CSE 6140 / CX 4140

Computational Science & Engineering (CSE) Algorithms

Homework 2

1. (20 points) Consider sorting n numbers stored in array $A[1:n]$ by first finding the smallest element of $A[1:n]$ and exchanging it with the element in $A[1]$. Then find the smallest element of $A[2:n]$ and exchange it with $A[2]$. Then find the smallest element of $A[3:n]$, and exchange it with $A[3]$. Continue in this manner for the first $(n-1)$ elements of A. Write pseudocode for this algorithm, which is known as **selection sort**. Why does it need to run for only the first $(n-1)$ elements, rather than for all n elements? Give the worst-case running time of selection sort in O-notation. Is the best-case running time any better?

Solution: The pseudocode is in Figure 1.

```
Algorithm : Selection sort
-----------------------------------------
Inputs :  A  <--  Array to be sorted
          n  <--  Length of array A
Output :  Sorted array A

for i = 1 to (n-1) :
    min = i
    # Finding the min in A[i:n]
    for j = i+1 to n :
        if A[j] < A[min] :
            min = j
    # swap min with A[i]
    t = A[i]
    A[i] = A[min]
    A[min] = t
return A
```

Figure 1: Selection sort pseudocode

From the pseudocode we note that the outer loop considers the sub-array $A[i:n]$, finds the smallest element in this subarray, and swaps it with $A[i]$. So, as i increases, we have $A[1:i]$ to be sorted. If that wasn't the case there would have been $m < n < i$ with $A[m] > A[n]$. However, this would have been resolved in $i = m$ iteration and so we would have the subarray $A[1:i]$ to be sorted. So, after we are done with $i = n-1$, we have the whole array sorted, else the last

element is smaller than one of the previous ones. However, this is not possible since we find the minimum element in the subarray at each outer loop execution. Thus, after $i = (n-1)$, sorting ends.

For time complexity, we see that the outer loop runs for $(n-1)$ steps and the inner loop runs for $(i+3)$ steps (i steps for the loop and 3 steps for the swap). So, the total steps in our algorithm is:

$$T(n) = \sum_{i=1}^{(n-1)} (i+3) = (1+2+\cdots+(n-1)) + 3(n-1) = \frac{(n-1)(n-2)}{2} + 3n - 3$$

$$\Rightarrow T(n) = \frac{n^2-3n+2}{2} + 3n - 3 = \frac{(n^2+3n-4)}{2} = O(n^2)$$

Note that, even in the best case when array A is already sorted, the number of comparisons remains the same and the swap steps are also executed. So, $T(n)$ does not change for the best case either. So, we have the same time complexity of $O(n^2)$ for the best case as well.

---

2. Let $A[1:n]$ be an array of n distinct numbers. If $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A.

a. (5 points) List the five inversion of the array $\{2, 3, 8, 6, 1\}$

Solution: The five inversions are:

1. $(i, j) = (1, 5)$: $2 > 1$
2. $(i, j) = (2, 5)$: $3 > 1$
3. $(i, j) = (3, 4)$: $8 > 6$
4. $(i, j) = (3, 5)$: $8 > 1$
5. $(i, j) = (4, 5)$: $6 > 1$

b. (5 points) What array with elements from the set $1, 2, 3, \ldots, n$ has the most inversions? How many does it have?

Solution: The array with all elements in descending order: $\{n, (n-1), \ldots, 3, 2, 1\}$ has the maximum number of inversion since for every pair of $i < j$, the corresponding array elements follow $A[i] > A[j]$. Since every pair of i, j is an inversion, the number of inversions will be $\binom{n}{2} = \frac{n(n-1)}{2}$.

c. (10 points) Give an algorithm that determines the number of inversions in an array of n elements in $O(n \log n)$ worst-case time. (*Hint*: Modify merge sort).

Solution: Refer to Figures 2 and 3 for the pseudocode.

