二分搜尋法及其應用

迴圈與陣列
遞迴

丁培毅

二分搜尋的必要性?!

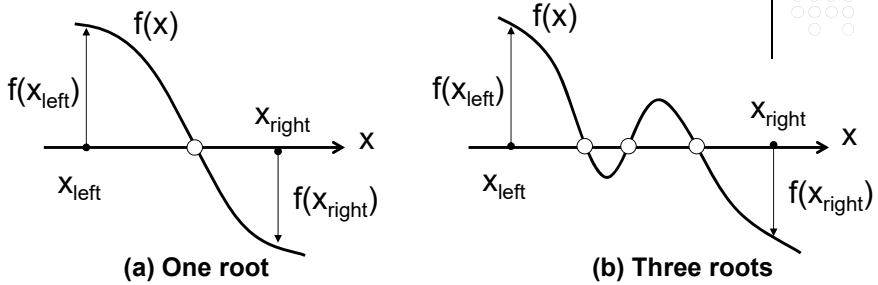
- 最直接的搜尋法當然是線性搜尋, 一個一個元素比對 直到相等或是全部比對完畢為止

```
for (i=0; i<ndata; i++)
   if (target == data[i])
      printf("%d found @ data[%d]\n", target, i);
```

如果 target 可能出現多次 每次都要全部比對完畢

- 不過當陣列裡面的資料已經排好順序時, 沒有必要一個一個慢慢比對, 每一次的比對相當於某一側所有資料的比對, 例如 data[i] < **target** 代表 … < data[i-2] < data[i-1] < **target** **target** < data[i] 代表 **target** < data[i+1] < data[i+2] < …
- 不能任性一點嗎? 雖然資料已經排序好了, 我一個一個去找 難道不行嗎? 請評估上面的迴圈執行的時間 (假設陣列大小 不是問題, 機器 1 秒鐘可以執行 10^9 個指令)
  如果 ndata 是 2^{32}, 需要的時間大概是 1~10 秒
  如果 ndata 是 2^{50}, 需要的時間大概是 3~30 天
  如果 ndata 是 2^{64}, 需要的時間大概是 134~1340 年
- 如果能夠做二分搜尋, 所需要的比對次數分別為 32, 50, 64 次

二分勘根法解 f(x)=0

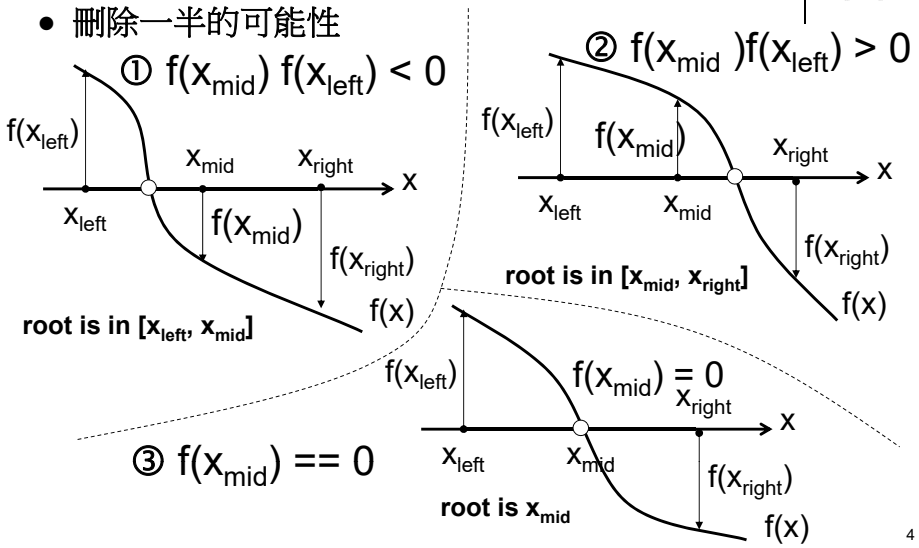


要求誤差小於 ε

- $f(x_{left})$ 和 $f(x_{right})$ 正負號不同代表 在區間 $[x_{left}, x_{right}]$ 內有奇數個根
- 假設在區間 $[x_{left}, x_{right}]$ 裡只有一個實數根
- 暴力解: 把區間切為 $n=(x_{right}-x_{left})/\varepsilon$ 個連續區段, 線性搜尋
- 二分法: n 個區段中只需要評估 $\log_2(n)$ 個
  - $(x_{right}-x_{left})/2^k \approx \varepsilon$ i.e. $k \approx \log_2(n)$

考慮以下三種狀況

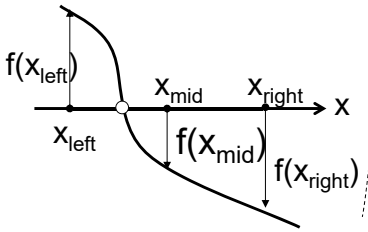
- 把區段 $[x_{left}, x_{right}]$ 切成兩等分
- **刪除一半的可能性**



使用 **while** 迴圈, 每一次將區間一分為二

1. 由 x_{left} 及 x_{right} 計算 $x_{mid} = (x_{left}+x_{right}) / 2$
2. a. 計算 $f(x_{mid})$
   b. if $f(x_{mid}) < 0$, $x_{right} = x_{mid}$
   c. else if $f(x_{mid}) > 0$, $x_{left} = x_{mid}$
   d. else if $f(x_{mid}) == 0$, 根就是 x_{mid}, break

重複以上兩個步驟



令 ε = 1.0e-10

```
01 while (1)
02 {
03     x_mid = (x_left + x_right) / 2.0;
04     if (fabs(f(x_mid)) < 1.0e-10)
05         break;
06     else if (f(x_left) * f(x_mid) < 0.0)
07         x_right = x_mid;
08     else // if (f(x_right) * f(x_mid) < 0.0)
09         x_left = x_mid;
10 }
```


