

University of Canterbury

End of Year Examinations 2010

Prescription number: COSC229

Paper title: Algorithms

Time allowed: 3 hours

Number of pages: 7

- Answer *all* questions.
- This test is worth a total of 100 marks.
- This test is **open book**. Calculators are allowed.
- Please check carefully the number of marks allocated to each question. This suggests the degree of detail required in each answer, and the amount of time you should spend on the question.
- Use the separate answer booklet for answering all questions.
- Answer *all* questions.

Question 1. [20 marks for the whole question]

The following is a C program for heapsort.

```

1. #include <stdio.h>
2. int a[100], i, n;
3. void siftup(int p, int q){
4.     int j, k, y, z;
5.     y=a[p]; j=p; k=2*j;
6.     while(k <= q){
7.         z=a[k];
8.         if(k<q)
9.             if(a[k]<a[k+1]) {k++; z=a[k];}
10.        if (y>=z) break;
11.        a[j]=z; j=k; k=2*j; // larger son goes up
12.    }
13.    a[j]=y; // y settles down
14. }
15. void buildheap(){
16.     int i;
17.     for(i=n/2; i>=1; i--)siftup(i, n);
18. }
19. void heapsort(){int w;
20.     for(i=n;i>=2;i--){
21.         w=a[1]; a[1]=a[i]; a[i]=w; //a[1] and a[i] swapped
22.         siftup(1,i-1);
23.     }
24. }
25. main(){
26.     scanf("%d", &n);
27.     getchar();
28.     for(i=1;i<=n;i++)scanf("%d", &a[i]);
29.     buildheap();
30.     heapsort();
31.     for(i=1;i<=n;i++)printf("%d ", a[i]);
32. }
```

The heap used is a so-called max-heap, that is, the maximum at the root. The idea of “siftup” is to copy $a[p]$ to y and traverse down the tree, moving the larger son, z , to the parent position until y becomes larger or we hit the bottom of the tree. Then y settles at the position.

When we come down the tree, we need two comparisons at each level; one for choosing the larger son, and one for comparison between y and the larger son.

(1) If we apply $\text{siftup}(1, 10)$ to the heap in array $a = (12, 10, 8, 6, 9, 5, 7, 3, 2, 1, 4)$ after $a[1]$ and $a[11]$ are swapped, we have the following trace. See Figure 1. Following this example, trace $\text{siftup}(1, 10)$ when applied to $a = (12, 10, 8, 9, 6, 5, 7, 3, 2, 1, 4)$ after $a[1]$ and $a[11]$ are swapped. That is, apply $\text{siftup}(1, 10)$ on $a=(4, 10, 8,$

9, 6, 5, 7, 3, 2, 1, 12). Data movements are shown by arrows. The new value of each node is at the left of each node. [5 marks]