The analysis of the algorithm is as follows:
Divide: We divide the array into two halves at each step. The base case is a 2-element array. Note that each division takes $O(1)$ time since we just have to

```
count_inversions(A):
    n = length(A)
    # base case
    if n == 2:
        if A[1] > A[2]:
            t = A[1]
            A[1] = A[2]
            A[2] = t
            return A, 1  # one inversion
        else
            return A, 0

    # divide and solve sub problems
    P, i1 = count_inversions(A[1 : n/2])
    Q, i2 = count_inversions(A[n/2 + 1 : n])

    # Merge
    M, k = merge(P, Q)
    i = (i1 + i2 + k)

    return M, i
```

Figure 2: Pseudocode for the *count_inversions* function

```
merge(P, Q):
    p = 1          # Pointer for P
    q = 1          # Pointer for Q
    M = array()    # empty array
    K = 0

    while p ≤ length(P) and q ≤ length(Q)
        if P[p] < Q[q]:
            M.add(P[p])
            p += 1
        # Inversion exists
        else:
            M.add(Q[q])
            K += (length(P) - p + 1)
            q += 1

    if p < length(P): Add remaing to M
    if q < length(Q): Add remaining to M

    return M, k
```

Figure 3: Pseudocode for the *merge* subroutine used in the *count_inversions* function

find the midpoint of the array. So, the whole **divide operation takes at most linear time**.

Base case: For the base case, if the two elements are out-of-order ($i < j$ and $A[i] > A[j]$), then we swap the two and increment `inv_count` by 1. Note **solving each base case takes constant time** since both these operations take constant time.

Merge: When we have to merge two arrays A_1 and A_2 to form A'. So:

$$\texttt{inv}(A') = \texttt{inv}(A_1) + \texttt{inv}(A_2) + \text{ (\# of inversions between } A_1 \text{ and } A_2)$$

We have the values of `inv(A_1)` and `inv(A__2)` from the recursion tree, and we have sorted arrays A_1 and A_2. Now, for the third term, we do the following:

- Maintain pointers p_1 and p_2 to A_1 and A_2, respectively. Begin with the first element in both.
- Compare the two elements $A_1[p_1]$ and $A_2[p_2]$. Append the smaller element to the array A'. If the smaller element is from A_1 just increment pointer p_1. Otherwise, increment `inv(A')` by number of elements left in A_1 (i.e., $(len(A_1) - p_1 + 1)$) and then increment pointer p_2.

Since during each merge we just traverse the array, **each merge takes linear time**.

So, this perfectly fits Template 1 discussed in class and thus the running time is $O(n\ log\ n)$ in the worst-case (whole array is in descending order).

---

3. (20 points) You are a hiker preparing for an upcoming hike. You are given heights, a 2D array of size $rows \times columns$, where $heights[row][col]$ represents the height of cell (row, col). You are situated in the top-left cell, $(0, 0)$, and you want to travel to the bottom-right cell, $(rows - 1, columns - 1)$ (i.e., 0-indexed). You can move up, down, left, or right, and you wish to find a route that requires the minimum effort.

A route's effort is the maximum absolute difference in heights between two consecutive cells of the route. Example in Figure 4.

Provide an algorithm that returns the minimum effort required to travel from the top-left cell to the bottom-right cell. Prove the correctness of your algorithm and state its running time.

Solution: We approach this problem similar to how we tackled the single source shortest path problem in graphs. Given that we start at cell $(0, 0)$ and can traverse only up, down, left and right at any step, we can look at this grid as a graph where each cell is a vertex with a given weight and the neighbors of each node (i, j) are $\{(i-1, j), (i, j-1), (i+1, j), (i, j+1)\}$ whichever are within the bounds of the grid. The idea of the algorithm is as follows:

