

# DSA HW2 hints

陳步青、周紀寧、劉承昌

## 一、參考資料結構

### (0) 資料介紹:

In this homework, we are required to process a 5-dimension sparse array. Since it is sparse, we should not use a real 5-D array to save these data, because the empty entries alone would take more memory than we have. So here we give two example data structures for handling the data.

### (1) Sparse array:

Given a sparse array, how do we save all its entries?

Take the following matrix for example:

row/col	0	1	2	3	4	5	6	7	8
0									
1		9						8	
2					10				
3									
4			3						
5						2			
6									
7				1					
8									

This is a 9x9 array, but only 6 'real' entries exist. Can we do better than using  $81 * \text{sizeof}(\text{entry})$  memory for this array?

One idea is to compress it into a 1-D array. Each entry is (row,column,value). The array becomes the following:

(1,1,9)	(2,4,10)	(4,2,3)	(5,5,2)	(1,7,8)	(7,3,1)
---------	----------	---------	---------	---------	---------

You can see that we used almost optimal space to save the entries. But the search speed is  $O(\text{number of entries})$ , which is very slow.

Another idea is to compress it into a 1-D array of shorter arrays

row		
0	NULL	
1	(1,9)	(7,8)
2	(4,10)	
3	NULL	
4	(2,3)	
5	(5,2)	
6	NULL	
7	(3,1)	
8	NULL	

Now we take some more space, but searching via the row index is  $O(1)$ , and the remaining search can speed up via binary search.

1-D壓縮的資料結構，就是文字檔的儲存方式。它將 5欄的資料變成entry的一部分，總共 1.4億筆資料存起來。

#### Data Structure 1:

Load all the data into a 1-d array, then sort all the data by user -> adid -> advertiser ...

If you do this with proper optimization, you can finish all the test data in 40 mins.

The query time for DS1 can be done as follows:

**get(u,a,q,p,d):  $O(\log n)$**

**clicked(u):  $O(\log n) + O(\# \text{ of impressed ads of } u)$**

**impressed(u1,u2):  $O(\log n) + O(\# \text{ of impressed ads of } u1 \text{ \& } u2)$**

**profit(a,theta):  $O(n)$ , or  $O(\log n)$**

Implementation: Hidden, please ask TA.

如果覺得1-D不夠快的話可以做2-D壓縮。這邊**不負責任**提供資料資訊:

**user id**總共有**22023547**筆，但是最大的**user id**是**23907634**。因此可以直接開一個23907634大小的陣列，直接存每個user的資料。雖然有~200萬筆空的user，但是相當的省時間，查user只需要 $O(1)$ ，剩下的資料量也從**1.4億**降到最大**90萬**。

user	vector<data>
0	data with u = 0...
1	data with u = 1...
2	data with u = 2...
.	.
.	.
.	.
23907634	data with u = 23907634...

### Data Structure 2:

Load all the data into a 2-d array, use `user_id` as first array index, then sort all the data by `adid` -> advertiser ...

If you do this with proper optimization, you can finish all the test data in 30 mins.

The query time for DS1 can be done as follows:

**get(u,a,q,p,d):  $O(\log(\text{entries of } u))$**

**clicked(u):  $O(\log(\text{entries of } u)) + O(\# \text{ of clicked ads of } u)$**

**impressed(u1,u2):  $O(\# \text{ of impressed ads of } u1 \text{ \& } u2)$**

**profit(a,theta):  $O(\# \text{ users the impressed on } a)$**

Implementation: [Hidden](#), please ask TA.

(\*)  $O(n)$  comparison algorithm:

有兩串資料，要怎麼找出共同的資料呢？

方法一：

for entry1 in array1:

    for entry2 in array2:

        if(entry1 == entry2):

            output entry1

這個方法要 $O(n^2)$ 時間，但是**排序過**，可以更快!!

