Sorting Report 2

資工一B 曹咏萱 409410082

(一)排序演算法

1.quick sort

(1)partitoin:選出一個key，把整個陣列分成小於key和大於key兩個數列，使得key左邊的數字皆比它小，右邊的數字皆比它大。

(2)將小於key和大於key的兩邊分別進行重複的動作直到無法再被分割成兩個數列。

若我們固定將陣列的第一個(也就是最左邊)當作key

從陣列的頭開始向右尋找>key的人，同時從陣列的尾開始向左尋找<=key的人，各自找到後交換，當兩邊碰到(相鄰)時，就結束尋找的動作，並將兩者交換，最後把key放入中間，這就完成一次的partition。

測資:41 11 25 25 26 79 22 79 12 77 76 1 29 42 32

向右尋找>key→　　←向左尋找<=key

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
41	11	25	25	26	79	22	79	12	77	76	1	29	42	32

將a[0]=41當作key，讓i=0+1=1、j=14，i負責記錄向右找的idx，j負責向左找的idx。

向右找尋>41的值，a[1]=11<41略過，一路到a[5]=79>41，此時i=5；

向左找尋<=41的值，a[14]=32<41，此時j=14；將兩個人交換。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
41	11	25	25	26	32	22	79	12	77	76	1	29	42	79

重複這個動作直到交換完a[9]=77和a[11]=1兩人，此時i=9、j=11

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
41	11	25	25	26	32	22	29	12	1	25	77	79	42	79

繼續往11的前一個找去，找到a[10]=25<41，此時j=10

往9的下一個找去，發現這時候i=10，i=j，兩邊碰頭，結束尋找

這時候i=j，不必再進行交換，將key和a[i]=a[10]交換，得到

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

25	11	25	25	26	32	22	29	12	1	41	77	79	42	79

檢查一下，key(41)的左邊都比它小，右邊都比它大，partition成功

接下來將兩邊視作兩個陣列分別進行重複的動作

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**		**10**	**11**	**12**	**13**	**14**
25	11	25	25	26	32	22	29	12	1		41	77	79	42	79

