

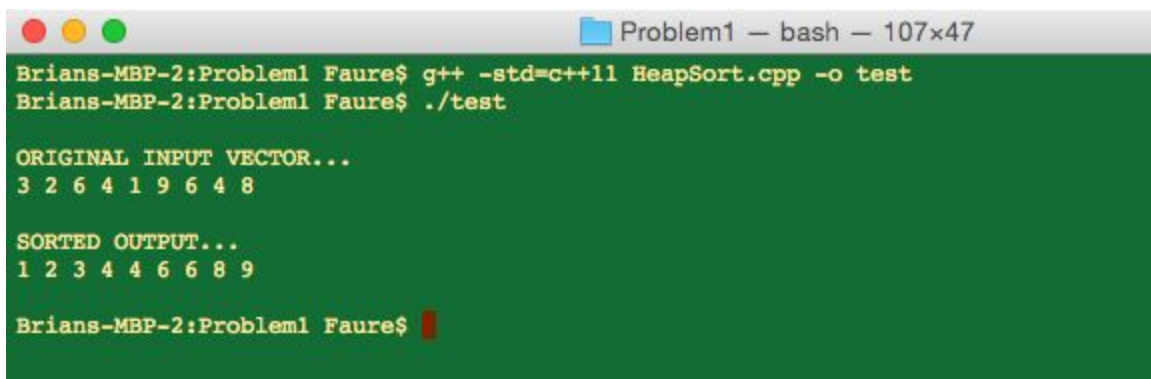
Brian Faure
RUID:150003563
NetID:bdf39
Prog. Methodology II
Homework #5

Problem #1

1. Implement HeapSort. Using the data set from earlier sorting homeworks, plot and analyze the runtime performance of HeapSort. [6 points]

→ HeapSort.cpp is located in the Problem1 Folder along with the provided datasets & screenshots

→ Running HeapSort.cpp with main that's commented out (to check it sorts properly):



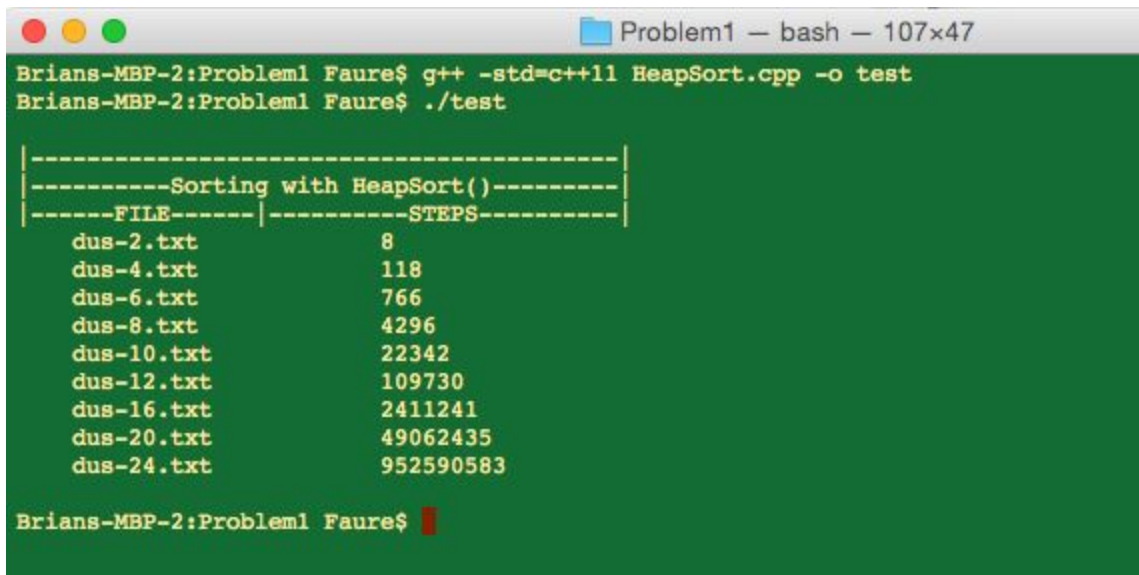
```
Problem1 — bash — 107x47
Brians-MBP-2:Problem1 Faure$ g++ -std=c++11 HeapSort.cpp -o test
Brians-MBP-2:Problem1 Faure$ ./test

ORIGINAL INPUT VECTOR...
3 2 6 4 1 9 6 4 8

SORTED OUTPUT...
1 2 3 4 4 6 6 8 9

Brians-MBP-2:Problem1 Faure$
```