運用額外的變數去除重複的運算

- 前面這一段程式裡, 每一個點的 f() 函數值都計算了三次, 其中兩次是 $f(x_{mid})$, 一次是在下一次迴圈執行時呼叫 $f(x_{left})$ 或是 $f(x_{right})$
- 如果函數 f() 的計算需要相當長的時間, 其實是有點浪費的, 可以運用額外的變數把計算過的函數值紀錄起來, 下一次要用到時就不用再重算了

```
01 f_left = f(x_left);
02 f_right = f(x_right);
03 while (1) {
04     x_mid = (x_left + x_right) / 2.0;
05     f_mid = f(x_mid);
06     if (fabs(f_mid) < 1.0e-10)
07         break;
08     else if (f_left * f_mid < 0.0) {
09         x_right = x_mid;
10         f_right = f_mid;
11     }
12     else /* if (f_right * f_mid < 0.0) */ {
13         x_left = x_mid;
14         f_left = f_mid;
15     }
16 }
```


基本的遞迴函式設計方法

- 基本步驟

a. 定義出遞迴函式以及其參數 (遞迴函式一定需要參數)
b. 想清楚這個遞迴函式在某一參數時能夠解的問題是什麼
c. 把需要解決的問題拆解為較小的問題
d. 呼叫遞迴函式解決這個小問題, 並且用這個答案組合出原來問題的答案
e. 問題縮到最小時 (base case) **可以直接寫出答案**

- 例如: 計算陣列 data[] 裡 n 個元素 data[0]~data[n-1] 的總和
  a. int sum(int data[], int n)
  b. 這個函式能夠計算並回傳 data[0]+data[1]+…+data[n-1]
  c. 拆解為小一點的問題: n-1 個元素的總和
  d. sum(data, n-1) 可以算出 data[0]+…+data[n-2], 所以 sum(data, n) 的結果應該是 sum(data, n-1) + data[n-1]
  e. sum(data, 0) 的結果是 0

求陣列裡所有元素總和的遞迴函式

```
int sum(int data[], int n) {
  if (n==0)
    return 0;
  else
    return sum(data, n-1) + data[n-1];
}
```

- 前面這個範例裡，呼叫 sum(data, n) 以後會依序呼叫 sum(data, n-1), sum(data, n-2), …, sum(data, 0) 共 n+1 次遞迴函式呼叫，遞迴深度 n+1, 才能夠計算出答案
- 在撰寫遞迴函式時，前面這樣的問題拆解方式是比較沒有效率的，需要 O(n) 次的函式呼叫，在許可的情況下應該**盡量尋找呼叫次數少一些的問題拆解方法**，例如在計算陣列 data[] 裡 n 個元素 data[0]~data[n-1] 的總和時，可以考慮拆成下面兩個子問題：
  計算 **data[0]~data[(n-1)/2]** 的總和以及
  計算 **data[(n+1)/2]~data[n-1]** 的總和
  最後把兩者加總起來

遞迴的二分勘根法

❶❷

```
01 double findRoot(double x_left, double x_right, double eps) {
02     double x_mid = (x_left + x_right) / 2.0;
03     double f_mid = f(x_mid);
04     double f_left = f(x_left);
05     if (fabs(f_mid) < eps))
06         return x_mid;                          ❺
07     else if (f_left * f_mid < 0.0)
08         return findRoot(x_left, x_mid, eps);
09     else // f_mid * f_right < 0.0              ❸❹
10         return findRoot(x_mid, x_right, eps);
11 }
```

```
01 f_left = f(x_left);
02 f_right = f(x_right);
03 while (1) {
04     x_mid = (x_left + x_right) / 2.0;
05     f_mid = f(x_mid);
06     if (fabs(f_mid) < 1.0e-10)
07         break;
08     else if (f_left * f_mid < 0.0) {
09         x_right = x_mid;
10         f_right = f_mid;
11     }
12     else /*if (f_right * f_mid < 0.0)*/ {
13         x_left = x_mid;
14         f_left = f_mid;
15     }
16 }
```


二分搜尋 (Binary Search)

- **Iterative**

```
int binarySearch(int target, int data[], int left, int right) {
    int mid;
    while (left <= right) {
        mid = (left+right)/2;
        if (target == data[mid]) return mid;
        else if (target > data[mid]) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

left mid right

2	3	4	5	6	8	13	16	18	23	24

data[]

find the last element <= target i.e. U-1

```
int mid, result = left-1;
while (left <= right) {
    mid = (left+right)/2;
    if (data[mid]<=target)
        left = mid + 1,
        result = mid;
    else
        right = mid - 1;
}
return result;
```

- **Recursive**

❶❷

```
int binarySearch(int target, int data[], int left, int right) {
    int mid = (left+right)/2;
❺   if (left > right) return -1;
    if (target == data[mid]) return mid;
❸❹  else if (target > data[mid]) return binarySearch(target, data, mid+1, right);
    else return binarySearch(target, data, left, mid-1);
}
```



```
// lower_bound()
int mid, result = right+1;
while (left <= right) {
    mid = (left+right)/2;
    if (data[mid] < target)
        left = mid + 1;
    else // >= target
        result = mid,
        right = mid - 1;
}
return result;
```

```
// upper_bound()
int mid, result = right+1;
while (left <= right) {
    mid = (left+right)/2;
    if (data[mid] <= target)
        left = mid + 1;
    else // > target
        result = mid,
        right = mid - 1;
}
return result;
```

核心

```
int mid;
while (left <= right) {
    mid = (left+right)/2;
    if (data[mid] < target)
        left = mid + 1;
    else
        right = mid - 1;
}
```

```
int mid, result = left-1;
while (left <= right) {
    mid = (left+right)/2;
    if (data[mid] <= target)
        result = mid,
        left = mid + 1;
    else
        right = mid - 1;
}
return result;
```

upper_bound()-1

