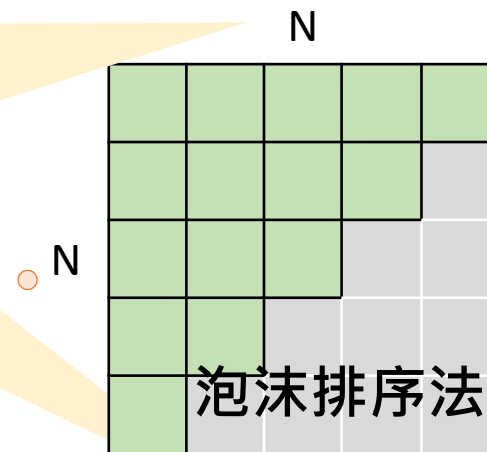
```
for (i=0; i<5; i++) {
  flag = i;
  for (j=i+1; j<5; j++) {
    if (num[flag]>num[j]){
      flag = j; //最小值位置
    }
  }
  temp = num[flag];
  num[flag] = num[i];
  num[i] = temp;
}
```



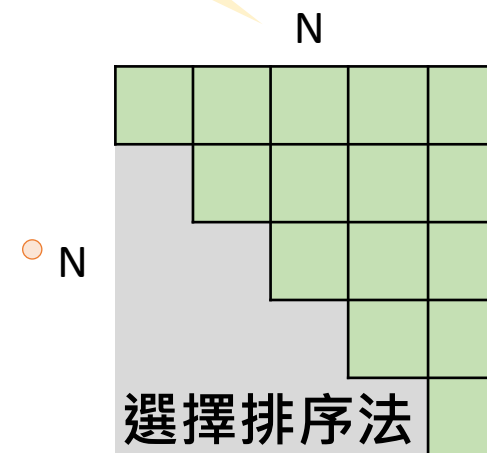
# $O(n^2)$

要做排序，每個元素必定會被取出至少比較一次。

因此寬度的  $N$  無法避免。

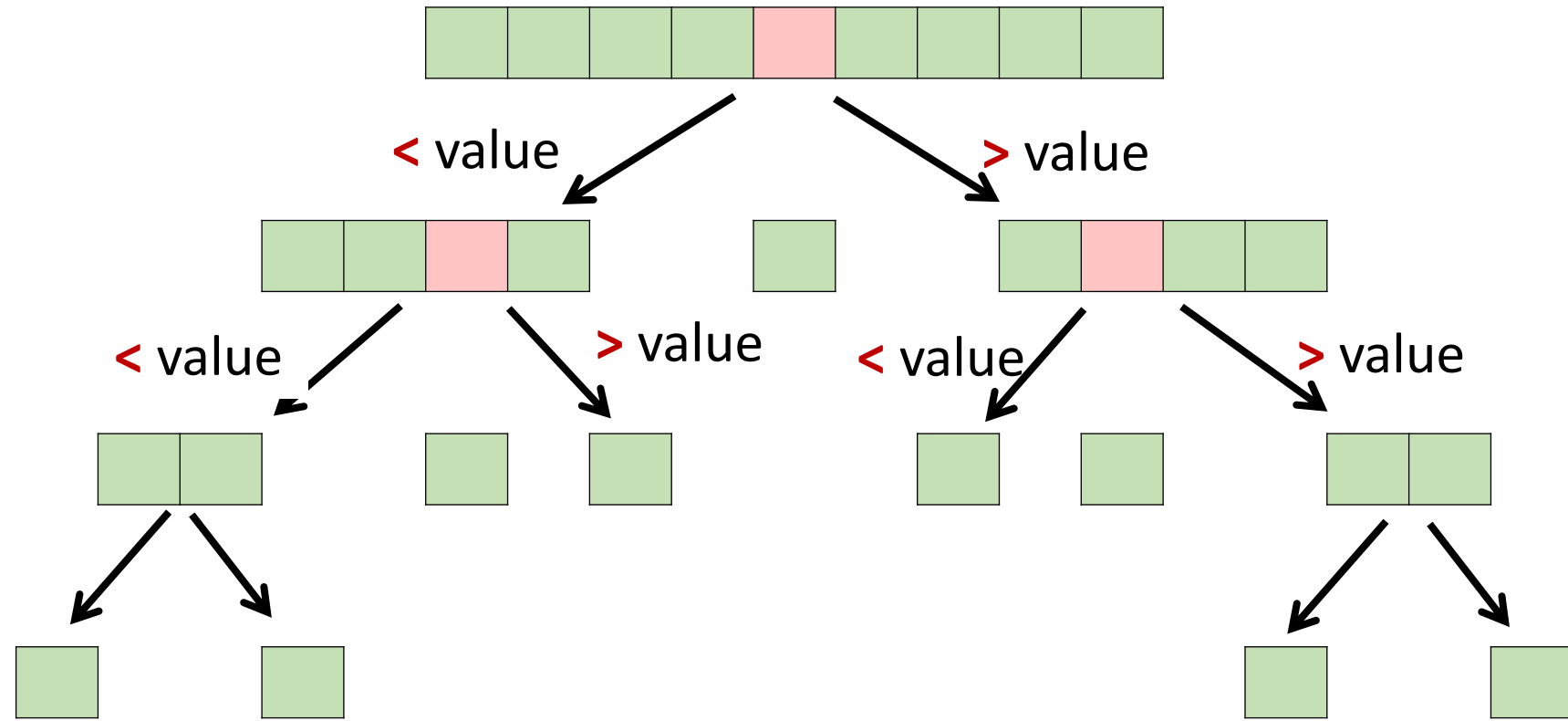


若是讓高度的  $N$  降低呢？



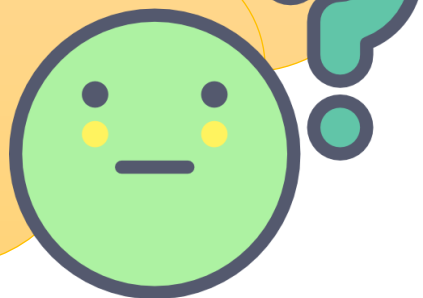
# 如果這樣排序呢？

- 每一回合找出中間值，根據中間值將資料分兩區，一區內的值都會大於中間值，另一區內的值都會小於中間值
- 若每回合可以準確的切成數量相同的兩區，那麼就可以降低走訪每個元素的次數。



# 分兩區的排序

如果每次都要先找中間值的話，那執行時間好像不會比較快？



原始資料

- [4, 3, 2, 5, 1]

第 1 回合

- 找出 [4, 3, 2, 5, 1] 的中間值 3

第 2 回合

- 將 3 移到資料中間，1 和 2 移前面，4 和 5 移後面
- 產生 3 一定在中間的列表: [2, 1, 3, 5, 4] 或 [2, 1, 3, 4, 5] 或 [1, 2, 3, 4, 5] 或 [1, 2, 3, 5, 4]

第 3 回合

- 將 3 前面的資料 [1, 2] 重覆使用分兩區的方式排序
- 將 4 前面的資料 [5, 4] 重覆使用分兩區的方式排序

結束

- 每區的資料只剩一個元素時，就排序完成了。



