二分搜尋法及其應用

迴圈與陣列
遞迴

丁培毅

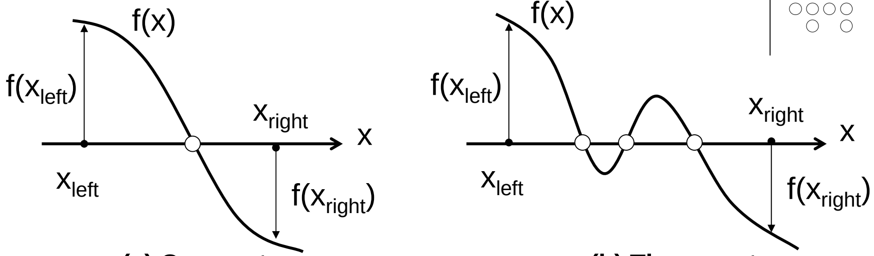
二分搜尋的必要性?!

- 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統統比完為止

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

- 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 target > data[i] 代表 target > data[i-1] > data[i-2] > …
- 不能任性一點嗎? 雖然排序好了, 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

Bisection Root Finding of f(x)



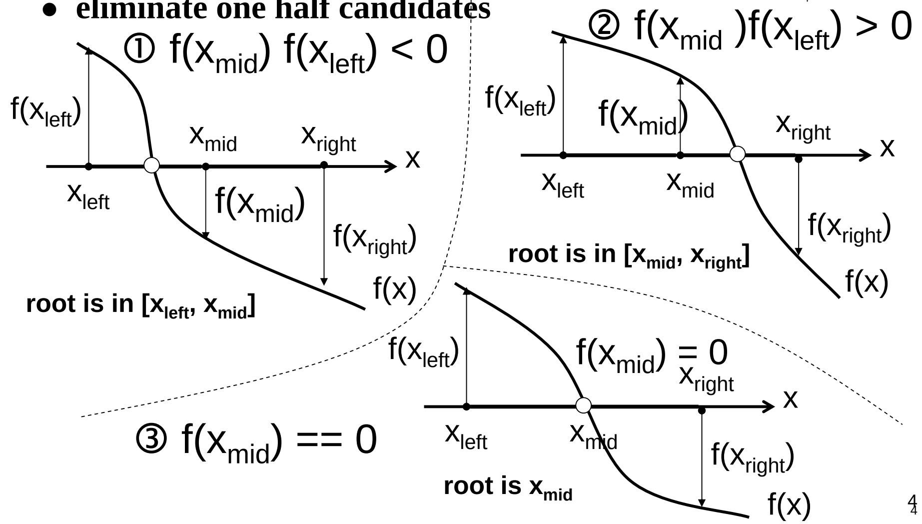
(a) One root (b) Three roots

demanded accuracy ε

- Change of sign implies an odd number of roots in $[x_{left}, x_{right}]$
- Assume there is only one root in $[x_{left}, x_{right}]$
- Brute-force: linear search over $n=(x_{right}-x_{left})/\varepsilon$ segments
- Bisection: $Log_2(n)$ evaluations out of n segments
  - $(x_{right}-x_{left})/2^k \approx \varepsilon$ i.e. $k \approx \log_2(n)$

Three Cases

- divide $[x_{left}, x_{right}]$ as two equal segments
- **eliminate one half candidates**

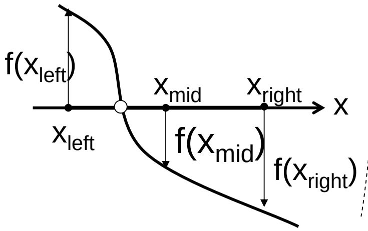


Use a while Loop to divide the interval by 2 at each repetition

1. Given x_{left} and x_{right} , $x_{mid} = (x_{left}+x_{right}) / 2$
2. a. Calculate $f(x_{mid})$
   b. if $f(x_{mid}) < 0$, $x_{right} = x_{mid}$
   c. else if $f(x_{mid}) > 0$, $x_{left} = x_{mid}$
   d. else if $f(x_{mid}) == 0$, root is x_{mid}, break

Repeat the above two steps

Let ε = 1.0e-10




```
while (1)
{
    x_mid = (x_left + x_right) / 2.0;
    if (fabs(f(x_mid)) < 1.0e-10)
        break;
    else if (f(x_left) * f(x_mid) < 0.0)
        x_right = x_mid;
    else // if (f(x_right) * f(x_mid) < 0.0)
        x_left = x_mid;
}
```


Eliminating Redundant Evaluations

- function f() on each point x_{mid} is called 3 times in one iteration, and is called once as x_{left} or x_{right} in the following iteration
  - Use variables to save the function values calculated previously

```
f_left = f(x_left);
f_right = f(x_right);
while (1) {
    x_mid = (x_left + x_right) / 2.0;
    f_mid = f(x_mid);
    if (fabs(f_mid) < 1.0e-10)
        break;
    else if (f_left * f_mid < 0.0) {
        x_right = x_mid;
        f_right = f_mid;
    }
    else if (f_right * f_mid < 0.0) {
        x_left = x_mid;
        f_left = f_mid;
    }
}
```