```
int mid, result = left-1;
while (left <= right) {
    mid = (left+right)/2;
    if (data[mid] < target)
        result = mid,
        left = mid + 1;
    else
        right = mid - 1;
}
return result;
```

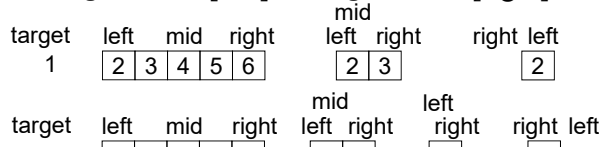
lower_bound()-1

分析

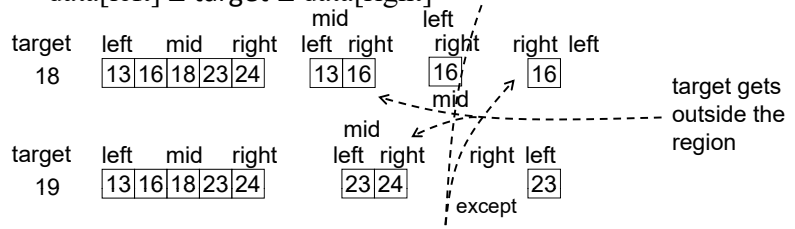
data[]

left								mid		right
2	3	4	5	6	8	13	16	18	23	24

- binary_search(int left, int right, int data[], int target)
- target < data[left] or target > data[right]



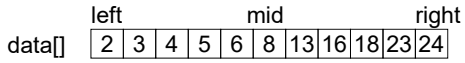
- data[left] ≤ target ≤ data[right]



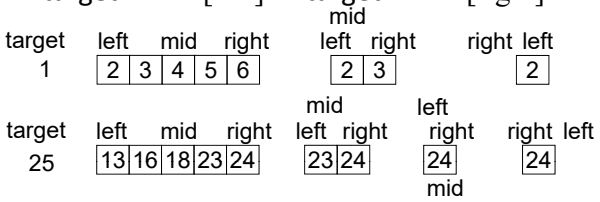
- **the region shrinks to half its size for each iteration, O(log n)**

```
int mid;
while (left <= right) {
    mid = (left+right)/2;
    if (data[mid] < target)
        left = mid + 1;
    else
        right = mid - 1;
}
```

分析

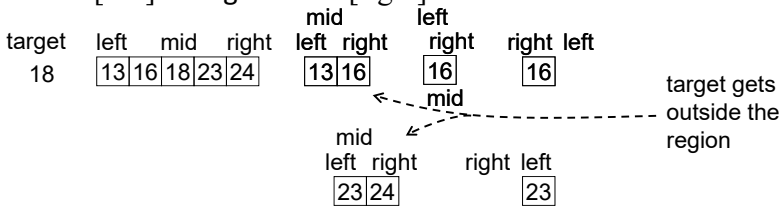


- binary_search(int left, int right, int data[], int target)
- target < data[left] or target > data[right]




```
int mid;
while (left <= right) {
   mid = (left+right)/2;
   if (data[mid] <= target)
      left = mid + 1;
   else
      right = mid - 1;
}
```

- data[left] ≤ target ≤ data[right]



- **the region shrinks to half its size for each iteration, O(log n)**

Applications

- LeetCode 34 Find First and Last Position of Element in Sorted Array
  - lower_bound() and (upper_bound()-1)
- APCS 2022/06 #3 雷射測試 (ZJ i401)
  - upper_bound() and (lower_bound()-1)
- LeetCode 744 Find Smallest Letter Greater Than Target
  - lower_bound()
- LeetCode 611 Valid Triangle Number
- LeetCode 2824 Count Pairs Whose Sum is Less than Target
- LeetCode 2563 Count the Number of Fair Pairs
- Other binary search problems on LeetCode : https://leetcode.cn/circle/discuss/SqopEo/

LC 611 Valid Triangle Number

- Given n sticks with length b_0, b_1, …, b_{n-1}, find the number of valid triangles that can be formed from any 3 of them
- Brute-force method would takes $O(n^3)$ to check every possible combination (a,b,c) for a+b>c, a+c>b, or b+c>a
- O(nlogn) procedure that uses qsort() and upper_bound():
  1. sort the length sequence b_0, b_1, …, b_{n-1} with qsort()
  2. for each b_i, 2≤i≤n-1, use it as the longest side of the triangle

     Let c=0, L=0, U=i-1 (there are at most (i-1)*(i-2)/2 possibilities)

     while ($L<U$)

     a. find smallest $L^* \in [L, U-1]$ s.t. $b_{L^*}+b_U > b_i$, c+=$U-L^*$, $L=L^*$

     b. find largest $U^* \in [L+1, U-1]$ s.t. $b_L+b_{U^*} \le b_i$

     for each $U \in [U^*+1, U-1]$, c+=$U-L$

     $U=U^*$, $L=L+1$

upper_bound() >t
r_lower_bound() <=t
O(log(U-L))≈O(log n)

LC 2824 Count Pairs Whose Sum is Less than Target

- Given an integer array nums of length n and an integer target, return *the number of pairs* (i, j) *where* 0<=i<j<n *and* nums[i] + nums[j] < target
  1. sort the array nums with qsort()
  2. find the smallest nums[i] , 0≤i≤n-1, s.t. target <= nums[i]

     Let c=0, L=0, U=i-1 (there are at most (i-1)*(i-2)/2 possibilities)

     while (L<U)

     a. find smallest $L^* \in [L, U-1]$ s.t. $b_{L^*}+b_U > b_i$, c+=$U-L^*$, $L=L^*$

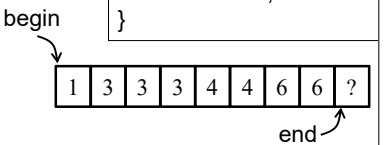
     b. find largest $U^* \in [L+1, U-1]$ s.t. $b_L+b_{U^*} \le b_i$, c+=U^*+1-L, $U=U^*+1$

upper_bound() O(log(U-L))≈O(log n)

和 STL 一致的界面

```
// find the first element >= target  in [begin, end)
int *lower_bound(int *begin, int *end, int target) {
    int mid, *result=end;
    while (begin<end) {
        mid = (end-begin)/2;
        if (*(begin+mid)>=target)
            result = end = begin+mid;
        else
            begin += mid+1;
    }
    return result;
}
```

```
// find the first element > target in [begin, end)
int *upper_bound(int *begin, int *end, int target) {
    int mid, *result=end;
    while (begin<end) {
        mid = (end-begin)/2;
        if (*(begin+mid)>target)
            result = end = begin+mid;
        else
            begin += mid+1;
    }
    return result;
}
```




