

## 1 字串匹配

字串是人類生活中最為常見的一種訊息，舉凡數字 (“17”)、文字 (“a”) 或是同時包含兩者 (“1bd2”)，都可以稱為字串，簡單來說只要是一串可以寫出來的字就算是一個字串。

字串匹配是應用在兩個或多個字串之間的操作。簡言之，當我們有一個字串  $A$  時，我們時常會好奇一個字串  $B$  是否會出現在這個字串  $A$  中的某個地方。舉例來說，當你去查錄取名單時，便是在尋找你的名字是否出現在錄取名單之中，此時，錄取名單便是字串  $A$ ，而你的名字則是字串  $B$ 。事實上，字串匹配的實作方式非常簡單，步驟如下：

1. 先記錄字串  $A$  的長度為  $L_A$ ，字串  $B$  的長度為  $L_B$ 。此時，可以得知所有  $B$  可能出現在  $A$  的位置只有  $0 \sim L_A - L_B$ 。(0-base)
2. 枚舉這些可能的位置一一檢驗。假設當前可能的位置為  $i$ ，則一一比對  $A[i + j]$  是否與  $B[j]$  相同 ( $j \in [0, L_B)$ )。如果不同，便是比對失敗；如果對於所有  $j \in [0, L_B)$  都相同，則表示  $B$  出現在  $A$  中  $i$  的位置。

而以程式碼寫成即為：

```

1 void string_matching( string A , string B ){
2     int lenA = A.length();
3     int lenB = B.length();
4     for( int i = 0; i <= lenA - lenB; i ++ ){
5         bool fail = false;
6         for ( int j = 0; j < lenB; j ++ ){
7             if( A[i + j] != B[j] ){
8                 fail = true;
9                 break;
10            }
11        }
12        if ( !fail )
13            cout << "B matches A at " << i << endl;
14    }
15 }
```

由以上程式碼，以及先前我們所提過複雜度的估計可以發現，對於兩個字串長度分別為  $N, M (N \geq M)$  的字串做匹配，時間複雜度為  $O((N-M) \times M) = O(NM)$ 。因此，如果對於兩個字串長度超過  $10^5$  的字串做匹配時，依複雜度估計來看所花費的時間會非常長，這麼說來，這種字串匹配不就是很沒有效率的演算法嗎？

其實並不必然。就一般生活中所會遇到的字串而言，可以發現字串中的字元有非常多種，假設我們現在只考慮小寫英文字母，則當我們在做字串匹配比較兩個字元時，會匹配成功的機率是  $\frac{1}{26}$ 。就算我們現在假設世界上只有兩種字元，此時對於每對字元匹配成功的機率是  $\frac{1}{2}$ ，在比對字串時，我們需要去比對第二個字元的機率只有  $\frac{1}{2}$ ，需要去比對第三個字元的機率只有  $\frac{1}{4}$ ， $\dots$ ，需要去比對第  $k$  個字元的機率只有  $\frac{1}{2^{k-1}}$ 。所以，我們對於每個可能的位置，期望來說需要比對的字元數量為

$$1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^{M-1}} < 2$$

因此，我們可以把複雜度修正為  $O((N-M) \times 2) = O(N)$ ！(可以試著推導，當字元種類數為  $C$  時，每個位置期望的比對字元數小於  $\frac{C}{C-1}$ 。)

以上的線性時間複雜度是基於字串是隨機的假設，而推導出的期望值，所以我們仍然可以構造出讓演算法跑到  $O(NM)$  時間的測資。我們將在作業中探討如何構造這種測資。

## 2 排序

排序是人類生活中最常遇到的問題之一，對大量數據進行排序更是資料分析的關鍵步驟。底下介紹一些常見的排序算法：

### 1. 氣泡排序法 (Bubble sort)

時間複雜度： $O(n^2)$

額外空間複雜度： $O(1)$

氣泡排序的原理是，每回合當前最大的元素都會透過不斷地與其右手邊的元素交換「浮到」它最終的所在位置，從而在進行  $n - 1$  回合後確定所有元素都已經到達正確的位置。

```
1 void bubble_sort( int array[], int n ){
2     for( int i=1; i<n; ++i )    // run n-1 times is enough
3         for( int j=0; j<n-1; ++j )
4             if( array[j] > array[j+1] )
5                 swap(array[j], array[j+1]);
6 }
```

### 2. 選擇排序法 (Selection sort)

時間複雜度： $O(n^2)$

額外空間複雜度： $O(1)$

選擇排序法的原理是把要排序的序列分成兩堆，一堆是由原序列最小的前  $k$  個元素所組成並且已經照大小排列，另一堆則是原序列中剩餘的  $n - k$  個尚未排序的元素。算法每回合都會從未排序堆中選出最小的元素，然後將其移動到已排序堆中的最後面，類似根據大小一個一個叫號排隊， $n - 1$  回合後便把原序列給排列好了。

```
1 void selection_sort( int array[], int n ){
2     for( int i=0; i<n-1; ++i ){
3         int minIdx = i;
4         for( int j=i+1; j<n; ++j )
5             if( array[j] < array[minIdx] )
6                 minIdx = j;
7         // array[minIdx] is the smallest element in array[i ~ n-1]
8         swap(array[i], array[minIdx]);
9     }
10 }
```