2.merge sort

(1)divid:將整個陣列切成多個子陣列，各自排序。

(2)merge:將排序好的子陣列合併起來。

測資:18 94 29 6 17 74 37 6

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
18	94	29	6	17	74	37	6

將這個擁有8個數字的陣列一路切下去，直到每個數列剩下一個元素。

0	**1**	**2**	**3**		**4**	**5**	**6**	**7**
18	94	29	6		17	74	37	6

0	**1**		**2**	**3**		**4**	**5**		**6**	**7**
18	94		29	6		17	74		37	6

0		**1**		**2**		**3**		**4**		**5**		**6**		**7**
18		94		29		6		17		74		37		6

依照大小順序倆倆合併起來

0	**1**		**2**	**3**		**4**	**5**		**6**	**7**
18	94		6	29		17	74		6	37

比較兩個子陣列的頭，18>6，所以6放下來，接著往後比，18<29，18放下來，以此類推

0	**1**	**2**	**3**		**4**	**5**	**6**	**7**

6	18	29	94		6	17	37	74

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
6	6	17	18	29	37	74	94

3.heap sort

Heap sort 需要用到一種名為「樹」的結構，而此處用到的是其中的二元樹(Binary Tree)。

【Min heap】每個子樹都滿足root的值最小。

【Max heap】每個子樹都滿足root的值最大，當違反此原則時，我們就必須要調整它。

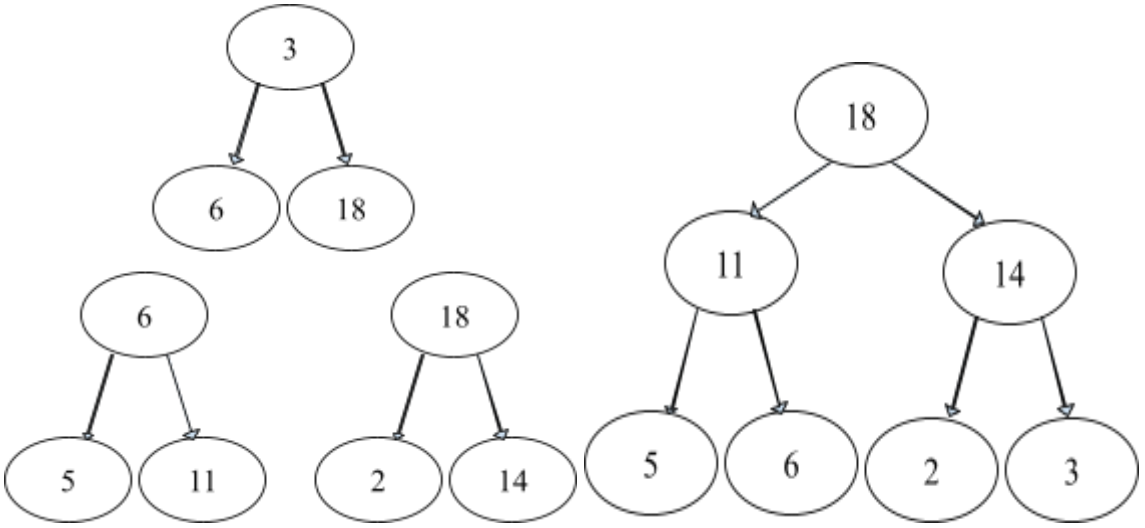
以下使用Max heap。

測資:3 6 18 5 11 2 14

0	**1**	**2**	**3**	**4**	**5**	**6**
3	6	18	5	11	2	14

這個陣列有7個元素，我們可以從第(7/2=)3個元素開始建樹。

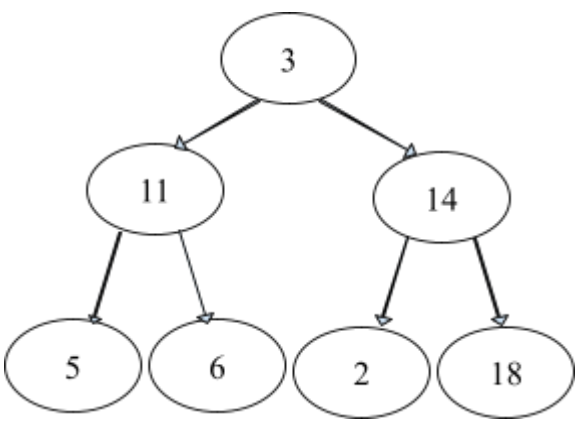
把n當作root和2n+1和2n+2兩個位置建成一棵樹，將自己和兩個child比較，如果發現child>自己就交換過來。像是此時6<11，不符合root最大的原則，交換過來。

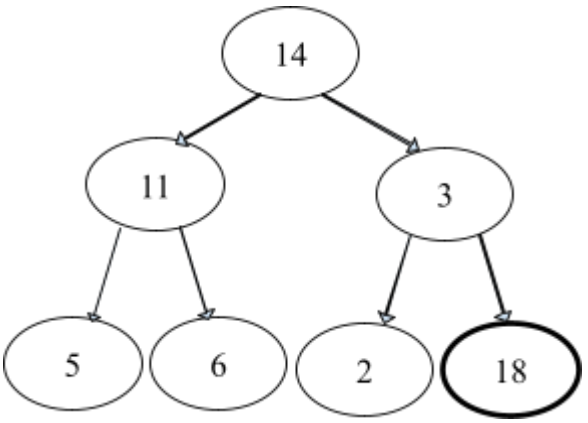


0	**1**	**2**	**3**	**4**	**5**	**6**
18	11	14	5	6	2	3

接下來開始進行排序的動作:

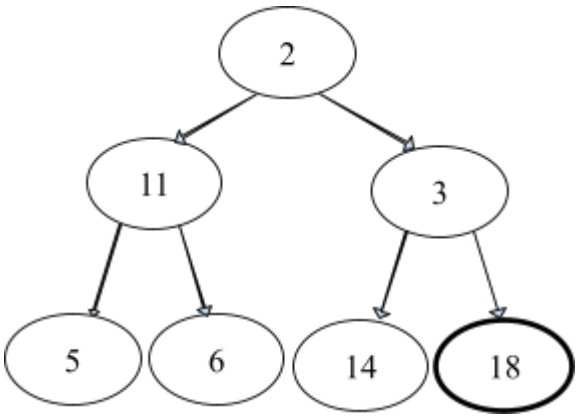
首先將root(18)和陣列的最後一個數字(3)進行交換，交換完之後發現不符合原則了，調整。