```
int data[] = {1,3,3,3,4,4,6,6};
L = lower_bound(data, data+8, 3);
U = upper_bound(data, data+8, 3);
```

Results are 1 and 4

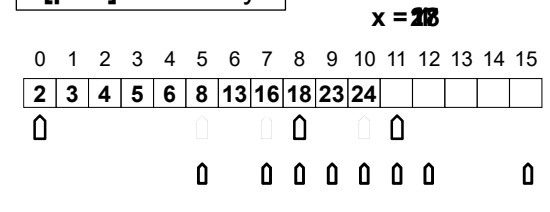
另一種二分搜尋寫法

在 **a[0]** 到 **a[n-1]** 間搜尋 **x** 的位置
回傳 ≥ **x** 最小元素的位置

```
int jump_search(int a[], int n, int x) {
    if (a[0]>=x) return 0; // check the first
    int pos=0; // the current position
    for (int jump=n/2; jump>0; jump/=2) {
        while (pos+jump<n && a[pos+jump]<x)
            pos += jump;
    }
    return pos+1;
}
```

逐步二分縮小搜尋範圍

a[pos] < x always

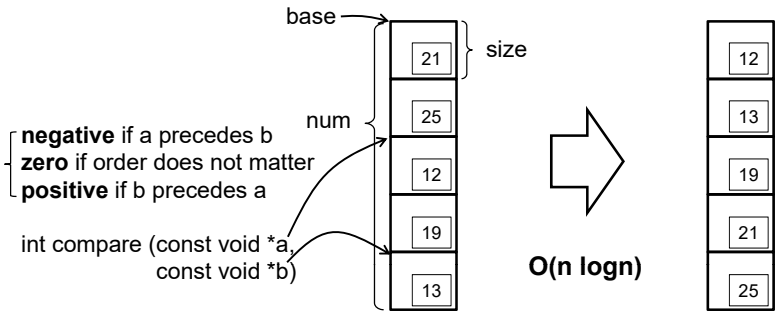


Sorting an array of data

stdlib

- void **qsort**(void* **base**, size_t **num**, size_t **size**, int (***compare**)(const void*,const void*));

quick sort: divide the data into approximately a half at each step, and sort them individually

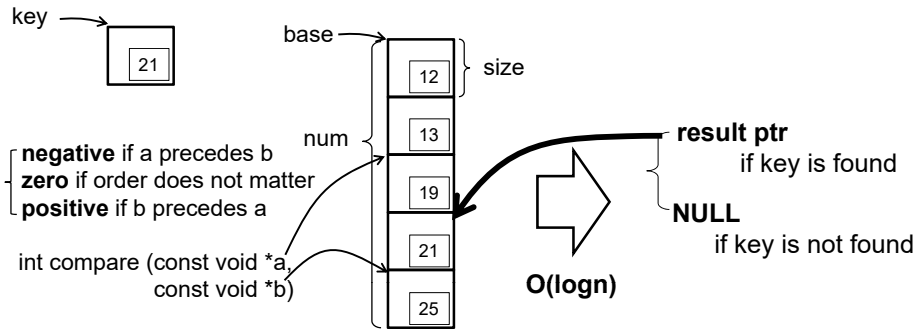


Searching data in a sorted array

stdlib

- const void* **bsearch**(const void ***key**, void* **base**, size_t **num**, size_t **size**, int (***compare**)(const void*,const void*));

Binary search : divide the sorted data into approximately a half at each step, and eliminate one of them with a single comparison



Find Duplicated Number (LC287)

- 有一個整數陣列 data 裡有 n 個數字, 這些數字的範圍在 1~n-1 之間, 1≤n≤500, 其中只有一個數字出現多次, 在下面限制下請分別撰寫函式 int findDuplicate(int n, int data[]), 在**不修改原來陣列**的情況下, 找到並且回傳這個重複出現的數字
  - 不使用額外的陣列, 程式執行 $O(n^2)$ 次比對
  - 使用額外的 n 個元素陣列, 程式執行 O(n) 次比對

```
01 int findDuplicate(int n, int data[]) {
02    for (int i=0; i<n; i++)
03       for (int j=i+1; j<n; j++)
04          if (data[i]==data[j]) return data[i];
05    return -1;
06 }
```

```
01 int findDuplicate(int n, int data[]) {
02    int i, count[500] = {0};
03    for (i=0; i<n; i++)
04       if (count[data[i]] == 1)
05          return data[i];
06       else
07          count[data[i]] = 1;
08    return -1;
09 }
```

data[ ]	1	3	2	2	6	8	2	7	10	2	9

n=11, 重複的數字: 2,
沒有出現的數字: 4, 5

Duplicated Number (cont'd)

- 不使用額外的 陣列, 程式執行 $O(n \log_2 n)$ 次比對

找出一個方式, 一次刪除一半的可能解

不搜尋 data 陣列而是搜尋所有可能的數值 1~n-1, 因為是連續的自然數, 所以不需要使用額外的陣列來記錄

1	2	3	4	5	6	7	8	9	10

left mid right

```
01 int findDuplicate(int n, int data[]) {
02    int left=1, right=n-1, mid, i, count;
03    while (left < right) {
04       mid = (left+right) / 2;
05       for (i=0, count=0; i<n; i++)
06          if (data[i] <= mid) count++;
07       if (count > mid) right = mid;
08       else left = mid + 1;
09    }
10    return left; //  假設一定有一個數字重複
11 }
```

data[]	1	3	2	2	6	8	2	7	10	2	9

n=11, 重複的數字: 2, 沒有出現的數字: 4, 5

findDuplicate(n, data, 1, n-1);

- 遞迴