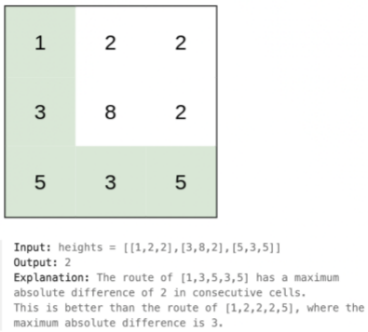

Figure 4: An Example for Q3

- Initialize a new empty matrix B of same size as A where $B[i,j] = (e,(x,y))$ where e is the minimum effort of traveling to (i,j) from $(0,0)$, and (x,y) is the predecessor of cell (i,j) in this minimum effort path discovered so far.
- We initialize all $B[i,j] = (\infty,(0,0))$ (set to a really large number like 10^8) for all i,j and have $B[0,0] = (0,(0,0))$ since effort required is 0 for the first cell.
- Initialize a queue containing $(0,(0,0))$ (the element represent *(effort, cell)*). In fact, since the cell with lower absolute difference value is preferred, we could maintain this as a priority queue.
- Until the queue is empty or we reach the destination
  - Pop the queue to get the element in the front. Let it be $(e,(x,y))$.
  - If $(x,y) == (rows-1, columns-1)$ (i.e., we have reached the destination), return e as the minimum effort to reach destination $(rows-1, columns-1)$
  - Compute the cost of reaching its neighbors from (x,y). If the new cost k for a neighbor (x_1,y_1) is lower, then update $B[x_1,y_1] = (k,(x,y))$ since in the updated min effort path, (x,y) is the parent node of (x_1,y_1). Also add $(k,(x_1,y_1))$ to the queue for further computations.
  - If the queue empties before reaching $(rows-1, columns-1)$, then there is no path to that cell and thus we return -1.
- By tracking the predecessor in each tuple from $B[rows-1, columns-1]$, we can obtain the minimum effort path. (If $B[rows-1, columns-1] = (e,(m,n))$, add $(rows-1, columns-1)$ and (m,n) to empty path and next go to $B[m,n]$ to get the predecessor of (m,n) in the minimum effort path and so on).

Refer to Figure 5 for the algorithm pseudocode.

Correctness of the Algorithm: The correctness proof is very similar to that of Djikstra's algorithm. Let S be the set of grid cells discovered ($B[i, j]$ are finite for these). Now we prove the following claim:

Claim: *Out of all $v = (i, j) \in S$, the path P_v formed by tracing the predecessor of each node starting from v gives the minimum effort path to v from $s = (0, 0)$*

Proof: We prove this by induction on size of S.

- BASE CASE ($|S| = 1$): Now S contains only s and minimum effort to reach it is 0 since we are already there.
- INDUCTION HYPOTHESIS: Suppose that the result is true for $|S| = k$ for some $k \geq 1$.
- INDUCTIVE STEP: We grow the set S by adding grid cell $v = (i, j)$ with the cell $u = (m, n)$ being its neighbor in set S. By the inductive hypothesis, P_u is the minimum effort $S - u$ path. We note that $s - v$ is the minimum effort $s - v$ path. Consider an alternate $s - v$ path P via cell x in S and let y be the first cell on this path outside S (neighbor of x). First, the priority queue chose u over x since exploring v would take less effort than exploring y. So, by the time the path P leaves S, the effort has exceeded the $s - v$ path containing P_u. Thus, now for $S' = S \cup \{v\}$, we have shortest $s - u$ path P_u from S and path $P_v = P_u \cup \{(u, v)\}$ to be shortest $s - v$ path in S'. So, the hypothesis is true for $|S| = k + 1$.
- Hence, our algorithm is correct for any $|S| \geq 1$

Time complexity: $r :=$ Number of rows in A and $c :=$ Number of columns in A

- In the initialization step, the expensive operation is setting each element of B to $(\infty, (0, 0))$, which takes $O(rc)$ time (since there are rc elements in B)
- Then, compute minimum effort for each element in B - so rc subproblems.
- For each subproblem, we recalculate costs for the neighbors (at most 4 neighbors for each) and update B and add an element to the queue. Thus, each sub problem $O(log\ rc)$ time due to the addition and deletion to the priority queue.

Thus, the running time of the algorithm provided is **O(rc log rc)**.

```
Minimum_effort ( A ) :
    r = A.shape[0]  # rows
    c = A.shape[1]  # columns
    dx = [-1, 0, 1, 0]
    dy = [0, -1, 0, 1]
    B = array(r,c)  # empty array of same size as A
    for i = 0 to r-1
        for j = 0 to c-1
            B[i,j] = (1e8, (0,0))  # (effort, parent)
    B[0,0] = (0, (0,0))  # min effort for (0,0) is Zero
    q = queue( (0, (0,0)) )  # priority queue with element (effort, cell)
    while q is not empty
        p = q.pop()
        e = p[0]
        x = p[1][0]
        y = p[1][1]
        # If we reached destination return effort
        if x == r-1 and y == c-1
            return e
        # else compute for neighbors
        for i = 0 to 3
            # coordinates of neighbor
            x1 = x + dx[i]
            y1 = y + dy[i]

            if x1 >= 0 and x1 < r
                if y1 >=0 and y1 < c
                    K = max{ e , |A[x,y] - A[x1,y1]| }
                    # lower effort
                    if K < B[x1,y1][0]
                        B[x1,y1] = (K, (x,y))  # update effort and parent
                        q.add( (K, (x1,y1)) )  # add updated tuple to queue
    return -1
```

Figure 5: Algorithm for finding minimum effort path from cell $(0, 0)$ to $(r-1, c-1)$ in a grid with r rows and c columns.