# 找中間值(pivot)的方法

- 方法 1：固定將資料的第一個元素或是最後一個元素視為中間值
  - 優點：簡單，不需多餘的比較就可決定中間值
  - 缺點：若是原本資料已經是排序好的，那兩端一定是極端值，切出來的兩區內的元素數量一定相差極大，完全無法發揮分兩區排序的優勢。
- 方法 2：隨機將資料內的某個元素視為中間值
  - 優點：簡單，很大機率避開方法 1 的缺點。
  - 缺點：隨機仍代表有機會找到極端值。
- 方法 3：在資料內找三個元素，只從這三個元素中挑出中間值。
  - 優點：避免出現方法 1 的極端情況，也不需要太多額外的操作。
  - 缺點：仍有可能是找到資料內的三個極端值。

# 找中間值(pivot)的方法

- 只有全部元素值看過一次才能保證找到的中間值絕對是中間值，但這不服合要降低排序時間的原則。
- 不論是哪種方法，都有可能找到極端值當中間值，但只要降低找到極端值的機率，就可提昇排序的平均時間。

# 將方法變成程式碼

```
#include <stdio.h>
#define MAX 10

void showList(int number[])
{
    int i;
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);
    printf("\n");
}

int main() {
    int i, number[MAX] = {0};
    for(i = 0; i < MAX; i++)
        scanf("%d", &number[i]);
    printf("Before Sort: ");
    showList(number);
    quickSort(number, 0, MAX-1);
    printf("After Sort: ");
    showList(number);

    return 0;
}
```

