CECS 328 Final

1)

```
a = randFun(n) // size n
Fun(a, 0, n-1);

Fun(a, start, end)
  n = end - start;
  if n<=1
    return a[end]*5 -8;
  else
    newEnd = start + n/2;
    if flag(n)      // random true/false, θ(1)
      sol = Fun(a, start, newEnd);
    else
      sol = 1;
    end
    ans = 0;
    itr = 0;
    while itr < a.length()
      for i=1 : a[itr]
        ans += sol;
      end
      itr++;
    end
    return ans;
end
```

Best Case:

flag(n) always sets sol = 1.

$n <= 1$: $\Theta(1)$

$n > 1$

Series of $\Theta(1)$ lines

while loop : itr < a.length

- n (a is size n)
- nested for loop: i=1 : a[itr]
  - $\sum_{i=0}^{n-1} a[i] = \log n$
- end
- itr++

end

return

$\Omega(n \log n)$

*based on comment

Worst Case

flag(n) always calls recursive function

$T(n) = T(n/2) + n \log n$

Step	Size	Tree
0	$n-1$	$n \log n$
1	$\frac{n-1}{2}$	$\frac{n}{2} \log(\frac{n}{2})$
2	$\frac{n-1}{4}$	$\frac{n}{4} \log(\frac{n}{4})$
⋮	⋮	⋮
n	1	$n \log n + \frac{n}{2} \log \frac{n}{2} + \frac{n}{4} \log(\frac{n}{4}) + \cdots \frac{n}{2^{k-1}} (\log \frac{n}{2^{k-1}}) + 1\,\Theta(1)$

$1 = \frac{n-1}{2^k}$

$k = \log n$

$2^k = n-1$

back →

$$\left(\sum_{i=0}^{k-1} \frac{n}{2^i} \log \frac{n}{2^i}\right) + 1 \quad \rightarrow \quad \left(\sum_{i=0}^{k-1} \frac{1}{2^i} + \sum_{i=0}^{k-1} (1) \log n - \sum_{i=0}^{k-1} i \log n\right)$$

$$n\left(\sum_{i=0}^{k-1} \frac{1}{2^i} + \log n - i(\log 2)\right) + 1$$

$$n\left(\frac{2 - \cancel{1}^{0}}{\cancel{2^{k-1}}} + (k-1)\log n - \frac{k-1(k)}{2} \log 2\right) + 1$$

$k = \log n$ $\quad$ $n \log^2 n$ + ~~$\log^2 n \log 2 + 1$~~

$T(n) = T(n/2) + n \log n$

$n \log^2 n + n \log n$

$O(n \log^2 n)$

2) Sort numbers to [1, 2, 3, 4] (any order)
Can only swap two numbers at a time

Graph and Nodes method

- Every possible ordering of the numbers represents a node.
- A node's neighbors represents valid swaps

- Find minimum # of swaps by performing BFS and storing the minimum value from initial to end [1, 2, 3, 4]

Iterative

- Check an index. If the index != value, swap it to the correct index.
- Use an array of booleans to track which indices are in the right position. X not using this step
- For every swap add 1 to min.
- If we visualize the minimum swaps as a connected tree, then there are at most V-1 edges.

Iterative probably runs faster than the graph approach.

- 4 elements vs. permutation of nodes > 4
- Since elements are always swapped to correct position, no unnecessary runtime is used.
- Graph method would have to traverse all adjacent nodes to exhaust possible min paths.

```
minSwap(int[] arr)
    min=0;
    for(int i=0; i<arr.length && min < arr.length-1; i++)
        if(arr[i] != i+1)
            int temp = arr[i]
            arr[i] = arr[temp];
            arr[temp] = temp;
            min++;
            i--;
        end
    end
    if (min==0)
        print("no swaps");
    else
        print("# swaps");
    end
end
```

pseudocode

Problem 2

```java
public static void minSwap(int[] arr)
{
    int min = 0;
    for (int i = 0; i < arr.length && min < arr.length - 1; i++) {
        if (arr[i] != i + 1) {
            int temp = arr[i];
            arr[i] = arr[temp - 1];
            arr[temp - 1] = temp;
            min++;
            i--;
        }
    }
    if (min == 0)
        System.out.println("No need to do any swaps! You win!");
    else
        System.out.printf("The min nr of swaps is: %d\n", min);
}
```

```
Example 1: Input: [1, 2, 3, 4]
No need to do any swaps! You win!

Example 2: Input: [2, 3, 1, 4]
The min nr of swaps is: 2

Example 3: Input: [3, 4, 2, 1]
The min nr of swaps is: 3

Process finished with exit code 0
```

Runtime

```
public void minSwap(int[] arr)
{
    int min = 0;
    for (int i=0 ; i< arr.length && min < arr.length -1; i++){
        if(arr[i] != i+1) {
            int temp=arr[i];
            arr[i] = arr[temp-1];
            arr[temp-1] = arr[i]
            min++;
            i--;
        }
    }
    if (min==0)
        print(~~~~);
    else
        print(~~~~);
}
```

No swaps

Entire array iterated.