```
❶❷01 int findDuplicate(int n, int data[], int left, int right) {
   02    int mid=(left+right)/2, i, count;
   03 ❺ if (left == right) return left;
   04    for (i=0, count=0; i<n; i++)
   05       if (data[i] <= mid) count++;
   06 ❸❹ if (count > mid) return findDuplicate(n, data, left, mid);
   07    else return findDuplicate(n, data, mid+1, right);
   08 }
```


Find Minimum of Sorted and Rotated Sequence (LC153)

- 有一個排序過的陣列, 0 1 2 4 5 6 7, 被旋轉過變成 4 5 6 7 0 1 2, 請用 $O(\log_2 n)$ 的演算法找尋其中的最小值的位置 (假設陣列中資料沒有重複, 有重複的話會比較複雜一點)
- $O(\log_2 n)$ 的演算法需要是二分法, 不過還是需要仔細確認一下是不是可以每次刪除一半的可能性
  - int data[]={4,5,6,7,0,1,2},left=0,right=7,mid=(left+right)/2;
  - 一開始會滿足 data[left]>data[right], 如果發現 data[left]<data[right], 那麼data[left]一定是最小的
  - 如果 data[left]<data[mid] 則最小值在 mid ~ right 中間
  - 如果 data[left]>data[mid] 則最小值在 left ~ mid 中間

每一次比對都可以把可能的範圍縮小一半

逆序數計算 APCS 2018/06/09 #4 反序數量

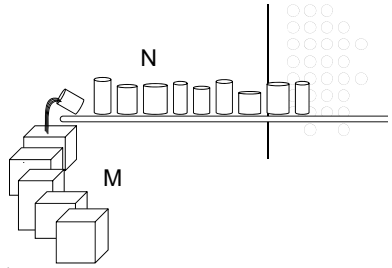
- 在數列 A = (3, 1, 9, 8, 9, 2) 中，共有 (3, 1)、(3, 2)、(9, 8)、(9, 2)、 (8, 2)、(9, 2) 6 個逆序對，逆序數 W(A) 是 6
- 分治法 (divide-and-conquer)
  1. 將 A 分為前後兩個數列 X 與 Y，其中 X 的長度是 n/2
  2. 遞迴計算 W(X) 和 W(Y)
  3. 計算 W(A) = W(X) + W(Y) + S(X, Y)
     其中 S(X, Y) 是由 X 中的數字與 Y 中的數字所構成的反序數量
     注意這個數量和 X 或是 Y 序列中數字的順序無關
- 例如： A = (3, 1, 9, 8, 9, 2)

  $O(n (\log n)^2)$
  1. 將 A 分為兩個數列 X = (3, 1, 9) 與 Y = (8, 9, 2)
  2. 遞迴計算得到 W(X) = 1 和 W(Y) = 2
  3. 計算 S(X, Y) = 3, 亦即 (3, 2), (9, 8), (9, 2)
     計算 W(A) = W(X) + W(Y) + S(X, Y) = 1 + 2 + 3 = 6
- 直接計算 S(X, Y) 需要 $O((n/2)^2)$，可以排序後用二分搜

最小化最大值、最大化最小值、最大/小化平均值

- 最小化最大值:
  - POJ 1505 Copying Books UVa 714
  - POJ 3273 Monthly Expense
  - Leetcode 1231 Divide Chocolate
  - UVa 11413 Fill the Containers
  - 基地台
  - 公平分區
- 最大化最小值:
  - POJ 3258 River Hopscotch
  - POJ 2456 Aggressive cows ZJ e616
- 最大化平均值:
  - POJ 3111 K Best
  - c904. 天龍國的蜜蘋果
  - POJ 2976 Dropping tests
  - POJ 1064 Cable master
  - POJ 3122 Pie

Fill the Container



- 酪農每天採集生乳以後送到收集站, 總共收集了 N 瓶的生乳按順序放在輸送帶上, 在收集站要合併到 M 個比較大的容器裡 (M 是一個預先指定的數字), 每個收集生乳的瓶子的容積都不大一樣
  - 一瓶生乳完全倒入容器後才能換下一瓶
  - 一個瓶子裡面的生乳只能倒入單一容器中, 如果容器還有空間但不夠裝完某一瓶生乳的話, 就只能把目前的容器封起來, 換下個容器繼續裝, 當然也不能打開先前封起來的容器繼續填裝
- 因為只能使用 M 個容器來裝這 N 瓶生乳, 所以容器不能太小, 如果最大的那個容器的容積不夠大的話, 有可能沒有辦法用 M 個容器依照前面的規則裝完所有 N 瓶生乳, **請寫一個程式計算最大容器的容積應該要大於多少?**
- 範例：收集了 5 瓶生乳 (1,2,3,4,5), 要倒入 3 個容器裡, 最大容器的容積至少要 6(=1+2+3) 才能滿足上述條件, 例如 第 1 瓶,第 2 瓶,第 3 瓶倒入第一個容器, 第 4 瓶 倒入第二個容器, 第 5 瓶 倒入第三個容器

- 這個題目並沒有要求你把每一個容器的容積都算出來, 程式計算的時候可以假設每一個容器的容積都相同, 例如 (6,6,6), 但是實際上是可以比較小的, 例如 (6,4,5)
- 這個問題等於是要把所有 N 瓶的生乳分成 M 組, 每一組只包含相鄰的瓶子, 也允許一組完全不包含任何瓶子, 第一組倒入第一個容器, 第二組倒入第二個容器, ... 如此每一種分組方法都會有一個容器裡面裝最多的生乳, 題目就是要找到一種分組方法, 使得最大的容器可以越小越好
- 暴力的方法就是把所有可能的分組都一一列出, 這可以用數數字的方法來完成, 例如上面 M=3, N=5 的例子中, (0,0,5), (0,1,4), (0,2,3), (0,3,2), (0,4,1), (0,5,0), (1,0,4), (1,1,3), (1,2,2), (1,3,1), (1,4,0), (2,0,3), ..., 然後把每一種分組方法中最大容器的容積算出, 最後找出最大容器的容積最小的那一組,當 M, N 很大時暴力的方法執行起來需要非常多時間
- 換一個角度由我們要的答案的範圍來想, 最大容器的容積 V 需要大於或等於最大的生乳瓶子的容積, 同時需要小於所有生乳容積的總和, 假設 M 個容器的容積都是 V, 我們可以測試是否有辦法把 N 瓶生乳依照前述規則到入 M 個容器中, 如果不需要 M 個容器就已經把所有瓶子的生乳裝完了, 就表示 V 值太大了, 如果裝不下，就表示需要的容積值大於等於 V

- 我們可以一個一個數值來測試, 但是因為在指定的連續區間裡, 如果 V 容積足夠大, 所有大於 V 的容器也都夠大, 如果 V 值太小, 所有小於 V 的容器也都太小, 所以可以用**二分法**, 寫迴圈或是**遞迴**來搜尋, 程式的執行速度才能達到題目要求的 $O(\log_2 n)$
- 在實作二分搜尋時, 可以寫一個函式測試指定的 V 值是否夠大, 太小的話沒有辦法把 N 瓶生乳分成 M 組, 且每一組的容積和都小於等於 V