→ Running HeapSort.cpp on provided datasets (with counters placed as they are in file):

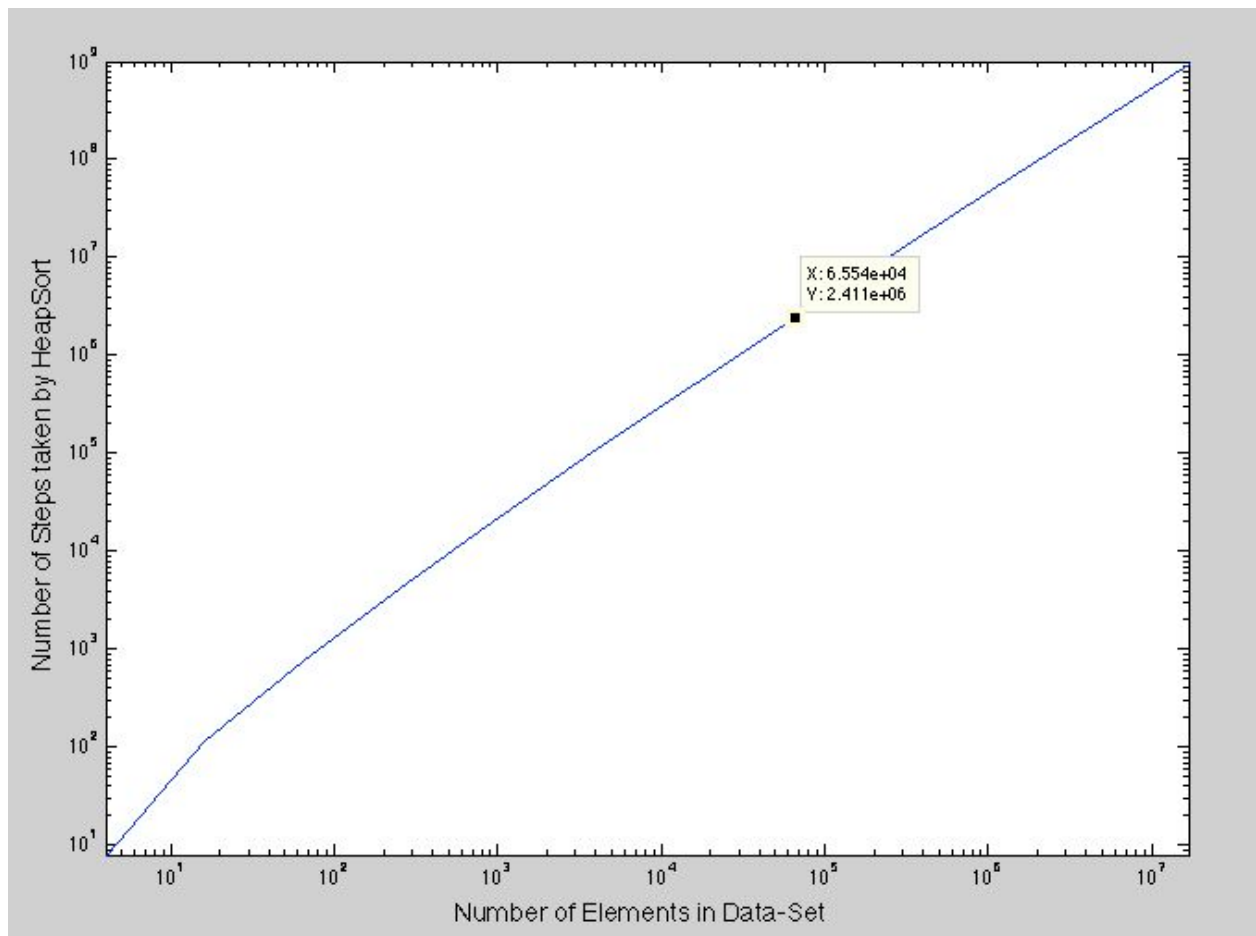


```
Problem1 — bash — 107x47
Brians-MBP-2:Problem1 Faure$ g++ -std=c++11 HeapSort.cpp -o test
Brians-MBP-2:Problem1 Faure$ ./test

-----Sorting with HeapSort()-----
-----FILE-----|-----STEPS-----|
dus-2.txt          |          8         |
dus-4.txt          |         118        |
dus-6.txt          |        766         |
dus-8.txt          |       4296         |
dus-10.txt         |      22342         |
dus-12.txt         |     109730         |
dus-16.txt         |    2411241         |
dus-20.txt         |   49062435         |
dus-24.txt         |  952590583         |

Brians-MBP-2:Problem1 Faure$
```