方法二：

sort(array1)

sort(array2) //  $O(n \log n)$  time

i = 0; j = 0;

```


while (i < array1.size && j < array2.size){
    if(array1[i] < array2[j])
        i++;
    else if(array1[i] > array2[j])
        j++;
    else if(array1[i] == array2[j])
        output array1[i]
        i++;j++;
}


```


因為排序過，保證輸出不重複，比對次數也變少。不算排序只要 $O(n)$ 時間。  
 在impressed中，可以先用這個演算法找出共同的adid，再用相同的演算法，輸出兩者出現的properties.


## (2) std::Map

在C++ stl裡面，有一個神奇的東西叫做map. 老師到很晚才說可以用，不然我們不會在這裡寫。

 **陳步青** 2015-4-06 21:32  
 我還沒寫出來 真的不行我想辦法請教承昌紀寧研究到懂

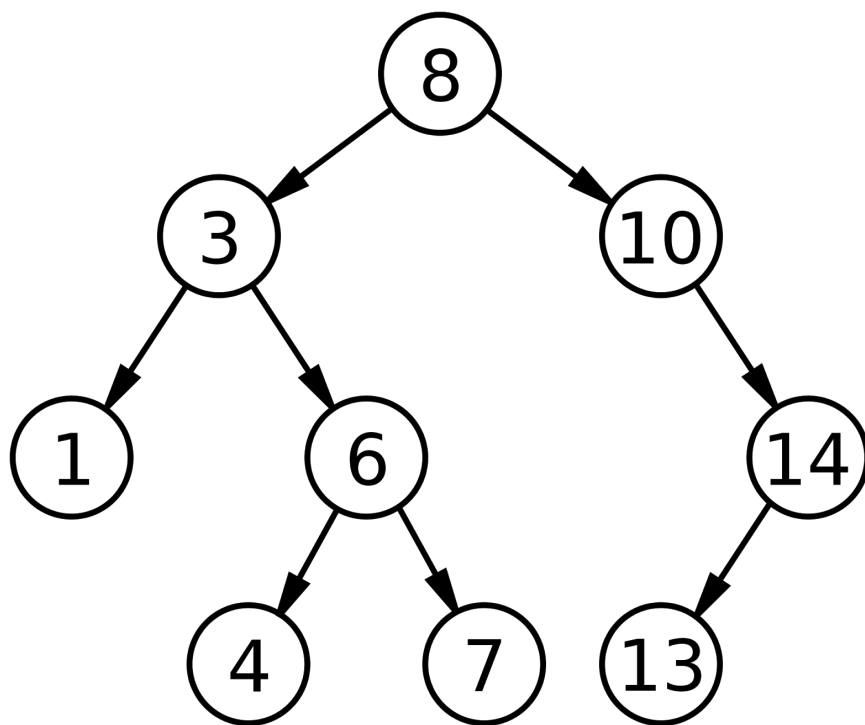
 **林軒田** 2015-4-06 21:32  
 了解，大家辛苦了！  
 我的希望(不管對同學或助教)是我們不用搞得太複雜的結構/  
 演算法，而是用一些基本的工具在合理的時間內做出來

 **陳步青** 2015-4-06 21:36  
 所以應該是希望用老師上課提到的方法就能完成嗎  
 對於合理的界定沒有很清楚 還是我們寫完後丟上來即時討論

 **林軒田** 2015-4-06 21:40  
 我覺得用到簡單的 stl 工具(如 map) OK，但是不需要用到  
 (自己)很複雜的資料結構設計

 **陳步青** 2015-4-06 21:41  
 了解！

老師說:可以跟同學說用法跟array一樣。  
 但我覺得還是要說一下，map其實是一個binary search tree.



picture from wiki

所以能夠支援跳來跳去的index.

用這個的人，希望你們自己去查他的效能，寫在report裡面。

大O分析，會跟array差不多。但因為是stl，所以比較肥、慢。不建議用五層。

可以

user 用array, add當map ...

map<int,ad>	map<int,other_data>
0	
1	
2	
.	
.	
.	
23907634	

Data Structure3,4,5...:

vector<map<vector<data>>>

map<<map<data>>>

map<map<map<data>>> ...