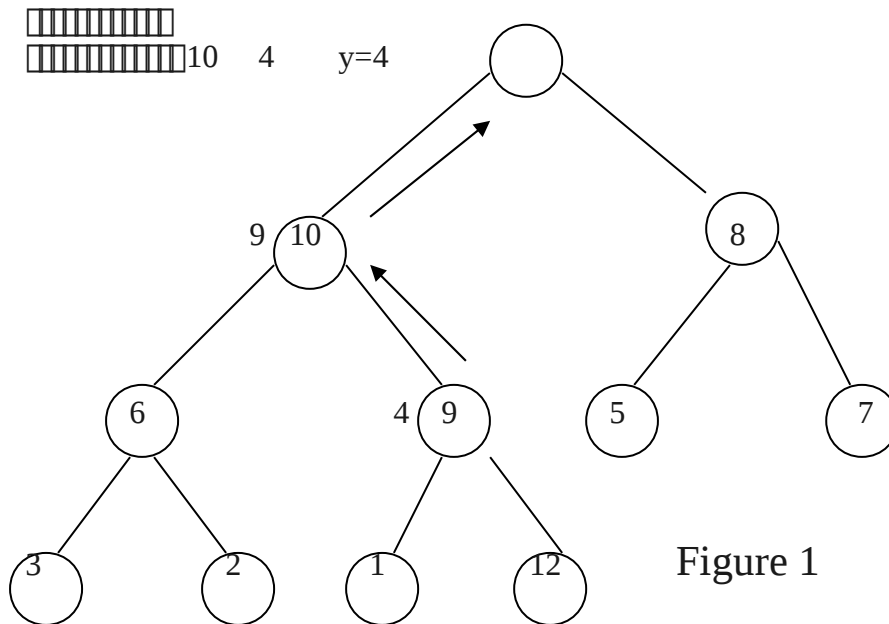


Figure 1

(2) We want to minimize the number of key comparisons. Suppose we keep moving the larger son up without comparing it with y , arriving at the bottom of the tree. That is, we remove line 10. The leaf is not necessarily the resting position for y ; maybe a few levels up. See Figure 2. A partially done C program for the new siftup follows.

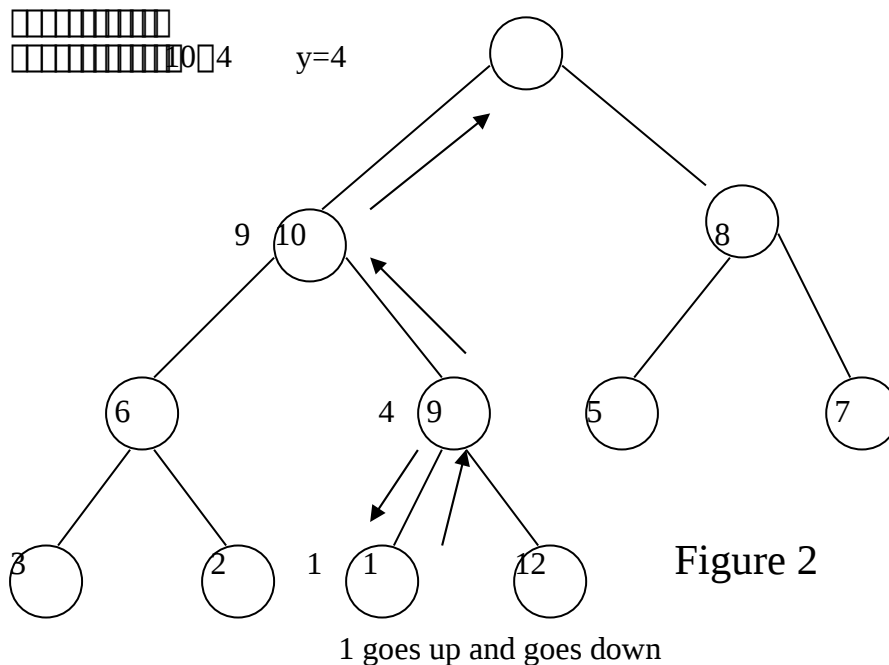


Figure 2

```
void siftup(int p, int q){
    int j, k, y, z;
    y=a[p]; j=p; k=2*j;
    while(k <= q){
```

```

    z=a[k];
    if(k<q){if(a[k]<a[k+1]) {k++; z=a[k];}}
    a[j]=z; j=k; k=2*j; // Data movement ; larger son goes up.
}
a[0]=99; // 99 for infinity
while(y>___){a[j]=a[j/2]; j=j/2;} // a[j/2] goes down
a[j]=y;
}

```

Fill the unfinished part (underlined). Explain why this version is more efficient for sorting the random data. Trace this siftup for siftup(1, 10) on a=(4, 10, 8, 9, 6, 5, 7, 3, 2, 1, 12). [10 marks]