→ LogLog plot of number of inputs, n , vs. number of steps taken (complexity):



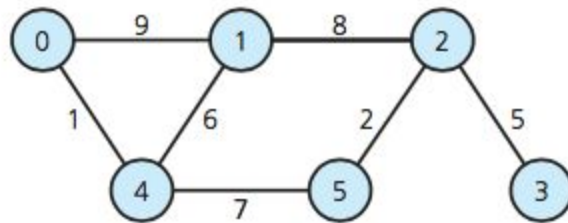
Given its number of comparisons, the HeapSort function should theoretically have about an $O(n \log n)$ complexity. This characteristic is visible by the fact the complexity is about linear with a slope slightly greater than 1 with the linear aspect meaning it more or less fits the form $y = ax^k$. The fact the slope is around 1 signifies that the exponential k value is slightly greater than 1 but certainly nowhere near 2 and this would be expected by $n \log(n)$.

Problem #2

2. (a) Pg 632, Exercise 1

1. Give the adjacency matrix and adjacency list for

a. The weighted graph in Figure 20-33



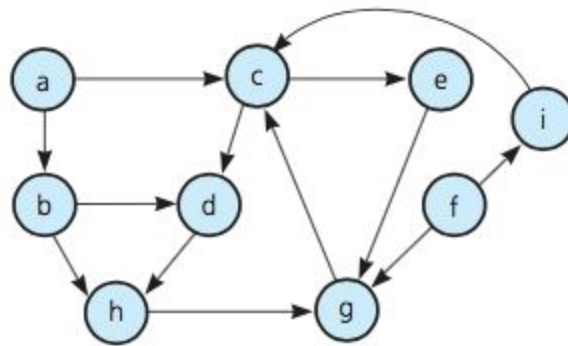
Adjacency Matrix:

Element	0	1	2	3	4	5
0	∞	9	∞	∞	1	∞
1	9	∞	8	∞	6	∞
2	∞	8	∞	5	∞	2
3	∞	∞	5	∞	∞	∞
4	1	6	∞	∞	∞	7
5	∞	∞	2	∞	7	∞

Adjacency List:

Element		Element Weight		Element Weight		Element Weight		Element Weight
0	→	1 9	→	4 1	→		→	
1	→	0 9	→	2 8	→	4 6	→	
2	→	1 8	→	3 5	→	5 2	→	
3	→	2 5	→		→		→	
4	→	1 6	→	0 1	→	5 7	→	
5	→	2 2	→	4 7	→		→	

b. The directed graph in Figure 20-34



Adjacency Matrix:

	A	B	C	D	E	F	G	H	I
A	0	1	1	0	0	0	0	0	0
B	0	0	0	1	0	0	0	1	0
C	0	0	0	1	1	0	0	0	0
D	0	0	0	0	0	0	0	1	0
E	0	0	0	0	0	0	1	0	0
F	0	0	0	0	0	0	1	0	1
G	0	0	1	0	0	0	0	0	0
H	0	0	0	0	0	0	1	0	0
I	0	0	1	0	0	0	0	0	0

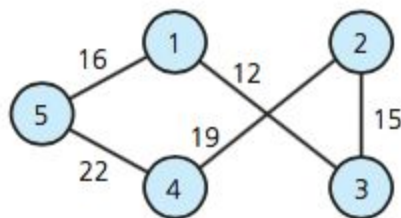
Adjacency List:

Element		Points To		Points To		Points To		Points To
A	→	B	→	C	→		→	
B	→	D	→	H	→		→	
C	→	D	→	E	→		→	
D	→	H	→		→		→	
E	→	G	→		→		→	
F	→	G	→	I	→		→	

