# EECS 560 Final Exam

**Name: Alicia Maria Zavala Ferreira** **KU ID: 2951680**

The exam is closed-book and closed-notes.

(1) Show that all comparison-based sorting algorithms have a time complexity lower bound of $O(n\log n)$, where $n$ is the number of items to be sorted. You can assume each comparison only takes $O(1)$ time. (10pts)

The comparison-based sorting sorts the list based on comparison between the items. To show, let's have the elements be x, y, and z, after we compare them, we have that z < x < y, it might be possible that there weren't any comparisons, but these can reject conflicting rearrangements. So, every comparison reduces the number of rearrangements by at most half. The initial possible rearrangement is N!. In this case that's already at least O(log N!) to reduce all possible rearrangements to just one, which equals O(N log N) for any comparison-based sorting algorithm.

(2) Show the time complexity of the quicksort algorithm? Note that simply giving the time complexity will not worthy any credit; you will have to show the detailed analysis and proof. (10pts)

The best-case scenario is O(N log N), to analyze this, we can say that the best case comes when the pivot partitions into two equal sizes subsets every time, this, making it like a merge sort, but consider that you need to merge two sorted lists

The average-case scenario is O(N log N), normally, the partitions will have similar sizes and from that you can take its running time algorithm and simplify, after its simplified as much as you can you get O(N log N)

The worst-case scenario is O(N!), this will be when all the elements are either smaller or larger that the chosen pivot, so it will need N phases and in all of those phases there would need to be a comparison to the pivot

(3) Give the pseudocode of the Dijkstra's algorithm for finding the shortest path from an edgeweighted graph that contains no negative weight. (10pts)

We set the source as visited, while the other vertices as unvisited. Each time, we will select one unvisited vertex, which is adjacent to some visited vertices and has the shortest distance to the source among all other unvisited vertices. Let u be some visited vertex and v be some unvisited vertex that are adjacent to u, also denote the shortest path length between the source and u as $d_u$, and the edge length between u and v be $l_{u,v}$. The shortest distance between the source and w will be update as $d_v = d_u + l_{u,v}$

```
function dijkstra(vertex s)
#set initial values for distances and not visited nodes
for each Vertex v
      v.dist = infinity;
      v.known = false;
#we mark the first source as visited and distance here to 0 since it's the distance to the same
node
s.dist = 0;
while( there is an unknown distance vertex )
        v = smallest unknown distance vertex;
        v.known = true;
        for each Vertex w adjacent to v
           if( !w known )
              let cvw = cost of edge from v to w;
              if( v.dist + cvw < w.dist )
                 // Update w
                 decrease( w.dist to v.dist + cvw );
                 w.path = v;
endwhile
```

(4) Prove that the Dijkstra's algorithm is correct when assuming no negative weight. (10pts)

Let S the set of vertices that have been visited, we are currently at s with edges u and x, and these have edges v and y respectively, and v is and edge of y. Path P' is to x and path Pu is to u. If distance $d_v = d_u + l_{u,v}$ is smaller than $d_y = d_x + l_{x,y}$ then includes v instead of y to S since it guarantees a shorter path.


If there are no negative weights, then all this will hold, let's say $d_y > d_v$, if there aren't any negative paths, then the length path Py,v is positive, saying that $d_y + P_{y,v} > d_v$, since vertices x and y are chosen arbitrarily, it follows that $d_v$ is the shortest path for all the cases.

(5) What would happen if you run Dijkstra's algorithm on a graph with negative edges? If you think the algorithm will still work, prove it. Otherwise given a counterexample to show why it will not work. Note that simply answering will or will not work worth no credit. (10pts)

Let S the set of vertices that have been visited, we are currently at s with edges u and x, and these have edges v and y respectively, and v is and edge of y. Path P' is to x and path Pu is to u. If distance $d_v = d_u + l_{u,v}$ is smaller than $d_y = d_x + l_{x,y}$ then includes v instead of y to S since it guarantees a shorter path.

x - y
S < |
u - v

If we have negative edges on the graph for this, it will give out incorrect lengths and choose the wrong edges to add to S. An example could be if we let the distance y be 10 and distance v 7, and the Path Py,v -5. The shortest path will be y, which would be 5, but not going through u, dv should be 5 instead of 7

(6) Given the high-level idea for the array-based implementation of the union-find data structure. Show that you can perform find in $O(1)$ time and union in $O(\log n)$ time, where $n$ is the total number of elements in the disjoint sets. (10pts)

Since we are using an array, let our array be A**,** we will use A[i] to store each corresponding key to its element.

Example = array [ 1, 3 ,4,6]
We know that the first parent has one child , the second one has 3 children, and so on.

Find(x) takes O(1) time, since we return A[x], since we know that it stores the key that we are looking for.

Union(x, y) takes O(n) time, since in merges two sets through relabeling, we first get the number of children for correspondent to the parents ( A[x] A[y] ) and scan the entire array to relabel A[y] to A[x] or the other way around.

(7) Describe the heap property and implementation of the priority queue. Show that you can perform enqueue() and dequeue() in $O(\log n)$ time, where $n$ is the number of elements in the priority queue. (10pts)

The heap property is similar to a binary search tree, but in this case a heap is full that's why you use an array to store it, and not balanced(compared to BST) and the parents are smaller than its respective children nodes, and it does not matter which child is larger, which we know is not the case in BST. Given that we have an array, we also can get rid of unnecessary pointers. For any element in the array, we can derive its index from its position in its "binary tree" that we mentioned before. In an array, enqueue performs in O(log N) time when using a binary search tree or arrays, where finding the lowest rightest element to then enqueue would take O (log n) time and dequeue would be removing the root but finding the next value to be the root would take O(log N).

(8) Describe the implementation of the binomial queue. Show that you can perform merge() (which combines to binomial queues into one) in $O(\log n)$ time, where $n$ is the total number of elements in both binomial queues. (10pts).

Merge time complexity is O(log(n)) since we know that the binomial queue that contains n elements has log(n) binomial trees, because each tree is at least twice larger than the previous one. Merging each pair of binomial trees only takes O(1) time

(9) Sort the following texts in ascending lexicographic order using radix sort (from the last to the first position). Show the complete intermediate results for each pass to receive full credit. (10pts) Hint: empty space should be lexicographically ordered before any non-empty character.

Items to be sorted: “google” “some” “goggle” “from” “the” “google” “website”
E,m
-> google, some, goggle, the, google, website
-> from
L,h,m,o,t
->the
->google, goggle, google
-> some
->from
->website
G,I,o,r,t
-> google, goggle, google,
-> website
-> some
-> from
-> the
B,f, o, s ,t
-> website
-> from
-> goggle, google, google,
-> some
-> the
E,f,g,s,t
-> website
-> from
-> goggle, google, google
-> some
-> the
F,g,s,t,w
-> from
-> goggle, google, google
-> some
-> the
-> website
Final
“from” goggle” “google” “google” “some” “the” “website”

(10) In the context of C++ STL, explain what are “sequential container”, “associative container”, and “container adapter”. Given two examples for each of the three categories. (10pts)

These containers are what we can call the data structures, or how we hold the data in the code. More specifically, the sequential containers are linear data structures, for example vectors and list. The associative containers would be more like mapping, they support keys for the values, examples are sets and maps. The container adapter can put constrains when holding the data, examples include queue and stack.