$\Theta(n)$

Max Swaps

Equal to n-1. Loop exits.

$\Theta(n-1) = \Theta(n)$

Worst Case

n-2 swaps performed

Then entire loop iterated.

$O((n-2)(n)) = n^2$

Vs. permutations from graph method

$P(n,n) = \frac{n!}{(n-n)!} = \frac{n!}{0!} = \frac{n!}{1}$

$n! > n^2$

$O(n^2)$

where n equals total numbers to swap.

3) Minimize Trip Cost.

Defined start and End nodes.

Kruskal/Prim's Algorithm

Finds minimum spanning time

X only works on undirected graphs

given graph is directed

Dijkstra's Algorithm

- Finds shortest dst need from initial to all reachable nodes
  - Modify algorithm to find node to node dst rather than all reachable ones

Breadth-First Search

Traverses a node's neighbors

Repeated for all paths

-Rather than using dst, track weights

-Can also track parents

Starting from initial node:

1. Choose an adjacent node and add its weight to the total.
2. Push and pop queue according to BFS algorithm.
3. If the weight at this time is less than the current node weight, set new minimum path. Repeat for other paths
4. At the end, add the minimum path to a list and return.

Pseudocode

```
minimumCost(initial)
    Queue q = new LinkedList();
    q.push(initial); //// Node path;
    while (q.size() > 0) {
        Node next = q.pop();
        for (Node n: next.adj())
            weight = next.dst + w(next, n);
            if (n != initial && (dst == 0 || weight < dst)
                q.add(n);
                n.parent = next;
                n.dst = weight
            end
        end
        if (next.adj().size() == 0) // end node
        | path = next;
        end
    end

    print weight and path
end
```

dst is modified to track min weight from initial to n. initially 0.

Problem 3

```
public void minimumCost(Node initial)
{
    Queue<Node> q = new LinkedList();
    Node path = null;
    q.add(initial);
    while(q.size() > 0) {
        Node next = q.remove();
        for (Edge edge : next.getAdj()) {
            Node n = edge.getNode();
            int weight = next.getDst() + edge.getWeight();
            if(((weight < n.getDst() || n.getDst() == 0))
                    && (n != initial || !q.contains(n))) {
                q.add(n);
                n.setParent(next);
                n.setDst(weight);
            }
        }
        if (next.getAdj().size() == 0)
            path = next;
    }
    System.out.printf("Minimum Cost: %d\n", path.getDst());
    System.out.println("Path (from destination)");
    while (path != null) {
        System.out.println(path + " ");
        path = path.getParent();
    }
}
```

```
Minimum Cost: 490
Path (from destination)
Glacer N-Park
Salt Lake City
Las Vegas
Long Beach

Process finished with exit code 0
```

All code not specified in brackets run in $\theta(1)$ time.

```
public void minimumCost(Node initial)
    Queue<Node> q = new LinkedList();
    Node path = null;
    q.add(initial);
    while(q.size() > 0)
        Node next = q.remove();
        for (Edge edge : next.adj())
            Node n = edge.getNode();
            int weight = next.getDst() + edge.getWeight();
            if((weight < n.getDst() || n.getDst() == 0)
                && (n != initial || !q.contains(n)))
                q.add(n);
                n.setParent(next);
                n.setDst(weight);
            end
        end
        if (next.getAdj().size() == 0)
            path = next;
        end
    end
    System.out.printf(~~~~);
    System.out.println(~~~);
    while (path != null)
        System.out.print(~~~);
        path = path.getParent();
    end
end
```

!q.contains(n)

Prevents the queue from adding duplicate nodes due to min weight updates.

While loop: $O(V)$

for loop: nested in while loop

checks all edges of a node

↓

Check all edges of the graph $O(E)$

$2|E|$ $|E|$ Directed

Worst Case

Minimum path contains all vertices in graph

$O(V)$

only one path

Runtime: ~~$O(1)$~~ $+ O(V) + O(E) + O(V)$

$O(2V + E) = \boxed{O(V+E)}$