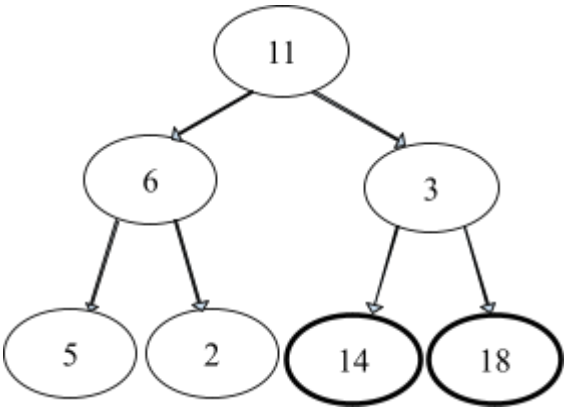




0	1	2	3	4	5	6
14	11	3	5	6	2	18

重複一樣的動作，將root(2)和陣列的倒數第二個數(2)交換，再調整。

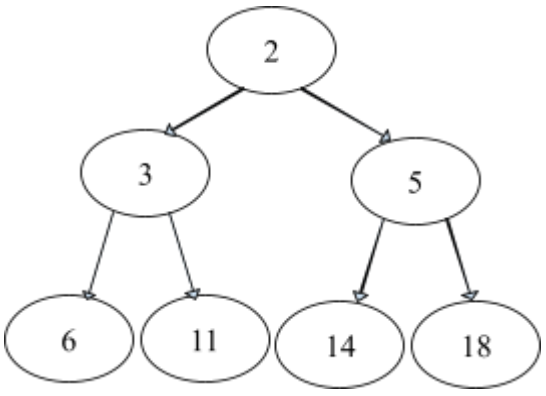




0	1	2	3	4	5	6
11	6	3	5	2	14	18

……

最後應該會變成:



0	1	2	3	4	5	6

2	3	5	6	11	14	18

//每個root(idex=n)的child在陣列裡都是在idex=2n+1和idex=2n+2的位置。

4.radix sort

依據整數的每個位數來排序。

【Most significant digit (MSD)】從最大的位數排到最小的。

【Least significant digit (LSD)】從最小的位數排到最大的。

以下使用LSD。

測資:170 45 75 90 802 2 24 66

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
170	45	75	90	802	2	24	66

從最右邊(最小)的位數開始排序:

0,0,2,2,4,5,5,6

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
170	90	802	02	24	45	75	66

接著比較下一位數, 位數不足的補0:

0,0,2,4,6,7,7,9

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
802	002	024	045	066	170	075	090

對最後一個位數進行排序:

0,0,0,0,0,0,1,8

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
2	24	45	66	75	90	170	802

排序完成。

實作的時候會用到一個tmp[10][cnt] //cnt是陣列大小

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
170	45	75	90	802	2	24	66

將數字分別放到對應的欄中

0	1	2	3	4	5	6	7	8	9
170		802		24	45	66			
90		2			75				
...	...	...	...	...	...	...	...	...	...

然後再依照欄目順序放入陣列，就完成第一次的排序了

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
170	90	802	2	24	45	75	66

(二)Make file

```
all: main_number main_alpha
main_number: main_number.c (所有sort的.o檔)
    gcc main_number.c (所有sort的.o檔) -o main_number
main_alpha: main_alpha.c (所有sort的.o檔)
    gcc main_alpha.c (所有sort的.o檔) -o main_alpha
quicksort.o: quicksort.h quicksort.c
    gcc quicksort.c -o quicksort.o
(......)
clean:
    rm (所有檔案)
```

(三)測試條件

1.測試環境

作業系統 : Ubuntu/工作站

CPU : intel i5

RAM : 8.00GB

2.建立測資

(1)number

使用rand()隨機產生。

(n=1,000,000)

(n=5,000,000)