## 二、報告撰寫

跟手寫一起繳交，**兩頁以內，清楚即可**，建議包含以下內容：

Emphasize on why you think the data structure would be (time-wise) efficient for the four desired actions

### 1. 分步驟解釋實作的過程 (一定要寫)

- a. 如何處理讀檔？用什麼資料結構存？為什麼？
- b. 四個函式怎麼實作的，各用什麼演算法在你的資料結構中做搜尋？有先Sort過嗎？為什麼這樣做？這些做法的時間複雜度各是？
  - i. Get
  - ii. Clicked
  - iii. Impressed
  - iv. Profit

### 2. 時間 (optional)

- a. 讀檔案時間
- b. 四個指令分別花的平均時間

### 3. 你是如何有效率的寫完這次作業？跟同學討論還是找TA協助還是自幹？有沒有團隊合作或是組讀書會？希望TA提供哪些東西，對你們完成作業會更有幫助？(optional)

## 三、優化 (痛苦指數: 1~5)

**Compiler Optimization:** (痛苦指數: 0, 加速: 大大大)

- Try the -O2 or -O3 option, either on g++ or gcc.

**Misc:**

- 用C寫，速度會比C++快一點點 (痛苦指數: 3, 加速: 少少)

Use C to implement, it is a little bit faster than C++

下面很多優化都是假設你用C寫，因為C比較接近系統

- 狀態壓縮 (痛苦指數: 0, 加速: 少)

depth, position  $\leq 3$ , 多用一個index很浪費，不如用 $3 * \text{depth} + \text{position}$ 。5層map就可以少一層，省很多記憶體。 \*\*map很肥

**Algorithms:**

- online sorting

用前面DS2的寫法時，每個user所存的資料不用先排好，因為排好也不一定會查到。所以可以等備查到的時候再排序。

- no sorting

如果為四個函數分別寫四個資料庫，那get、clicked這兩個函數可以不用做排序。因為分析助教只有1200筆測資，重複出現的應該不多。兩個函數如果做排序，第一次要 $O(n \log n)$ 時間，之後都是 $O(\log n)$ ，不排序的話只要一次 $O(n)$ 時間，會比較快。

- pointer sorting

為四個函數分別寫四個資料庫，可以都用指標陣列，排序快速，也不會多用太多記憶體。

## Light weight STL:

- **vector** (痛苦指數: 2, 加速: ?)

vector實際上是一個growing array，支援均攤 $O(1)$ 寫入，在ADA會介紹。實作概念是**滿了就長兩倍**。以下psudocode

```
insert(my_vector, entry){
    if(my_vector.size == my_vector.max_size){
        my_vector.max_size *= 2
        my_vector.array = realloc(my_vector.array, sizeof(entry)\
                                   *my_vector.max_size);
    }
    my_vector.array[my_vector.size] = entry;
    my_vector.size++;
}
```

//這邊省略掉初始化malloc，以及realloc的時候要先檢查有沒有realloc成功等步驟。

- **map** (痛苦指數: 2, 加速: ?)

map 相當肥，也不好寫。好在linux/rb\_tree.h有一個**效能非常好的紅黑樹**。他是kernel在系統排成的時候用的資料結構，所以寫得很有效率。可以直接用。

另外，MIT也有實作紅黑樹：

[http://web.mit.edu/~emin/www.old/source\\_code/red\\_black\\_tree/index.html](http://web.mit.edu/~emin/www.old/source_code/red_black_tree/index.html)

都可以研究一下拿來用。

## Loading data:

Here we list functions faster than fstream. fstream

- scanf(): 3分30秒 (痛苦指數: 1, 加速: 多)
- fgets(): 未測試 (痛苦指數: 2, 加速: 多)
- fread(): 未測試
- open()、read():未測試
- mmap (痛苦指數: 5, 加速: 快1.7倍)