18dice

```
void swap(int *x, int *y)
{
    int t;
    t = *x; *x = *y; *y = t;
}

int partition(int number[], int left, int right)
{
    int i = left, j, pivot = right;
    for(j = left; j < right; j++) {
        if(number[j] < number[pivot])
            swap(&number[i++], &number[j]);
    }
    swap(&number[i], &number[pivot]);
    return i;
}

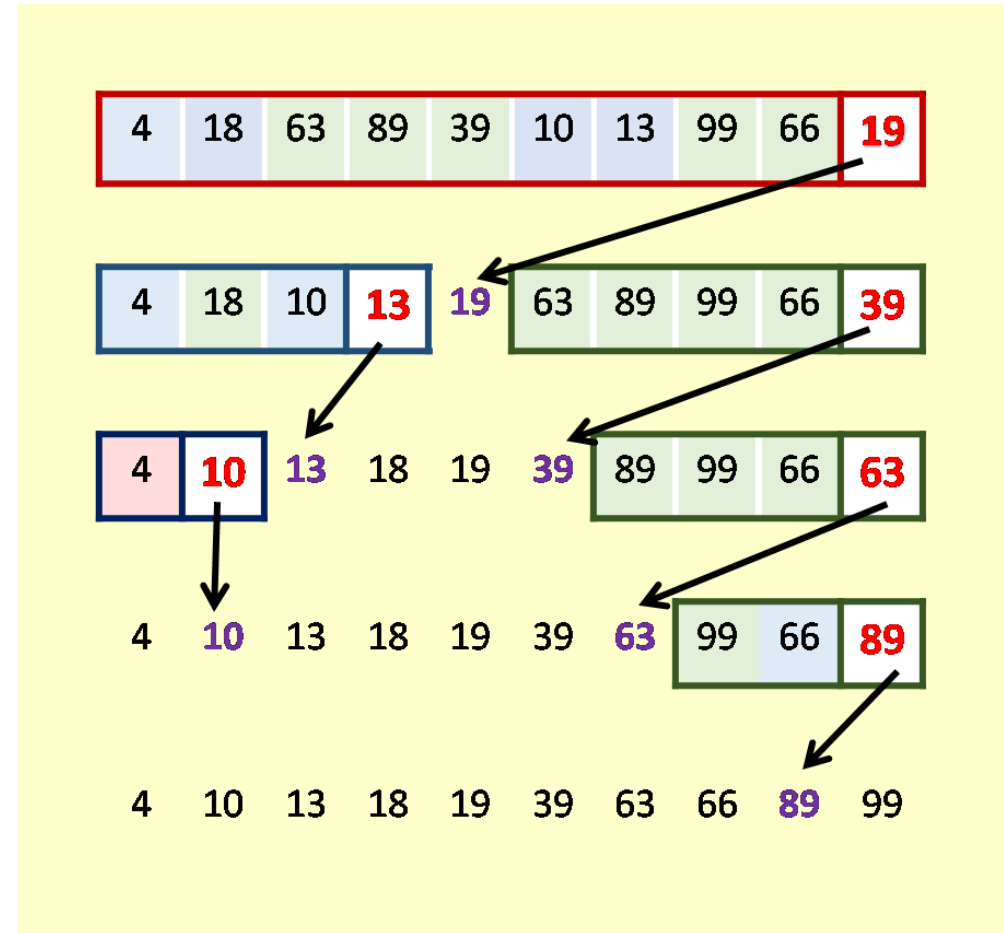
void quickSort(int number[], int left, int right)
{
    int q;
    if(left < right) {
        q = partition(number, left, right);
        showList(number);
        quickSort(number, left, q-1);
        quickSort(number, q+1, right);
    }
}
```

執行結果

```
dice - sort $ ./quicksort.exe
21 20 33 19 2 11 39 10 99 60
Before Sort: 21 20 33 19 2 11 39 10 99 60
21 20 33 19 2 11 39 10 60 99
2 10 33 19 21 11 39 20 60 99
2 10 19 11 20 33 39 21 60 99
2 10 11 19 20 33 39 21 60 99
2 10 11 19 20 21 39 33 60 99
2 10 11 19 20 21 33 39 60 99
After Sort: 2 10 11 19 20 21 33 39 60 99
```

# 這是快速排序法 (quick sort)

- 在資料內找一個元素做為中間值 (pivot)
- 小於中間值的放一區，其餘的放另一區
- 重複找中間值分區的動作，直到每區內的數量只剩一個



# 快速排序法的時間複雜度

---

最差時間複雜度

$O(N^2)$

最佳時間複雜度

$O(N\log N)$

平均時間複雜度

$O(N\log N)$

---

若中間值一直取到極端值，那將會與泡沫排序與選擇排序法有一樣長的執行時間。

當中間值可以平均將資料分成兩群時，整體時間複雜度是  $O(N\log N)$



延伸的概念

# 概念1: 更多排序

## 內部排序：

排序的資料量小，可以完全在記憶體內進行排序。

氣泡排序法 (bubble sort)

選擇排序法 (selection sort)

插入排序法 (insertion sort)

快速排序法 (quick sort)

堆積排序法 (heap sort)

謝爾排序法 (shell sort)

基數排序法 (radix sort)

