Hermmy Wang
704978214
Discussion 1B

Homework 4

1.(a)

heapify(array, n_remain, index):

```
  if n_remain equals 0:
    return heap

  root <-- index
  left <-- 2*index+1
  right <-- 2*index+2

  if root or left or right is less than 0:
    return heap

  root <--array[n-1-index]

  left child of root <-- array[n-1-left]
  heapify(array, n_remain-1, left)

  right child of root <-- array[n-1-right]
  heapify(array, n_remain-1, right)
```

Time complexity: O(n)
The goal is to build a max heap tree where the file accessed most frequently is placed at the least depth (or the root). Since the array is already sorted in an increasing order, we do not need to compare the elements anymore. We assign each value once to the left/right child of its parent, so the time complexity is O(n). The algorithm looks at the array from the highest index to the lowest index. index is the distance from the end of the array (e.g. the last item has an index of 0), and it is used to locate the root, the left child index, and the right child index.

0	1	2	4	6	7	9

```
        9
    7       6
  4   2     1   0
```

For example, 6 is array[4], and it is 2 indices away from the end of the array. We calculate its left child as array[7-1-2*2-1] = array[1] and right child [7-1-2*2-2] = array[0]. Recursively, we heapify each left and right child of the current root, until we have visited every element of the array and build a binary heap tree.

2.

heapify(array, i):

```
largest = i;
left = i*2+1;
right = i*2+2;

Set largest to the index with the largest value

if largest is not i anymore:
  swap(i, largest)
  heapify(array, largest)
```

buildHeap(array[n]):

```
for i <-- n/2-1 to 0:
  heapify(array, i)
```

Time complexity: O(n)

There are 1/2 of the elements doing one round of comparison, 1/4 doing two rounds, 1/8 doing three rounds... k is the height of the heap, or the maximum number of comparisons.

$$T(n) = n * \sum_{k=1}^{\log(n)} \frac{k}{2^k}$$

$$T\left(\frac{1}{2}n\right) = n * \sum_{k=1}^{\log(n)-1} \frac{k}{2^{k+1}}$$

$$T\left(\frac{1}{2}n\right) = T(n) - T\left(\frac{1}{2}n\right) = n * \sum_{k=1}^{log(n)} \frac{k}{2^k} - \sum_{k=1}^{log(n)-1} \frac{k}{2^{k+1}}$$

$$= n * \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^k} - \frac{1}{2^{k+1}} * k\right) \leq n$$

T(n) <= 2n

T(n) = O(n)

Since the question did not ask the list to be sorted, the algorithm stops after building a heap.

3(a).

Prove that there exists a tree T, such that the path from s to any other node v in T is the shortest path from s to v in G.

Proof by induction:

Base: For a graph with two vertices and an edge connecting them, the tree with the given property is the graph itself.

Induction: Suppose we have a directed weighted graph G = (V,E), and assume that there exists a directed tree T in G with root s, such that the path from s to any v is the shortest path. If we add one more vertex to G, G' = (V+1, E+n). The new added vertex has to have an edge directing from at least one of the vertices in G in order for s to reach all other nodes. Therefore, if we add to T the edge which comes from a vertex closest to s (e.g. we have a->v, b->v, and path(s->a)=3, path(s->b)=6. We should add a->v to T), we obtain T' with the given property. for G'.

3(b).

Starting off from the original two paths of identical path lengths, if we continue squaring each weight, the weights grow exponentially

$$w'(e) = w_1(e)^t$$

Larger weights will grow in an increasingly faster rate than the lower weights. Therefore, if we determine the shortest path in this way, we do not need to add each weight to determine the path length; instead, we only need to check which path contains the heaviest weight. This heaviest weight will eventually make the path it is at the longest. Once that path becomes identical or longer than the other path, it will only be longer after further squaring because the heaviest weight is growing even faster. Also, no matter how many edges a path is consisted of, the path with the heaviest edge will always eventually be the longest.

$y_1 = 3^t + 5^t + 7^t$

$y_1 = 34.8967514877$

$y_2 = 15^t$

$y_2 = 58.0947501931$

$t = 1.5$

If we determine the shortest path using this method, we do not need use addition or multiplication in the algorithm. We can search for the maximum weight among the two paths in O(n) and conclude the path without this weight to be the shortest path.