(2)string

使用rand()隨機產生整數，將其%26+'a，產生一個小寫英文字母，並重複100次，得到一個長度為100的字串。

重複執行1,000,000次，得到data。

3.測量工具

使用 <sys/time.h> 裡的函式gettimeofday()。

```
unsigned long diff;
struct timeval start;
struct timeval end;
gettimeofday(&start,NULL);
//sorting function
gettimeofday(&end,NULL);
diff=1000000*(end.tv_sec - start.tv_sec) + end.tv_usec -
start.tv_usec;
```

(四)實驗結果

1.number

當測資為1,000,000筆整數時。

(單位:sec)	quick sort	merge sort	heap sort	radix sort
1	0.193163	0.201076	0.313722	0.340786
2	0.194263	0.201700	0.313305	0.341824
3	0.191571	0.199095	0.315124	0.333726

當測資為5,000,000筆整數時。

(單位:sec)	quick sort	merge sort	heap sort	radix sort
1	1.119700	1.104908	1.985704	1.177064
2	1.117498	1.112455	2.002342	1.191527
3	1.105594	1.107864	1.990218	1.189169

2.string

當測資為1,000,000筆，長度為100的字串時。

(單位:sec)	quick sort	merge sort	heap sort
1	0.770335	2.154655	0.947233
2	0.771333	2.046131	0.942243
3	0.750734	2.013676	0.952786

radix sort

因為字串的比較和整數不一樣，是從前面開始依字母大小排序，若依照radix sort的方法可能會有問題。

例如:apple banana angry cat 執行完第一輪後會出現

a	**b**	**c**	**d**
apple	banana	cat	
angry			

接著按照radix sort的排序方法應該要比較第二位的字母

a	**...**	**n**	**p**
banana		angry	apple
cat			

這時候會發現兩個a開頭的字串都排到很後面去了，而繼續做下去就會發現得不到想要的結果。

如果要得到正確的排序，應該在做完第一輪之後繼續去比較a欄目裡面字串的大小，但我覺得這樣的作法不太像是radix sort，所以最後就沒有做這方面的實驗。

(五)時間複雜度與穩定性

1.quick sort

時間複雜度(time complexity)

Best Case：O(NlogN)

Worst Case：O(N^2) 當陣列是(接近)已排序，partiton的時候永遠都會切出一個大小為1的陣列

Average Case：O(NlogN)

空間：O(1) 都在原本的陣列進行動作

不穩定(unstable):值相同的資料，排序後順序和排序前不同

0	1	2	3	4
6	2	5	9	5*

經過第一次partition, 陣列會變成

0	1	2		3	4
5*	2	5		6	9

再將兩邊分別quicksort之後

0	1	2	3	4
2	5*	5	6	9

當程式跑完時, 5*在5的前面, 因此不穩定

2.merge sort

時間複雜度(time complexity)

Best Case：O(NlogN)

Worst Case：O(NlogN)

Average Case：O(NlogN)

空間 : O(N) 需要另開一個N大小的空間供merge時使用

穩定(stable):值相同的資料，排序後順序和排序前一樣

0	1	2	3	4
6	2	5	9	5*

0	1		2	3		4
6	2		5	9		5*

0		1		2		3		4
6		2		5		9		5*

0	1		2	3		4
2	6		5	9		5*

0	1		2	3	4
2	6		5	5*	9

0	1	2	3	4
2	5	5*	6	9

當程式跑完時，5在5*的前面，因此穩定

3.heap sort

時間複雜度(time complexity)

Best Case：O(NlogN)

Worst Case：O(NlogN)

Average Case：O(NlogN)

空間 : O(1) 都在原本的陣列進行動作

不穩定(unstable):值相同的資料，排序後順序和排序前不同

0	1	2
8	5	5*

經過一次heap sort會變成

0	1	2
5*	5	8

當程式跑完時，5*在5的前面，因此不穩定

4.radix sort

時間複雜度(time complexity)

Best Case：O(dN) //d為資料中最多有幾個位數

Worst Case：O(dN)

Average Case：O(dN)

空間 : O(N+d)