```
int isVFeasible(int N, int vessels[], int M, int V) {
    int i, j, amount;
    for (i=j=0; i<M && j<N; i++) { // 第 i 個容器, 第 j 瓶生乳
        amount=0;
        while ((j<N)&&(amount+vessels[j]<=V))
            amount += vessels[j++];
    }
    return j==N; // 代表所有 N 瓶生乳都分配完了, V 值不會太小
}
```

- 如果上述測試中 V 值太小 (j<N), 則可能的 V 值在 mid+1 到 right 中
  如果上述測試中 V 值不會太小 (j==N), 則可能的 V 值在 left 到 mid 中
- 這個題目如果不要求每一組只包含相鄰的生乳瓶子, 就會變成背包問題的變形, 變成有 M^N 種可能的分組, 就不是用二分法可以解的了

Copy the Books

- 影印店有 K 個員工, 有 K 台機器可以同時運作, 現在要拷貝一系列 M 本書(1≤K≤M≤500), 每本書的頁數不見得相同
  - 一本書只能分配給一個員工來拷貝
  - 每個員工分配到的書都是系列中連續的
- 拷貝所需要的時間和分配到的總頁數成正比, M 本書拷貝完成的時間隨著分配到頁數總和最多的那個員工而定, 現在希望能夠在最短時間內完成這 M 本書的拷貝, 請**寫一個程式分配工作給員工1, 員工2, ..., 員工K**。 如果有多組分配方法, 希望編號比較小的員工分配到的頁數越少越好, 但每個人至少要分配到一本書
- 這一題基本上和前一題是相同的, 只是程式需要輸出分割方法, 同時有多種分配方法時, 需要滿足額外的條件
- 基本上也是用二分法尋找分配到的頁數的最大值

Median of 2 Sorted Lists

- 有兩個資料已經排好順序的陣列, 請寫一個程式有效率地算出兩個陣列合併起來的「中數」
- 例如: n1=12, data1[] 1 3 5 6 8 **10** 11 13 16 19 20 25 , 中數為 10 (data1[median1])

  n2=6, data2[] 2 4 **6** 7 9 17 , 中數為 6 (data2[median2])

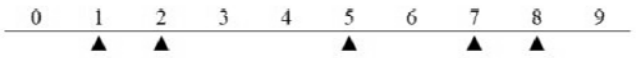
  合併的數列: 1 2 3 4 5 6 6 7 **8** 9 10 11 13 16 17 19 20 25, 中數為 8
- 暴力解: ❶全部合併起來再找第 (n1+n2)/2 個元素 ❷兩個數列裡比較小的中數定義了一個下限, 上例中data1[]的6和data2[]的6, 由這個地方開始合併, 直到合併數列的第 (n1+n2)/2 個元素為止
- 為什麼說是暴力解, 因為是已經排好順序的兩個數列, 應該可以更快, 如果不幸在合併的數列中 data2[median2] 和真正的中數之間差了 2^{20} 個元素, 一個一個合併就很浪費計算機時間了

Median of 2 Sorted Lists (cont'd)

- 以上例來說, 合併數列的中數會在 data1[] 的 6 到 10 中間, 或是在 data2[] 的 6 到 9 中間, data1 和 data2 數列都是已經排好順序的 ⇒ **二分法**
- **關鍵**在如何快速確定某一個數字在合併的數列中是在中數之前還是中數之後,
  例如 data2[] 中的 7:
  - ❶ 7 在 data2[] 裡前面有 3 個數字,
  - ❷ 如果是合併數列的中數, 前面需要有 (12 + 6)/2-1=8 個數字, 也就是說在 data1[] 裡應該要有 8-3 = 5 個數字小於或是等於 7, 但是 data1[4] 為 8, 所以小於 7 的數字最多 4 個
  - ❸ 7 不是合併數列的中數, 7 比中數小, 需要往後找

  同樣原理, data1[] 數列中的每一個數也都可以快速判斷是否為中數, 比中數大或是比中數小

基地台

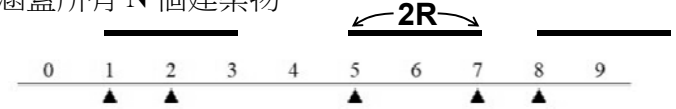


範例一輸入	範例二輸入
5 **2** 5 1 2 8 7	5 **1** 7 5 1 2 8
範例一輸出	**範例二輸出**
3	**7**

- 上圖代表一條直線道路, 圖中 1,2,5,7,8 的位置上有建築物, 希望架設 **K** 個服務半徑 **R** 的基地台來滿足所有 **N** 個建築物的需求
- 例如：
  - 假設 **K** 為 1, 則基地台應該架設在座標 4.5 的位置, 所有建築物與基地台的距離都在半徑 3.5 以內, 因此基地台的最小服務直徑 **2R** 需要是 **7**
  - 假設 **K** 為 2, 在 0.5 到 2.5 之間架設一個基地台來服務位置 1 與 2 的建築物, 另一個基地台架在 6.5 的位置, 服務位置 5,7,8 的建築物, 基地台的最小服務直徑 **2R** 需要是 3
  - 在 **K** 為 3 時, 基地台的最小服務直徑 **2R** 需要是 1
- 對於指定的 **K** 值,請找出架設時最短的基地台服務直徑 **2R**

問題分析

- 這個問題乍看之下有點像是叢集 (clustering) 分群的問題
  - 仔細想一下, 發現它的距離定義和中心的定義使得很難找到像是 k-means 的演算法來逐步找到各個子群, 也沒有用到各群連續的特點
- 不要由資料分群的角度來想, 直接看成**尋找基地台涵蓋直徑**的問題, 其實這個要尋找的直徑 **2R** (題目限制為整數)
  - 有**下限** 0 (如果 **K** 值 ≥ 所有位於不同位置的建築物數, **K** < **N**)
  - 也有**上限**⌈**T/K**⌉(**K** 個基地台沒有重疊的 **2RK** 服務範圍涵蓋全部 0 到 **T** 區間)
- 指定一個 **R** 值, 我們可以很容易地判斷 **K** 個基地台是否可以涵蓋所有 **N** 個建築物



- 線性搜尋的話需要花的時間正比於 **T**, 二分搜尋則只需要花正比於 $\log_2$ **T** 的時間

實作

第 1 子題組 10 分, 座標範圍 T 不超過 100, 1<=K<=2, K<N<=10
第 2 子題組 20 分, 座標範圍 T 不超過 1000, 1<=K<N<=100
第 3 子題組 20 分, 座標範圍 T 不超過 1000000000, 1<=K<N<=500
第 4 子題組 50 分, 座標範圍 T 不超過 1000000000, 1<=K<N<=50000

- 要在 **[0,⌈T/K⌉]** 範圍內所有整數中搜尋 **R**，有兩種方法：
  1. 做一個大小為⌈T/K⌉的陣列裡面放 0,1,2,…,⌈T/K⌉，撰寫一個比對的函式判斷答案是否小於指定的數值，然後使用 std::lower_bound()
     - 大部分時候使用 std::lower_bound() 是一個簡單有效的方法，但是第 3,4 子題中 T/K 的數值要求 500MB 以上的記憶體，就不能使用 std::lower_bound() 這種模組化的二分搜尋法了
  2. 類似下面的二分勘根法一樣寫一個二分的迴圈