## 外部排序：

排序的資料量無法直接在記憶體內進行排序，而必須使用到輔助記憶體

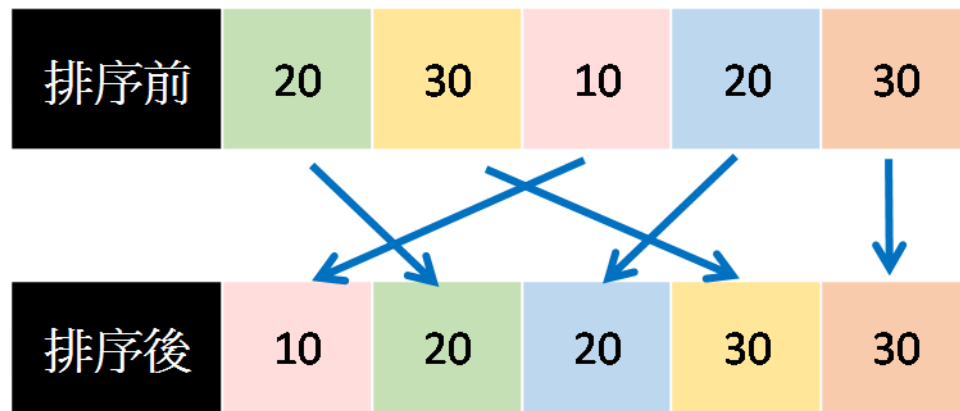
直接合併排序法(Direct Merge Sort)

k路合併法(k-Way Merge)

多相合併法(Polyphase Merge)

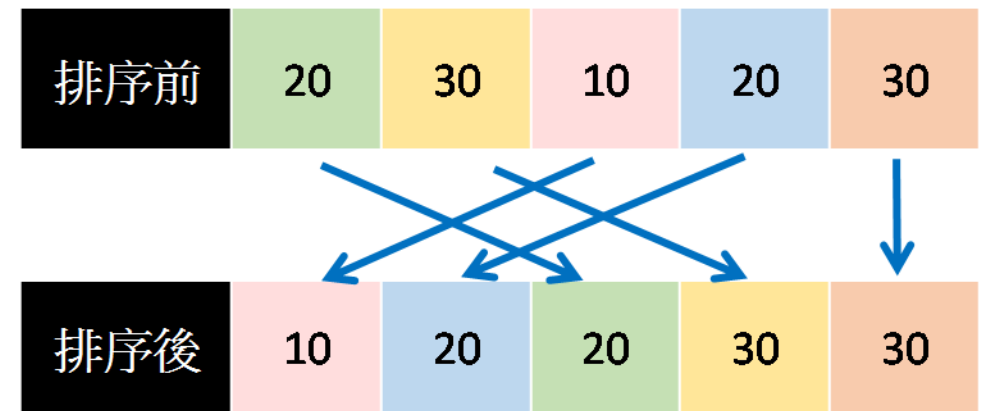
# 概念2: 穩定排序 與 不穩定排序

## 穩定排序 (stable sorting)



- 同樣的值經過排序後，原本在前面的仍會繼續在前面
- 綠色 20 與 藍色 20 排序後的順序仍是綠色 20 在前，藍色 20 在後

## 不穩定排序 (unstable sorting)



- 同樣的值經過排序後，原本在前面的可能會在後面，但也可能繼續在前面
- 綠色 20 與 藍色 20 排序後的順序變成綠色 20 在後，藍色 20 在前



# 概念3: qsort()

```
#include <stdio.h>
#include <stdlib.h> // 亂數相關函數
#include <time.h> // 時間相關函數
#define MAX 10
void showList(int number[])
{
    int i;
    for(i = 0; i < MAX; i++)
        printf("%d ", number[i]);
    printf("\n");
}
int cmp ( const void *a , const void *b )
{
    return *(int *)a - *(int *)b;
}
int main()
{
    int i, array[MAX];
    srand( time(NULL) ); // 設定亂數種子
    for(i = 0; i < MAX; i++) {
        array[i] = rand() % 100 + 1;
    }
    showList(array);
    qsort(array, MAX, sizeof(array[0]), cmp);
    showList(array);
}
```

library: stdlib.h 內提供了 qsort() 函式，可以對陣列變數資料使用 quick sort 方式排序。

執行結果

```
dice - sort $ ./qsort.exe
49 10 8 3 96 92 78 37 86 43
3 8 10 37 43 49 78 86 92 96
```

18dice

# 概念3: qsort()

要排序的  
陣列變數

1

每個元素的記憶  
體大小，可使用  
sizeof() 取得

3

```
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void*))
```

2

陣列內的  
元素數量

4

自訂的比較函式，  
qsort() 內會依照函式  
內定義的規則排序

# 概念3: qsort()

```
qsort(array, MAX, sizeof(array[0]), cmp)
```

```
int cmp(const void *a , const void *b)
{
    return *(int *)a - *(int *)b;
}
```

比較的函式固定會有兩個參數傳入，且型態都會使用 **const void \***，函式內再依照實際要比較的變數型態轉換。