穩定(stable):值相同的資料，排序後順序和排序前一樣

因為是按照順序和位數排序，不會改變到相同數字原本的排列方式。

(六)優化排序法

1.quick sort

原本的做法中，我將a[0]設為我的key，但如果今天很不幸a[0]是整個陣列的最小值或最大值，那麼partition的時候，就會不斷產生一個大小為1的陣列:

0	**1**	**2**	**3**	**4**	**5**	**6**
3	4	5	6	7	8	9

取a[0]=3當作key，partition完會長這樣

0	**1**	**2**	**3**	**4**	**5**	**6**
3	4	5	6	7	8	9

可以發現pivot左邊的陣列只有一個人，當兩邊分開quick sort時，左邊會直接return，等同於只進行了一邊的sorting，這樣做下去的話效率會很低。

所以我們可以藉由比較a[left],a[right],和a[mid]找出這三者的中間值，並選擇其來當key，以此解決這個問題。//(mid=(left+right)/2)

(單位:sec) (n=1,000,000)	quick sort (原版)	quick sort (改良)
1	0.193163	0.198121
2	0.194263	0.196959
3	0.191571	0.201148

(單位:sec) (n=5,000,000)	quick sort (原版)	quick sort (改良)
1	1.119700	1.080335
2	1.117498	1.095744

3	1.105594	1.083692

從實驗結果可以看到當資料量為1,000,000時，秒數反而少量增加了，我猜是因為資料是由亂數產生，所以原本的quick sort就夠快了，而改良後的版本還增加了比較的時間。不過當資料量放大到5,000,000時，就可以看到優化的成果了。

2.merge sort

當小於一定個數時改用insertion sort(以下只取有變動的code)

```
void mergesort(int a[],int cnt){
    if(cnt<15){ //小於一定數字時改成insertionsort
        insertionsort(a,cnt);
        return;
    }
```

()內的數字是<多少進行insertion sort

(單位:sec) (n=1,000,000)	merge sort (原版)	merge sort (15)	merge sort (25)
1	0.201076	0.183750	0.184017
2	0.201700	0.181309	0.188166
3	0.199095	0.184306	0.184411

(單位:sec) (n=5,000,000)	merge sort (原版)	merge sort (15)	merge sort (25)
1	1.104908	1.035727	1.035278
2	1.112455	1.036308	1.039130
3	1.107864	1.023196	1.042309

從實驗結果可以發現，當取陣列內元素剩下不多(cnt<15)時，改用insertion sort會比全部用merge sort還要快上一點，但如果取太多(cnt<25)可能反而會變慢。

(七)Github

code連結: https://github.com/reki9185/hw1

(八)總結

當排列整數時(1,000,000): quick sort < merge sort < heap sort < radix sort

當排列整數時(5,000,000): merge sort < quick sort < radix sort < heap sort

當排列字串時: quick sort < hepa sort < merge sort

優化之後，當排列整數時(1,000,000): merge sort < quick sort < heap sort < radix sort

優化之後，當排列整數時(5,000,000): merge sort < quick sort < heap sort < radix sort

(九)參考資料

Juice 實作考驗003 a.k.a. QuickSort

https://ccu.juice.codes/courses/aRsdZ/lessons/PGfwG/questions/aZtn8

Comparison Sort: Quick Sort(快速排序法)

https://alrightchiu.github.io/SecondRound/comparison-sort-quick-sortkuai-su-pai-xu-fa.html

Comparison Sort: Merge Sort(合併排序法)

https://alrightchiu.github.io/SecondRound/comparison-sort-merge-sorthe-bing-pai-xu-fa.html

Comparison Sort: Heap Sort(堆積排序法)

https://alrightchiu.github.io/SecondRound/comparison-sort-heap-sortdui-ji-pai-xu-fa.html

基數排序Radix sort - Rust Algorithm Club

https://rust-algo.club/sorting/radix_sort/

基數排序法

https://openhome.cc/Gossip/AlgorithmGossip/RadixSort.htm

簡單學makefile:makefile介紹與範例程式

https://mropengate.blogspot.com/2018/01/makefile.html

C 速查手冊 單元 9 - 標頭檔

http://kaiching.org/pydoing/c/c-header.html

助教的example

https://github.com/aqweteddy/SimpleMakefile