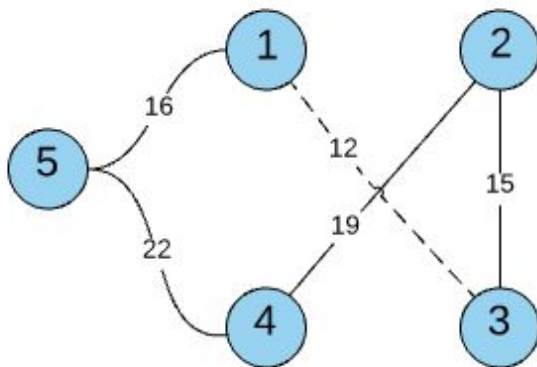
G	→	C	→		→		→	
H	→	G	→		→		→	
I	→	C	→		→		→	

(b) Pg 633, Exercise 11

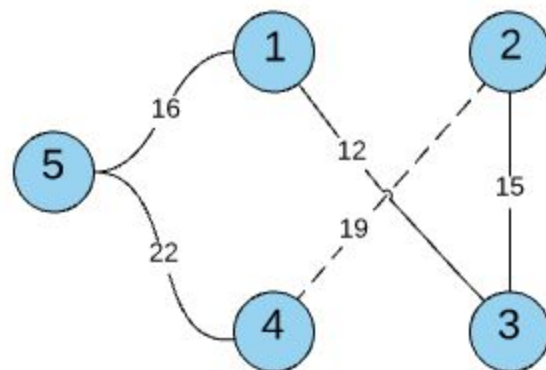
11. For the graph in Figure 20-38,



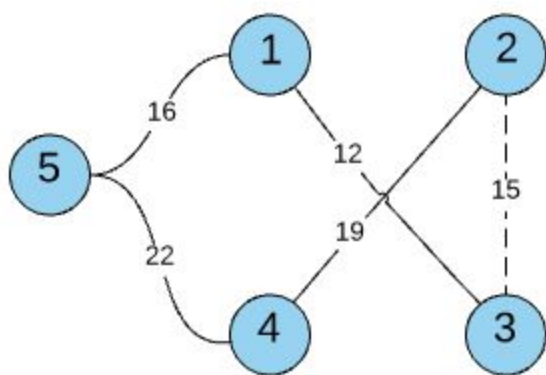
a. Draw all the possible spanning trees.



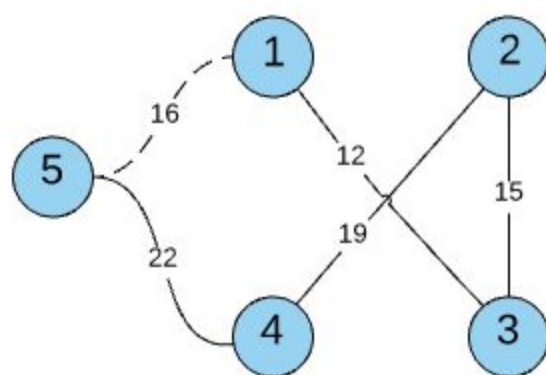
Cost = 65



Cost = 72

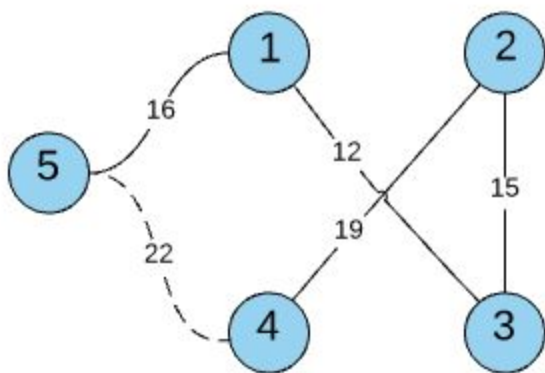


Cost = 69



Cost = 68

b. Draw the minimum spanning tree.