(3) In the above version, data movements take place along the path from the root to the node at the bottom specified by index j at the end of the first while loop. We do not need all of these data movements. Data only need to move above the final resting position. We write such siftup. Fill the blank part. Trace this siftup(1, 10) on the same a as in (2). [5 marks]

```

void siftup1(int p, int q){
    int j, k, x, y;
    y=a[p]; j=p; k=2*j;
    while(k <= q){ // To find the path of larger sons to the bottom
        if(k<q)if(a[k]<a[k+1])k++;
        j=k; k=2*j; // No data movement
    }
    (** Fill this part **)
}

```

Question 2. [15 marks for the whole question]

The single pair shortest path problem is to find the shortest path between a pair (s, t). We call s the source and t the sink. We extend Dijkstra's algorithm for the single source shortest path problem to a single pair problem by a bi-directional approach. The basic idea is that in phase 1 we expand the solution set S from s in the forward direction of edges and the solution set T from t in the backward direction of edges until there is a non-empty intersection of S and T. Then in phase 2, we compute the shortest path that uses only vertices in $S \cup T$. Infinity is given by 99. $c[v,w]$ is the cost of edge (v, w). We assume $c[v, v]=0$ for edge (v, v). The algorithm follows.

```

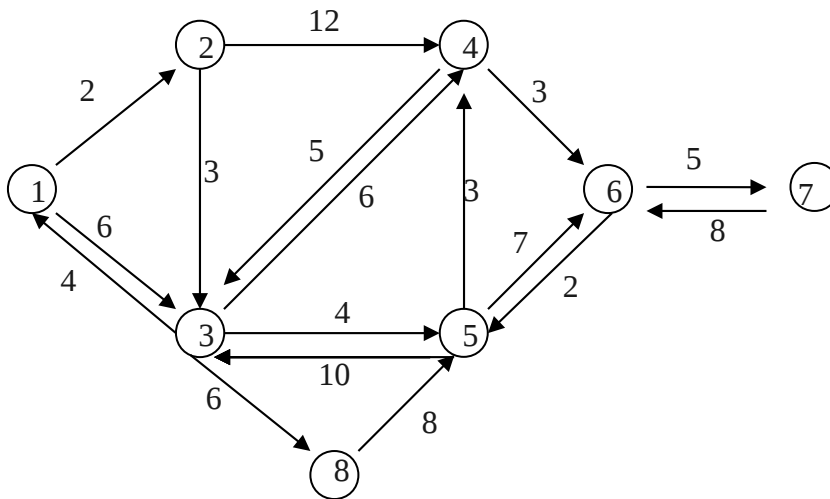
/** Phase 1 **/
1. S={s}; T={t};
2. for v in V do d[v]=c[s, v];
3. for v in V do e[v]=c[v, t];
4. while  $S \cap T = \emptyset$  do begin
5.   Let v be vertex u in V-S such that d[u] is minimum;
6.   Put v in S;
7.   for w in V-S do d[w]=min{d[w], d[v]+c[v, w]};
8.   Let v be vertex u in V-T such that e[u] is minimum;
9.   Put v in T;
10.  for w in V-T do e[w]=min{e[w], e[v]+c[w, v]};
11. end;
/** Phase 2 **/
12. d[s, t]=99;
13. for v in S do

```

14. **for** w in T **do**

15. **if** there is an edge (v, w) **then** $d[s, t] = \min[d[s, t], d[v] + c[v, w] + e[w]]$

(1) The following is a partial trace of the algorithm for the shortest path from $s=1$ to $t=7$ with the following graph. Complete the trace. An iteration means a snapshot at the end of while loop. For phase 2, check all possible edges (v, w) such that v in S and w in T. [5 marks]



[5 marks]

Iteration 1: $v=2$, $S=\{1, 2\}$, $d = (0, 2, 5, 14, 99, 99, 99, 99)$

$v=6$, $T=\{6, 7\}$, $e = (99, 99, 99, 8, 12, 5, 0, 99)$

Iteration 2: $v=3$, $S=\{1, 2, 3\}$, $d=(0, 2, 5, 11, 9, 99, 99, 11)$

$v=4$, $T=\{4, 6, 7\}$, $e=(99, 20, 14, 8, 11, 5, 0, 99)$

(2) Prove that we can check only vertices in $S \cup T$ in phase 2 to finalize the shortest distance. In the example vertex 8 is not checked in phase 2. [5 marks]