```
01 f_left = f(x_left), f_right = f(x_right);
02 while (x_right-x_left > 1.0e-10) {
03     x_mid = (x_left + x_right) / 2.0, f_mid = f(x_mid);
04     if (f_left * f_mid < 0.0)
05         x_right = x_mid, f_right = f_mid;
06     else
07         x_left = x_mid, f_left = f_mid;
08 }
```


實作 (續)

- 由於建築物的位置不見得依照順序、也可能相同，所以先要寫一小段程式處理，找到位置不同的建築物有幾個，同時也將位置陣列依照順序排好

```
std::sort(p, p+N);
for (distinctN=i=1; i<N; i++)
   if (p[i]!=p[i-1]) distinctN++;
```

- 每個建築物都有自己的基地台或是每個位置都有自己的基地台 ⇒ **R** 為 0

```
if ((distinctN<=K)||
    (p[N-1]-p[0])<K) {
   printf("0\n");
   return 0;
}
```

- 二分法

```
01 left = 1.0, right = ((double)(p[N-1]-p[0]))/K;
02 while (right-left>1.0-1e-10) {
03     mid = (left+right)/2.0;
04     if (coverAll((int)mid, N, K))
05         right = mid;
06     else
07         left = mid;
08 }
09 printf("%d\n", (int)right);
```


實作 (續)

- 檢查是否 K 個直徑 d 的基地台能夠服務全部 N 棟建築物