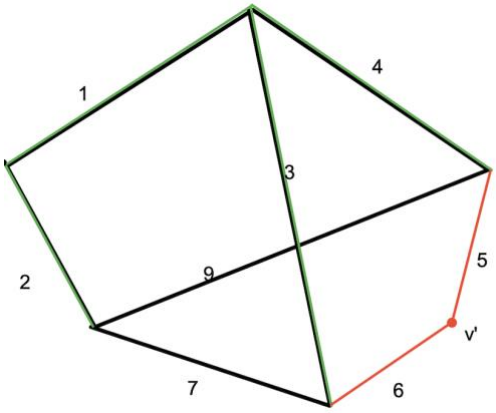
3(c).

Prove T' is a minimal spanning tree of G' if ignoring the directions on G'.
A minimal spanning tree is a tree that covers all the nodes and takes the minimal weights.
Since a node s can reach all other nodes, G is connected. Since each edge is bi-directional, we can consider G as undirected. Therefore, G is a *connected undirected* graph.

Proof by induction:

Base: In a tree with two vertices and a bi-directional edge, the minimal spanning tree is the graph itself.

Induction:
Suppose we have a graph G' = (V,E), and assume there is a minimal spanning tree T' rooted at an arbitrary node v in G'. If one new node v' is added to G', G' is now G'_k=(V+v', E+n). There are *n* new bi-directional edges added; each edge might be connected to any other vertices in G'. For each path, select the shortest path based on the idea in part(b). Avoid the path with the heaviest weight.



In this illustration, path 5 and path 6 is connected to path 3 and path 4, respectively. Although 3+6=4+5 and 3<4, we still choose the path 4+5 based on the idea in part(b). 6 is the heaviest edge and we should avoid it. Then, add v' and path 5 to T'_k.

Then we can add v' and the new edge in the shorted path to T'_k. T'_k will be stable as long as (1) every edge has a different weight and (2) we have squared the weights enough times so that it will always give us the same result.

3(d).

If we have two spanning trees with identical path sum, we may compare these two trees by comparison rather than addition.

Part(b) states that we can determine when two paths are identically long, we may square each weight and sum them until one path is longer than the other one. Therefore, no two paths have identical lengths unless all the weights on them are identical. Also as shown in part(b), the heaviest edge among all the edges in these two paths will lead to the longest path. Therefore, considering this idea from part(b), we can just compare the edge weights in each spanning tree to figure out which one is better.

As soon as we traverse all the path weights and find out the heaviest one, the spanning tree that contains this weight is not the optimal spanning tree.

In part(c), the path-selection algorithm I used is also the squaring algorithm. Recursively, every time I look for a new edge to add to the current MST, I avoid the path with the heaviest weight. Thus, for every node in the final MST, the path from it to the arbitrary root node will have the "lightest" path.

If we just look for the smallest tree by comparison, the final spanning tree also contains the "lightest" path from the root node to every other node, since the path-selection process is the same.

4(a)

(1) Keep a set as a rooted tree with depth O(logn). In order to look up the root for an arbitrary node, I also construct an array roots[v], with roots[i] returns the root of the tree that contains it.

(2) Query if there is a union containing x and y

findroot(sets, x)

```
  if roots[x] is x:
    return x

  else if roots[x] is not x:
    return findroot(sets, roots[x])
```

query(x, y):

```
  if findroot(sets, x) = findroot(sets, y):
    return true
  else:
    return false
```

(3) Union two sets

union(root, x, y)

```
  xroot <-- findroot(sets, x)
  yroot <-- findroot(sets, y)

  if xroot.height = yroot.height:
    roots[x] <-- yroot
    xroot.height < xroot.height+1

  else if xroot is higher yroot:
    roots[y] <-- xroot

  else:
    roots[x] <-- yroot
```

Time complexity: O(logn)

The time complexity for union-find algorithm is O(logn), since we arrange the sets into a rooted tree with height log(n). findroot() will take O(logn) because every time it recurs, the root will be at a higher level. In the worst case, it will run O(logn) times.

4(b).

Kruskal(Graph):

```
sort edges in increasing order
result[] <-- 0

for every edge in Graph:

  if query(src, dst):
    skip this edge

  else:
    add edge to result[]
    union(src, dst)

return result[]
```

Time complexity: O(|E|log|V|)

Querying every edge and unioning the vertices take 2log|V|. Therefore, inside each iteration of the loop, the time complexity is O(log|V|). There are |E| edges that we need to iterate. The total time complexity is O(|E|log|V|). Kruskal's algorithm works because it constantly looks up the lightest edge and check whether it completes a cycle (where the source vertex and the destination vertex are both visited by previous edges). Thus, the algorithm guarantees that we always find the lightest edge and never form a cycle.