4. (20 points) There are n cities in a state. Given any two cities x and y, we know the cost of building a road between x and y. We want to build a network of roads such that any city can be reached from any other city (through a direct road or via intermediate cities). We want to do this at minimum cost. Provide an algorithm to solve this problem. Prove it's correctness and state its running time.

Solution: Since, we want to connect the n cities at minimum cost, it is required that we should restrict to one path per pair of cities since additional paths add to the cost. So, if we construct a graph G with each city as a node ($|V(G)| = n$), and connect each pair of nodes (since potentially every pair is connected and has an associated cost), then the cost of building a road between cities x and y is the edge weight of the edge (x, y) in the constructed graph. In essence, we wish to span all the n nodes with one path per pair of cities and at minimum cost. So, the problem reduces to finding the Minimum Spanning Tree of this complete graph $G(V, E)$.
For this, we use the Kruskal's algorithm (Refer to Figures 6 and 7).

```
KRUSKAL(G, w):
    G ← network whose MST we compute
    w ← edge weights
    parent = [ ]
    T = [ ]    # stores edges of MST
    # Put each vertex in different sets
    for v in G.V:
        parent.add(v)
    # Sorted edges in increasing order of weight
    L = sorted(list(G.E))

    # Iterate through each element in L to either choose or omit
    for (u,v) in L:
        # To ensure no u-v paths exist
        if FIND-SET(parent, u) ≠ FIND-SET(parent, v)
            # Add edge to MST and combine trees
            T = T.add((u,v))
            parent = UNION(parent, u, v)

    return T
```

Figure 6: Pseudocode of Kruskal's algorithm to find the Minimum Spanning tree of a graph G given edge weights w

Algorithm Correctness: We utilise the following property in the proof:

```
FIND-SET ( parent, x)
  # return representative of set which has x
  if  parent[ x] != x
      return   FIND-SET (parent, x)
  else
      return  x

UNION ( parent, x , y )
    a =  FIND-SET (parent, x)
    b =  FIND-SET (parent, y)

    if  a ≠ b
        parent [ a] =  b    # set root of one as parent
                               of other root
    return parent
```

Figure 7: Pseudocode of *UNION()* and *FIND-SET()* utilized in the Kruskal's Algorithm.

CUT PROPERTY: Let $S \subset V$ be a non-empty set of nodes. Let e be a minimum cost-edge with one end in S and the other in $V - S$. Then, every minimum spanning tree of G contains e (Proof done in the lectures).

We require to prove the following claims:

(i) **The output T is a tree:**

- First, if suppose the output T was not connected, then there will be a partition S and $V - S$ between which no edges are there. However, since the original graph is connected, there is an edge connecting these two subsets of nodes, and the algorithm would add it since adding one edge connecting two acyclic components won't create a cycle. Thus, T is **connected**.
- Also, we ensure that there is no $u - v$ path exists before adding edge (u, v), the edge addition does not form a cycle, and thus the output T is **acyclic**. Thus, the **output T is a tree**.

(ii) **The output T spans G:**

- Since the algorithm starts with all the nodes and iteratively adds edges connecting them to get a connected graph, T **spans the original**

graph G.

(iii) T **is a minimum spanning tree:**

- Consider any edge $e := (u, v)$ added by the Kruskal's algorithm. Before adding it, the vertex u is part of a connected component S and v is in $V - S$. The algorithm chooses the minimum cost edge among the remaining edges that has one end in S and the other in $V - S$. So, e is the min-cost edge that has one edge in S and the other in $V - S$.
- Thus, by the **CUT PROPERTY** mentioned above, e belongs to every minimum spanning tree of G and thus the output T is a **Minimum Spanning Tree**.

Running Time: Here $G(V, E)$ is complete graph with n nodes (so $O(n^2)$ edges). We derive the running time of the algorithm in terms of n as follows:

- The initial step of forming a set for each $v \in V(G)$ takes constant time per vertex. So, this step takes $O(n)$ time.
- The next step of sorting the edges by weight is $O(|E(G)|\ log\ |E(G)|)$ in the worst case (using Merge sort or similar sorting). Here $|E(G)| = O(n^2)$, so this step takes $O(n^2\ log\ n)$ time.
- For the final step of looping through the sorted edges: We have $|E(G)|$ iterations and for each iteration we just perform Union-find algorithm. Each find operation traverses the whole height of the tree containing the node, which takes $O(log\ |V(G)|)$ in the worst case. Also, the union operation involves two find operations followed by a constant time operation, thus taking $O(log\ |V(G)|)$ in the worst case. So, $O(n^2)$ iterations each with $O(log\ n)$ operation. Thus this step takes $O(n^2\ log\ n)$ time.