```
int coverAll(int d, int N, int K) {
   int i, *start=p;
   for (i=0; i<K; i++) {
      start = std::upper_bound(start, p+N, *start+d);
      if (start>=p+N) return 1;
   }
   return 0;
}
```

- 不論陣列裡是否有相同的元素，std::**upper_bound**(start, end, target) 可以找到陣列 [start, end) 中第一個大於目標數值 target 的元素

遮住圍牆空隙的木板

- 假設你的圍牆上有一些木條在強風中被吹落了，圍牆上原本有 n 條長條，用 $x_0, x_1, \ldots, x_n, x_i \in \{0,1\}$ 來表示目前有那些長條是不見的 (x_i 為 0) ，現在我們要拿 k 條相同寬度的木板來暫時遮住這些縫隙，請問木板的寬度需要是多少才能用 k 條木板遮住所有的縫隙。
- 如果可以用 k 條寬度 w 的木板遮住，那麼更寬的木板一定可以遮住；反之如果 k 條寬度 w 的木板遮不住，比較窄的木板一定是遮不住的。
- 1<=w<=n

公平分區

- **問題描述**
  - 甲國是一個國土狹長的國家, 甲國的 **N 個城鎮**都沿著一條筆直的大道均勻分布。我們可以把這些城鎮看成在一條直線上編號 1,2,...,N 的點, 相鄰兩點的距離都是 1。每個城鎮擁有若干人口, 第 i 個城鎮的**人口數以 P[i]** 表示, 另外我們用 [i,j] 表示包含從 i 到 j 這些城鎮所形成的連續區域, 其中 i≤j。
  - 因為國土狹長, 甲國人希望能夠將國土**劃分層級**以便分區治理, 但是也不希望劃分的層級太多, 因此有個**層級的上限 K**。一個行政區是某個連續區域 [i,j], 全國是一個第 0 級行政區 [1, N]。行政區的劃分原則如下：

- 對於第 H 級行政區 [S,T], 如果滿足 H < K 且至少包含三個城鎮(T-S ≥ 2), 則此行政區中就會選出一個城鎮 P, S < P < T, 在城鎮 P 設置行政機構並將這行政區劃分成左右兩個 (H+1) 級的非空行政區域 [S,P-1] 與 [P+1,T]。城鎮 P 的選擇方式是必須使得 [S,P-1] 與 [P+1,T] 左右兩個行政區人民到達 P 城鎮的總距離差異越小越好, 如果有兩個城鎮可以選擇, 會選擇編號較小的城鎮。
  舉例來說，甲國的各城鎮人口如右表：

編號	1	2	3	4	5	6	7
人口數	2	4	1	3	7	6	9

- 編號 1234567 人口數 2413769 全國 [1,7] 是一個行政區, 我們可以找到 P=5, 左邊區域 [1,4] 的人民到達 P 的距離總和是：2×(5-1)+4×(5-2)+1×(5-3)+3×(5-4) = 25, 而右邊區域 [6,7] 的人民到達 P 距離總和是：6×(6-5)+9×(7-5) =24。總距離差異為 |25-24| = 1, 這個差值是所有可能選擇 P 城鎮中最小的, 所以甲國人會在 P=5 設置行政機構將全國劃分成 [1,4] 與 [6,7] 兩個 1 級行政區。假設 K=1，則兩個 1 級行政區都不再考慮劃分, 編號 5 的城鎮是唯一設置行政機構的城鎮。

現在假設人口數如上但 K=3。那麼就必須考慮那兩個 1 級行政區是否可進一步劃分。其中 [6,7] 不能再劃分了, 因為至少要包含 3 個城鎮才能劃分。但是左邊的 [1,4] 可以進一步劃分, 若以 P=2 劃分成 [1,1] 與 [3,4] 兩區, 兩邊的距離總和分別是：2×(2-1) = 2 以及 1×(3-2)+3×(4-2) = 7, 差異是 5；而若以 P=3 劃分成 [1,2] 與 [4,4], 則兩邊的總距離差異剛好也是 5。兩處條件一樣, 因此會選擇編號較小的 P=2 來設置行政機構。即使分層的上限 K=3 還沒超過, 但所劃分的 [1,1] 與 [3,4] 兩個 2 級行政區都不能再劃分了。所以這個例子中，設置行政機構的城鎮為 5 與 2。

給定甲國各個城鎮的人口數, 請幫忙找出甲國人**如何劃分行政區**, 計算出所有設置行政機構的城鎮人口總數。計算過程中，總距離可能超過 231 但不會超過 260。

- **輸入測試資料一**:

  7 1↵
  2 4 1 3 7 6 9↵
  第一行有兩個正整數 N 與 K, 分別是甲國的城鎮數與分層上限。第二行有 N 個正整數, 依序代表各個城鎮的人口數 P[1]~P[N], 數字間以空白隔開, 人口數總和不超過 109。

- **輸出測試資料一**:

  7↵
  輸出所有設置行政機構的城鎮人口總數。
  (說明)只有城鎮 5 設置了行政機構, 它的人口數是 7。

- **輸入測試資料二**:

  7 3↵
  2 4 1 3 7 6 9↵

- **輸出測試資料二**:

  11↵
  (說明)只有城鎮 5 與 2 設置了行政機構, 人口總數是 7+4=11

ZJ c904. 天龍國的蜜蘋果

- **說明**： 有 N 個蘋果，每顆蘋果有自己的**重量**和**售價**，希望找到 K 個蘋果，使其「單位重量售價 PW」為最大，請問指定 K 時最大的**單位重量售價**為何?

範例一輸入
3 2
5 5
10 7
8 6
2
1

範例一輸出
0.85
1.00

範例二輸入
10 4
4 6
5 6
6 9
9 15
5 11
1 15
4 11
7 10
6 9
1 8
2
5
7
9

範例二輸出
11.50
3.40
2.56
2.20

$$\text{單位重量售價 PW} = \frac{\text{K 個蘋果的總售價}}{\text{K 個蘋果的總重量}}$$

2<=N<=1000　小數點後兩位
1<=M<=50
2<=K<=N

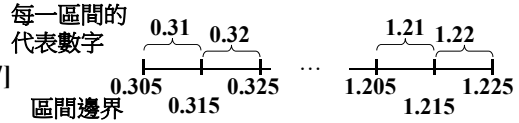
範例一　N＝3, M=2

	K＝2	K＝1
	$\frac{5+7}{5+10}$ = 0.80	$\frac{5}{5}$ = **1.00**
	$\frac{5+6}{5+8}$ = **0.85**	$\frac{7}{10}$ = 0.70
	$\frac{7+6}{10+8}$ = 0.72	$\frac{6}{8}$ = 0.75

最大化平均值　　POJ 3111 K Best

- 「2≤K≤N≤**1000**」說明這個問題不是**暴力列舉**
- 找出 N 個蘋果**個別**的「單位重量售價」的範圍 [rm,rM]
  ⇒ K 個蘋果平均「單位重量售價」的範圍還是 [rm,rM]
- 「小數點以下 2 位數」不是單純指定輸出格式，可能的答案數目變成**有限個**

例如 **[rm,rM]=[0.314,1.217]**



- 對於**範圍**內一個**邊界值 r** 用 n log n 時間可以判斷 $PW \overset{?}{<} r$

$$PW=\frac{\sum_S v_i}{\sum_S w_i} < r \quad \Rightarrow \quad \sum_S x_i = \sum_S (v_i - r\, w_i) < 0$$

S 是 K 個蘋果的子集合

❶ 計算 N 個蘋果個別的 $x_i = v_i - r\, w_i$
❷ 將 $\{x_i\}$ 由大到小排序
❸ 如果最大的 K 個 x_i 的和小於 **0**, 其它任意 K 個的和一定小於 **0**, 因此不存在 K 個元素的集合具有單位重量售價 **r**, **r** 變大時, x_i 變更小,亦即 **PW < r**

注意： **r** 改變時, x_i 的順序可能改變, 某一個 **r** 值最大 K 個的集合, 換一個 r 時不見得是最大的

在區間邊界集合中找最小而且 PW 小於它的 **r**, 答案是 **r**–0.005

結語

- 一下子看了好多用 Binary Search 來解的問題, 題目裡 ❶解答的空間要有上限 (可以到 $2^{p(n)}$)
  ❷需要依照某種順序排放
  ❸能夠有效率地比對順序
- 寫成程式時可以用**迴圈**來寫, 也可以用**遞迴**來寫, 要注意的關鍵點差不多一樣, 包括:
  ❶繼續執行(結束)的條件
  ❷判斷答案位於哪一邊 (哪一半該踢除)
  ❸縮小答案的範圍 (以迴圈或是遞迴繼續)
- 請回頭檢查一下這幾題, 是不是都找得到關鍵點?

更多二分法的應用

- ZeroJudge e809 1.字母排序 (Letters)
- UVa10487.Closest Sums CPE 2015/12/22
- UVa12190.Electric Bill
- Implementation of Robinson-Schensted-Knuth (RSK) algorithm for solving LIS problem in O(n log n)
- ZeroJudge
  f581 圓環出口 APCS 2020/07 #3, f315 低地距離 APCS 2020/10 #4, f608 飛黃騰達 APCS 2021/01 #4, g277 幸運數字 APCS 2021/09 #3, g598 真假子圖 APCS 2021/11 #4, h084 牆上海報 APCS 2022/01 #4, i401 雷射測試 APCS 2022/06 #3, j125 蓋步道 APCS 2022/10 #4