## 3. 插入排序法 (Insertion sort)

時間複雜度： $O(n^2)$ 額外空間複雜度： $O(1)$ 

在講解複雜度的投影片當中我們就有提過插入排序法以及其原理了，該算法一樣把原序列區分成已排序和未排序的兩堆，接著把未排序的元素一個一個插入到已排序堆中正確的位置。

```

1 void insertion_sort( int array[], int n ){
2     for( int i=1; i<n; ++i ){
3         int j, tmp = array[i];
4         for( j=i-1; j>=0 && array[j]>tmp; --j )
5             array[j+1] = array[j];
6         array[j+1] = tmp;
7     }
8 }

```

## 4. 合併排序法 (Merge sort)

時間複雜度： $O(n \log n)$ 額外空間複雜度： $O(n)$ 

合併排序法的原理是把要排序的序列分成前後兩等份，分別遞迴處理成兩個排序好的序列後，再將這兩個序列合併成一個排序好的序列。

```

1 void merge_sort( int array[], int n ){
2     if( n < 2 ) return;
3     // divide into two arrays
4     int len1 = n / 2;           // size of the first array
5     int len2 = n - len1;       // size of the second array
6     int *array1 = array;       // first array
7     int *array2 = array + len1; // second array
8     merge_sort(array1, len1);  // recursion on first array
9     merge_sort(array2, len2);  // recursion on second array
10    // merge
11    int *tmp = new int[n];      // temporary array
12    int len = 0;                // length of the temp array
13    int pos1 = 0, pos2 = 0;     // position of the two elements to compare
14    while( len < n ){
15        if( pos2 == len2 || ( pos1 < len1 && array1[pos1] <= array2[pos2] ) )
16            tmp[len++] = array1[pos1++];
17        else
18            tmp[len++] = array2[pos2++];
19    }
20    // assert( len == n );
21    for( int i=0; i<n; ++i )
22        array[i] = tmp[i];
23    delete[] tmp;
24 }

```

## 5. 快速排序法 (Quick sort)

時間複雜度：期望  $O(n \log n)$ ，最差  $O(n^2)$ 額外空間複雜度：期望  $O(\log n)$ ，最差  $O(n)$ 

快速排序法的原理是選擇序列中一個元素做為基準 (pivot)，接著將小於基準的元素放到序列左邊，大於基準的元素放到序列右邊（與基準相同的數可以放到

任意一邊)，接著遞迴處理左右兩個新的序列。快速排序法平均時間複雜度為  $O(n \log n)$ ，但在基準選得不好，導致左右兩序列大小差很多的情況下，可能達到  $O(n^2)$  的複雜度。一般為了避免這種情況，基準的選擇會是隨機的。

以下的實作中，不將與基準相同的數放到任意一邊，而是放在中間，而遞迴時只遞迴嚴格小於基準與嚴格大於基準的兩序列。

```

1 void quick_sort( int array[], int n ){
2     if( n < 2 ) return;
3     int pivotIdx = rand() % n;          // randomly pick a pivot index
4     int pivotVal = array[pivotIdx];     // pivot value
5     swap(array[pivotIdx], array[n-1]); // move the pivot to end of array
6     // length of the array of smaller elements
7     int len = 0;
8     for( int i=0; i<n-1; ++i ){
9         if( array[i] < pivotVal )
10            swap(array[i], array[len++]);
11     }
12     swap(array[len], array[n-1]);       // move the pivot to middle of the two arrays
13     // length of the array of smaller or equal elements
14     int len2 = len + 1;
15     for( int i=len+1; i<n-1; ++i ){
16         if( array[i] == pivotVal ){
17             swap(array[i], array[len2++]);
18         }
19     }
20     // array[0 ~ len-1] : array with elements smaller than pivotVal
21     // array[len ~ len2-1] : array with elements equal to pivotVal
22     // array[len2 ~ n-1] : array with elements larger than pivotVal
23     quick_sort(array, len);
24     quick_sort(array+len2, n-len2);
25 }

```

#### 6. 基數排序法 (Radix sort) (以 10 進位舉例)

時間複雜度： $O(n \log C)$ （其中  $C$  為數值範圍中的最大值）

額外空間複雜度： $O(n)$

基數排序法只適用於整數，由個位數開始，對第  $r$  位數都做一輪排序。每一輪排序只看第  $r$  位數的大小，以 10 進位整數來說，第  $r$  位數只有 10 種可能，於是就使用 10 個桶子，並將序列中的數字依照第  $r$  位數丟進相對應的桶子，再由 0 號桶子開始，將每個桶子中的數字依照放入的順序拿出，形成一個新的序列。由低位數一直做到最高位數，做完後就完成排序了。