遞迴函式的設計方法

- 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)
  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爲較小的問題
  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. 拆解爲小一點的問題: 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

arraySum

```
int arraySum(int data[], int n) {
   if (n==0)
      return 0;
   else
      return arraySum(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

Recursive Implementation

❶❷

```
double findRoot(double x_left, double x_right, double eps) {
    double x_mid = (x_left + x_right) / 2.0;
    double f_mid = f(x_mid);
    double f_left = f(x_left);
    if (fabs(x_mid) < eps))
        return x_mid;
    else if (f_left * f_mid < 0.0)
        return findRoot(x_left, x_mid, eps);
    else // f_mid * f_right < 0.0
        return findRoot(x_mid, x_right, eps);
}
```

❺ ❸❹

```
f_left = f(x_left);
f_right = f(x_right);
while (1) {
    x_mid = (x_left + x_right) / 2.0;
    f_mid = f(x_mid);
    if (fabs(f_mid) < 1.0e-10)
        break;
    else if (f_left * f_mid < 0.0) {
        x_right = x_mid;
        f_right = f_mid;
    }
    else if (f_right * f_mid < 0.0) {
        x_left = x_mid;
        f_left = f_mid;
    }
}
```


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

data[]											
	2	3	4	5	6	8	13	16	18	23	24

- **Recursive**

❶❷

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

❺ ❸❹

Find Duplicated Number

- 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(n²) 次比對
  - 使用額外的 n 個元素陣列, 程式執行 O(n) 次比對

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

```
int findDuplicate(int n, int data[]) {
   int i, count[500] = {0};
   for (i=0; i<n; i++)
      if (count[data[i]] == 1)
         return data[i];
      else
         count[data[i]] = 1;
   return -1;
}
```

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

```
int findDuplicate(int n, int data[]) {
   int left=1, right=n-1, mid, i, count;
   while (left < right) {
      mid = (left+right) / 2;
      for (i=0, count=0; i<n; i++)
         if (data[i] <= mid) count++;
      if (count > mid) right = mid;
      else left = mid + 1;
   }
   return left; //  假設一定有一個數字重複
}
```

找出一個方式, 一次刪除一半的可能解

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

left mid right

	1	2	3	4	5	6	7	8	9	10	
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

❶❷

```
int findDuplicate(int n, int data[], int left, int right) {
   int mid=(left+right)/2, i, count;
   if (left == right) return left;
   for (i=0, count=0; i<n; i++)
      if (data[i] <= mid) count++;
   if (count > mid) return findDuplicate(n, data, left, mid);
   else return findDuplicate(n, data, mid+1, right);
}
```

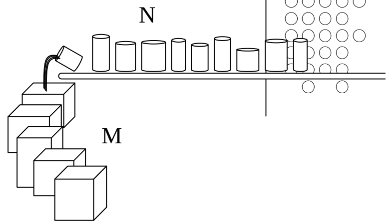
❺ ❸❹

Find Minimum of Sorted and Rotated Sequence

- 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 (假設陣列中資料沒有重複, 有重複的話會比較複雜一點點)
- $O(\log_2 n)$ 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;
  - 這種陣列一定滿足 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

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非常的慢
- 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, 對於某一個 V 值來說, 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, 如果可以, 就表示我們要找的容積的值大於 V

- 我們可以一個一個數值來測試, 但是因為這是連續的區間, 所以可以用**二分法**, 寫迴圈或是**遞迴**來搜尋, 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 值是否太小, 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 V

```
int isVFeasible(int N, int vessels[],  int M, int V) {
  int i, j, amount;
  for (i=0,j=N-1; i<M && j>=0; i++) { // i 是容器, j 是剩餘生乳瓶數
    amount=0;
    while ((j>=0)&&(amount+vessels[j]<=V))
      amount += vessels[j--];
  }
  return j<0; // 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>=0) 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<0) 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也需要列印指定的那一種
- 因此基本上也是運用二分法去找到分配到的頁數的最大值應該要大於多少

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」

data1[median1]

- 例如: n1=12, data1[] 1 3 5 6 8 **10** 11 13 16 19 20 25 , 中數爲 10

  n2=6, data2[] 2 4 **6** 7 9 17 , 中數爲 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, 中數爲 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 個元素爲止
- 爲什麼說是暴力解, 因爲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, 一個一個合併就很花計算機時間了

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] 爲 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爲中數, 比中數大或是比中數小

結語

- 一下子看了好多用 Binary Search 來解的問題, 題目裡 ❶解答的空間要有限制, (可以到 $2^{p(n)}$),
  ❷需要按照某種順序排放,
  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?