(3) A rough analysis of computing time follows. Suppose phase 1 ended with $|S|=|T|=k$. We do experiment of generating v at random in S , and see if it can hit an element in T . The probability that it can hit T is k/n . The expected number of trials for hitting is n/k . Thus from the equation $k=n/k$, we have $k=n^{1/2}$ for the expected value of k . The body of the while loop can be done in $O(n)$ time. Thus phase 1 takes $O(n^{3/2})$ expected time. How much time is needed for phase 2? [5 marks]

Question 4. [15 marks for the whole question]

(1) The following is the Kruskal algorithm for the minimum cost spanning tree. The algorithm is partially executed. Name vector is to tell the name of tree to which a vertex v belongs. $s[v]=k$ means vertex v belongs to the tree named k . Along with the name vector are the edge inspected and the number of name changes. Finish the trace. Confirm the total number of name changes does not exceed $n \log_2(n)$ with $n=9$.

Note. $\log_2(9)=3.17$. [5 marks]

Sort edges in non-decreasing order into list L

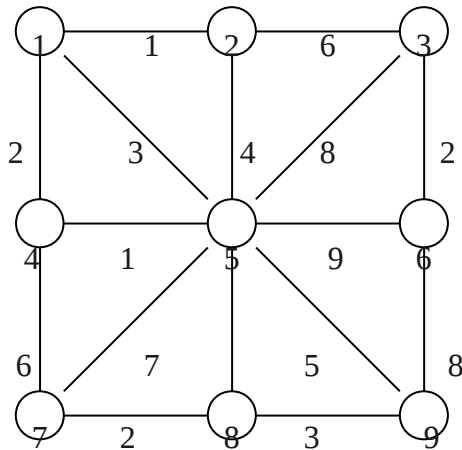
Initialize name vector s to $s=(1, 2, \dots, n)$, that is, $s[i]=i$ for $i=1, \dots, n$

while the number of trees is greater than 1 do begin

 Delete an edge from the first position in L, and let it be (v, w)

 if s[v] is not equal to s[w] then merge smaller tree to the larger tree by changing
 the names of smaller to that of larger

end



Partial trace

Name vector	edge	changes
s=(1,2,3,4,5,6,7,8,9)	(1,2)	1
s=(1,1,3,4,5,6,7,8,9)	(4,5)	1
s=(1,1,3,4,4,6,7,8,9)	(1,4)	2
s=(1,1,3,1,1,6,7,8,9)	(3,6)	1
s=(1,1,3,1,1,3,7,8,9)	(7,8)	1
s=(1,1,3,1,1,3,7,7,9)	(1,5)	skipped
	(8,9)	1
s=(1,1,3,1,1,3,7,7,7)		

Example. Line 3 to 4 in above
Edge (1, 4) taken. Two changes of
name 4 to name 1

(2) Let us measure the number of name changes. Whenever s[v] changes, v will join a tree which is at least twice as big. Thus each vertex goes through at most $\log_2 n$ name changes, meaning the total number of name changes is bounded by $n \log(n)$. We assume the base of logarithm is 2 from this point onwards. In following we have a sharper analysis. We prove that the total number of name changes, C, to complete the algorithm is bounded by $C \leq (1/2)n \log_2 n$.

The proof is by induction. Theorem it true for $n=1$, that is, no name change. We assume T_1 and T_2 , whose sizes are n_1 and n_2 with $n_1+n_2=n$, are merged at the last step. Suppose the theorem is true for trees of size up to n . Then by induction we have

$$C \leq \min\{n_1, n_2\} + (1/2)(n_1 \log n_1 + n_2 \log n_2)$$

(*** Fill this part ***)

$$\leq (1/2)(n_1+n_2) \log(n_1+n_2) = (1/2)n \log(n)$$

END OF PAPER