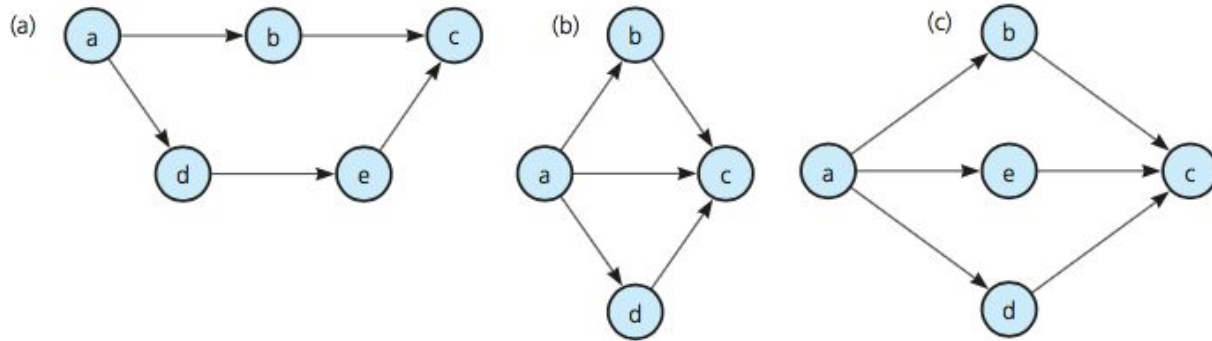


Cost = 62

Problem #3

3. Pg 632, Exercise 6 [4 points]

6. Using the topological sorting algorithm `topSort1`, as given in this chapter, write the topological order of the vertices for each graph in Figure 20-37.



Using `topSort1` function...

For Figure (a):

- Remove A & its edges
- Remove D & its edges
- Remove B & its edges
- Remove E & its edges
- Last vertex is C
- **Order is: A, D, B, E, C**

For Figure (b):

- Remove A & its edges
- Remove B & its edges
- Remove D & its edges
- Last vertex is C
- **Order is: A, B, D, C**

For Figure ©:

- Remove A & its edges
- Remove B & its edges
- Remove E & its edges
- Remove D & its edges
- Last vertex is C
- **Order is: A, B, E, D, C**

Problem #4

4. (a) Pg 633, Exercise 15 (You can use either C++ or Java or python).

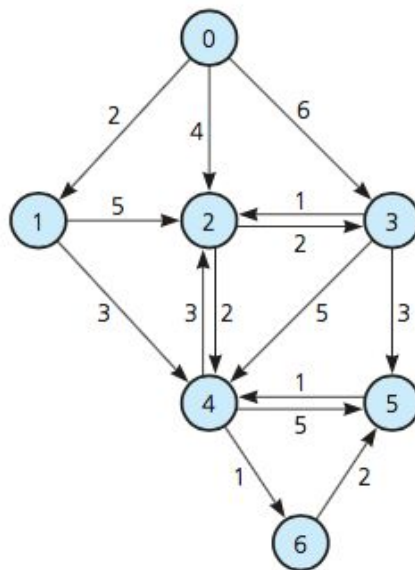
***15.** Implement the shortest-path algorithm in C++. How can you modify this algorithm so that any vertex can be the origin?

To select a different vertex for the origin, you can simply set its initial weight to zero in the adjacency matrix and move it to the top row (in my implementation). If you coded in a first step in the algorithm that checked the first column of the adjacency matrix for a value that equals zero then it would be able to identify which row corresponded to the origin without needing to manually move the origin row to the top. In my implementation the algorithm knows that the origin will have a value of zero and the origin adjacency row is already on the top of the matrix.

→ My algorithm is implemented with the Figure 20-39 as the input corresponding to problem b.)

(b) Implement this for Fig 20-39 using vertex 0 as the origin. [6 points]

FIGURE 20-39 A graph for Exercise 14



→ Initial Adjacency Matrix for Figure 20-39:

	0	1	2	3	4	5	6
0	0	2	4	6	∞	∞	∞
1	∞	∞	5	∞	3	∞	∞
2	∞	∞	∞	2	2	∞	∞
3	∞	∞	1	∞	5	3	∞

4	∞	∞	3	∞	∞	5	1
5	∞	∞	∞	∞	1	∞	∞
6	∞	∞	∞	∞	∞	2	∞

→ Initial Vertex Weight Matrix:

	0	1	2	3	4	5	6
<u>Weight:</u>	∞	∞	∞	∞	∞	∞	∞

→ See the Problem4 folder, ShortestPath.cpp file for the Shortest-Path C++ implementation.

→ Terminal Output when ShortestPath.cpp is run:

```

Problem4 — bash — 114x35
Brians-MBP-2:Problem4 Faure$ g++ -std=c++11 ShortestPath.cpp -o test
Brians-MBP-2:Problem4 Faure$ ./test

Shortest Path to Vertex 0 = 0
Shortest Path to Vertex 1 = 2
Shortest Path to Vertex 2 = 4
Shortest Path to Vertex 3 = 6
Shortest Path to Vertex 4 = 5
Shortest Path to Vertex 5 = 8
Shortest Path to Vertex 6 = 6

Brians-MBP-2:Problem4 Faure$

```