Thus the overall running time is:

$$\begin{aligned} &O(|V(G)|) + O(|E(G)|\ log\ |E(G)|) + O(|E(G)|\ log\ |V(G)|) \\ &= O(|E(G)|\ log\ |E(G)|) \qquad (\text{since } |E(G)| > |V(G)|) \\ &= \mathbf{O(|E(G)|\ log\ |V(G)|)} \qquad (\text{since } |E(G)| = O(|V(G)|^2)) \end{aligned}$$

In terms of $n(= |V(G)|)$, the running time is $O(n^2\ log\ n)$.

---

5. (20 points) Consider a trucking company that runs trucks daily between New York and Boston. Trucks have a fixed limit W on the maximum amount of weight they are allowed to carry. Boxes arrive at the New York station one by one, and each package i has a weight w_i. The trucking station is quite small, so at most one truck can be at the station at any time. Company policy requires that boxes are shipped in the order they arrive. At the moment, the company is using a simple greedy algorithm for packing: they pack boxes in the order they arrive, and whenever the next box does not fit, they send the truck on its way.

Is the company using too many trucks? Maybe one could decrease the number of trucks needed by sometimes sending off a truck that was less full, and in this way allow the next few trucks to be better packed.

Prove that, for a given set of boxes with specified weights, the greedy algorithm currently in use actually minimizes the number of trucks that are needed. Your proof should follow the type of analysis we used for the Interval Scheduling Problem: it should establish the optimality of this greedy packing algorithm by identifying a measure under which it "stays ahead" of all other solutions.

Solution: Let n be the number of boxes that arrived so far and they arrived in the order $x_1, x_2, \ldots, x_n$. Let the greedy algorithm assign them to trucks $T_1, T_2, \ldots, T_N$ (in the order the truck leaves the station). So, we note that none of the trucks are overloaded (total weight in truck $\leq W$). Also, the boxes are sent in the order they arrive (i.e.,) if box x_i arrives before x_j $(i < j)$ and are assigned to trucks T_a and T_b respectively, then $a < b$. Now, we prove the following claim:

Claim: Let the greedy algorithm A pack the boxes $x_1, x_2, \ldots, x_i$ into the first N trucks and any arbitrary solution pack boxes $x_1, x_2, \ldots, x_j$ into the first N trucks, then $i \geq j$.

Proof: We prove this by induction on N:

- **BASE CASE ($N = 1$):** Since the greedy algorithm packs as many boxes into one truck, it is not possible for $i < j$. So, $i \geq j$ and the result is true for $N = 1$.
- **INDUCTIVE HYPOTHESIS:** Suppose that the result is true for some $N = k \geq 1$ trucks.
- **INDUCTIVE STEP:** Now, for the case of $N = n + 1$ trucks, let's say the greedy algorithm packs the boxes $x_1, x_2, \ldots, x_{i'}$ into the first n trucks and the remaining $x_{i'+1}, x_{i'+2}, \ldots, x_i$ $(i - i'$ boxes) into the last truck. Similarly the arbitrary solution packs $x_1, x_2, \ldots, x_{j'}$ into the first n trucks and the remaining $x_{j'+1}, x_{j'+2}, \ldots, x_j$ $(j - j'$ boxes) into the last truck. From the inductive hypothesis (for $N = n$), we have $i' \geq j'$.

$$i' \geq j' \Rightarrow -i' \leq -j' \Rightarrow j - i' \leq j - j' \tag{1}$$

  So, from the inequality in 1, we note that the greedy algorithm can fit at least $x_{i'+1}, x_{i'+2}, \ldots, x_j$ into the last truck making it j boxes overall in $(n + 1)$ trucks by the Greedy algorithm. Note that, the last truck will not be overloaded since the arbitrary solution is able to fit boxes $x_{j'+1}, x_{j'+2}, \ldots, x_j$ into the last truck without exceeding the capacity W. So, it follows that $i \geq j$ since we have shown that greedy algorithm fits at least j boxes into the $(n + 1)$ trucks.

- Thus the claim it true for all $N \geq 1$.

Hence, the greedy algorithm indeed minimizes the number of trucks used for the transportation.