若欲排序數字最大為  $C$ ，則最多會進行  $\log_{10} C$  輪排序，每輪排序需檢查 10 個桶子，其中總共有  $n$  個元素，可以視為  $O(10 + n) = O(n)$ ，因此總時間複雜度為  $O(n \log_{10} C)$ 。

在一般情況，在數值範圍有限（ $C$  有限）之下，我們會將  $\log_{10} C$  視為常數，如排序 int 時該數為 10，因此 Radix sort 可以視為一個線性的排序方法。

```

1 void radix_sort( int array[], int n ){
2     vector<int> bucket[10];
3     for( int radix=1, r=0; r<10; radix*=10, ++r ){ // radix = 1, 10, ..., 10^9
4         for( int i=0; i<10; ++i )                // clear the buckets
5             bucket[i].clear();
6         for( int i=0; i<n; ++i ){                  // push elements into buckets
7             int digit = (array[i]/radix) % 10;     // r-th digit of array[i]
8             bucket[digit].push_back(array[i]);

```

```
9      }
10     int len = 0;
11     for( int i=0; i<10; ++i ){
12         int m = bucket[i].size();
13         for( int j=0; j<m; ++j )
14             array[len++] = bucket[i][j];
15     }
16     // assert( len == n );
17 }
18 }
```

## 習題

1. 了解基本的字串匹配的演算法後，請回答下列問題：
  - (a) (10 pts) 請列出 “mississippi” 與 “sip” 依上述字串匹配方式的匹配過程，及求出所需要匹配的字元對數。(需要找到所有匹配的地方，而不是找到一個之後就停止)
  - (b) (20 pts) 請敘述一種構造方式，構造出兩個長度不超過  $10^6$  的字串  $A, B$ ，使得字串  $B$  不在字串  $A$  中，且依上述字串匹配方式，所需要匹配的字元對數  $\geq 10^9$ ，且字串  $A$  包含至少  $10^3$  種字元。
  - (c) (20 pts) 請敘述一種構造方式，構造出兩個長度不超過  $10^6$  的字串  $A, B$ ，使得字串  $B$  不在字串  $A$  中，且依上述字串匹配方式，所需要匹配的字元對數  $\geq 10^9$ ，且字串  $A$  包含至少  $10^3$  種字元，也不存在連續相同的字元。
2. 了解基本的排序演算法後，請回答下列問題：
  - (a) (10 pts) 請列出使用 merge sort 排序序列  $[1, 8, 5, 3, 2, 6, 4, 7]$  的過程。
  - (b) (10 pts) 請列出使用 quick sort 排序序列  $[1, 8, 5, 3, 2, 6, 4, 7]$  的過程。(pivot 可以亂選，分堆的演算法也不一定要和範例程式碼相同)
  - (c) (10 pts) 請列出使用 radix sort 排序序列  $[26, 15, 27, 35, 17, 36, 28, 16]$  的過程。(只需要排序 2 輪即可)
3. Stability 是排序演算法的一個重要性質。我們說一個排序演算法是 stable，表示對於序列中任意兩個值完全一樣的元素，在排序前後不會改變他們的相對位置，也就是不會前後互換。舉例來說，序列  $[2, 1, 2']$  中有兩個 2，原本在後面的 2 多加上了標用以區別，如果經過排序後形成序列  $[1, 2', 2]$ ，則這個排序演算法就不是 stable。
  - (a) (10 pts) 請問 merge sort, quick sort, radix sort 三個排序演算法分別是否 stable？(以範例程式碼為主)
  - (b) (10 pts) 現在你想要排序一個資料型態為 Data 的序列，兩個該型態的物件可以用 “<”(小於) 運算子比較大小，若  $!(a < b) \ \&\& \ !(b < a)$ ，則表示  $a = b$ 。然而，你只能使用一個基於比較的排序函式，而這個函式使用的排序演算法並不是 stable。請想出一個方法使用這個函式，以得到一個 stable 的排序結果。  
Hint: struct
4. 文中的 Radix sort 是以 10 進位做為舉例，因此上文計算複雜度時，我們就將 10 當成了常數，而在某些步驟忽略了 10 造成的複雜度。但是實際上，Radix sort 並不一定要使用 10 進位，可以使用任意進位制，因此如果使用  $b$  進位的狀況下（簡而言之，將範例程式碼中的 10 換成  $b$ ），回答下列問題。
  - (a) (5 pts) 時間複雜度為何？
  - (b) (5 pts) 額外空間複雜度為何？  
Hint:  $b$  與  $\log b$  都不應視為常數，並用  $b, n, C$  表示答案。