mmap是一個把檔案整個讀進記憶體的函式，相當快速。**整的load完+parse完只要1分59秒**。以下為林祥瑞的範例code。

```

172     fprintf(stderr, "cannot read file\n");
173     return errno;
174 }
175
176 int fd_data = open(argv[1], O_RDONLY);
177 if (fd_data < 0)
178 {
179     fprintf(stderr, "cannot open file\n");
180     return errno;
181 }
182
183 struct stat stat_buf;
184 if (fstat(fd_data, &stat_buf))
185 {
186     fprintf(stderr, "cannot stat file\n");
187     return errno;
188 }
189
190 void *data_begin = mmap(NULL, stat_buf.st_size, PROT_READ | MAP_POPULATE | MAP_HUGETLB, MAP_SHARED, fd_data, 0);
191 if (data_begin == MAP_FAILED)
192 {
193     fprintf(stderr, "cannot mmap file\n");
194     return errno;
195 }
196 void *data_end = data_begin + stat_buf.st_size;
197
198 if (close(fd_data))
199 {
200     fprintf(stderr, "cannot close file\n");
201     return errno;

```

## Avoiding System Call Malloc:

獨立出來寫是因為加速超多。已以往經驗，DSA的final project都可以**節省一半以上的時間。但寫起來也相當痛苦。**

解釋一下，要求動態配置的記憶體時，會經過很多層的system call，等作業系統處理你的要求。相當花時間。

替代的方法是用國際空間，也就是 data\_segment 的記憶體。

(系統程式設計會講，function call 在 stack 上面，malloc 在 heap 上面)

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <assert.h>
5  #include "raw_data.h"
6  #include "get_clicked.h"
7  #include "impressed.h"
8  #include "profit.h"
9
10 char memory_pool[2000000000];
11 char *cur_pos;
12
13 int main(int argc, char *argv[]){
14     //初始化沒用到的位置
15     cur_pos = memory_pool;
16
17
18     struct raw_data_db raw_data;
19     if(argc<2){
20         printf("Usage: ./demo [data_file]\n");
21         exit(EXIT_FAILURE);
22     }

```



```

char* my_malloc(int size){
    char *ret;
    if(cur_pos+size-memory_pool <200000000){
        ret = cur_pos;
        cur_pos = cur_pos+size;
    }
    return ret;
}

```

\*如果寫平行運算，記得要mutex\_lock  
allocate到的記憶體，也要用比較特別的方式存：

```

int total_chunks;
int *chunks[100];
int chunk_size[100];
chunks[ 0 ] = my_malloc(size);
chunk_size[0] = size;
total_chunks = 1;

```

## Parallel Programming:

- openmp

```

4
5 #include<omp.h>
6
7 #pragma parallel for
8 for (int i = 0;i<10;i++){
9     printf("%d\n",i);
10 }
~

```

openmp是很容易上手的平行畫，可以直接平行單一迴圈。

好人教學：

<https://kheresy.wordpress.com/2006/06/09/%E7%B0%A1%E6%98%93%E7%9A%84%E7%A8%8B%E5%BC%8F%E5%B9%B3%E8%A1%8C%E5%8C%96%E6%96%B9%E6%B3%95%E5%BC%8Dopenmp%E5%BC%88%E4%B8%80%E5%BC%89%E7%B0%A1%E4%BB%8B/>

- pthread

pthread稍微難寫一點。助教一開始就想要用平行寫，所以四個函式就直接用三個資料結構存，可以平行建造，所以不多花時間。有興趣的人自己研究。

```
18     struct get_clicked_ds gc_db;
19     struct impressed_db impressed_ds;
20     struct profit_ds profit_db;
21     pthread_t t[3];
22     //make threads
23     pthread_create(t, NULL, make_get_clicked, (void*)raw_data);
24     pthread_create(t+1, NULL, make_impressed_db, (void*)raw_data);
25     pthread_create(t+2, NULL, make_profit_db, (void*)raw_data);
26     // wait for threads to finish and catch return value
27     void *rt_val;
28     pthread_join(t[0], &rt_val);
29     gc_db = *(struct get_clicked_ds*) rt_val;
30     pthread_join(t[1], &rt_val);
31     gc_db = *(struct impressed_ds*) rt_val;
32     pthread_join(t[2], &rt_val);
33     profit_db = *(struct profit_ds*) rt_val;
34
```