ESO207A: Data Structures and Algorithms	*July-Nov 2019*
Final Exam	*20th November, 2019*

Instructions.

a. Please answer all parts of a question together.

b. The exam is closed-book and closed-notes. Collaboration of any kind is ***not*** permitted. Cell phones on person are not allowed. Please use the mounted clock the examination hall for time reference.

c. You may cite and use algorithms and their complexity as done in the class.

d. Grading will be based not only on the correctness of the answer but also on the justification given and on the clarity of your arguments. Always argue correctness of your algorithms or conclusions.

Problem 1. *Short Answers* $(2.5 \times 6 = 15)$

Write down your recommendation for the data structure to use, that in each of the following cases, supports the given set of operations on a dynamic set S. Support your answer with time complexity reasons. Every member of S has a numeric (or alphanumeric) *key* attribute, in addition to perhaps other attributes. *ptr* is a pointer to an element of the data structure containing a member of S. Explain if probabilistic or deterministic guarantee requirements will change your choice?

(a) *Insert*(S, x), *DeleteMin*(S), *DecreaseKey*(x, key) and *FindMin*(S). ***Solution.***. This is the Min Priority Queue and can be implemented as a min-Heap. *Insert*, *DeleteMin* and *DecreaseKey* all take $O(\log n)$ time, where, $n = |S|$.

(b) *Insert*(S, x), *Delete*(S, ptr), *Search*(S, key).

Solution.. Usually, the best choice would be a hash table with a hash function drawn from a universal family of hash functions. Assuming that the hash function is computed in time $O(1)$, the time complexity *Insert* is $O(1)$, *Delete* is $O(1)$ (by maintaining doubly linked chains instead of singly linked ones), and *Search* takes expected time $O(1)$.

(c) *Insert*(S, x), *Delete*(S, ptr), *Search*(S, key), *Successor*(S, ptr).

Solution.. The best choice would be a balanced or nearly-balanced binary search tree structure, like red-black trees, where, all the above operations would be done in time $O(\log n)$, where, $n = |S|$.

For each of the following statements, either prove it or give a counterexample.

(d) Let e be any edge of minimum weight in G (there may be multiple edges with minimum weight). Then e must be part of *some* MST.

Solution.*Proof 1.* Let $e = \{a, b\}$ and consider the cut $(\{a\}, V - \{a\})$. Then, e is a lightest edge crossing this cut (since globally, it is a lightest edge). By the cut-property (let $X = \phi$) it is part of some MST.

Proof 2. Order the edges in the order of non-decreasing weight so that e is the first one. This is always possible. The first edge is always chosen by Kruskal's algorithm. So there is an MST, the one found by Kruskal, that contains e.

(e) Let e be any edge of minimum weight in G (there may be multiple edges with minimum weight). Then e must be part of *every* MST.

Solution.. This is false. See Figure 1.

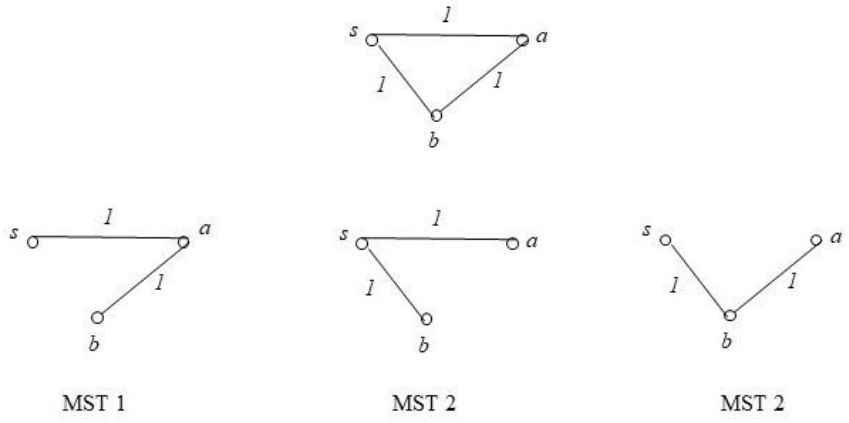


Figure 1: A (jointly) minimum weight edge in a graph may not be part of an MST.

(f) If G has a cycle with a unique lightest edge e, then e must be part of every MST.

Solution.. This is false. Two counterexamples are given. See Figure 2.

Problem 2. Give a linear-time $O(|V| + |E|)$ algorithm for the following task. (10)

Input: A directed acyclic graph G.
Question: Does G contain a directed path that touches every vertex exactly once.

Solution.Suppose $|V| = n$. Suppose G contains a directed path that touches every vertex exactly once if there is an edge from u_{i_j} to $u_{i_{j+1}}$, for each $j = 1, 2, \ldots n-1$. Then, $u_{i_1}, u_{i_2}, \ldots, u_{i_n}$ be a topological ordering of G and further, this topological ordering is unique. This can be computed as follows.

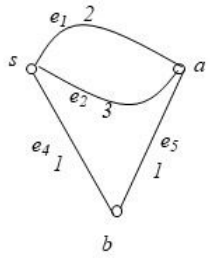
1. Compute a topological ordering $u_{i_1}, u_{i_2}, \ldots, u_{i_n}$ for G.
2. For each $j = 1, \ldots, n-1$, check if $u_{i_{j+1}} \in Adj[u_{i_j}]$.

Step 1 takes $O(|V| + |E|)$. Step 2 takes time $O(\sum_{u \in V} \deg(u)) = O(|E|)$. Total time is $O(|V| + |E|)$.

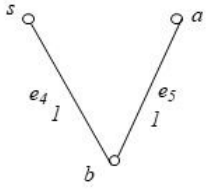
Problem 3. Give a linear time $O(|V| + |E|)$ algorithm that takes as input a directed acyclic graph $G = (V, E)$ and a fixed vertex s. For every vertex $t \in V$, output n_t the number of different directed paths from s to t. Note that $n_s = 0$. (10)

Solution.Topologically sort the graph. Initialize $n(s) = 1$ and $n(t) = 0, \forall t \in V - \{s\}$. Let v be a

(a)

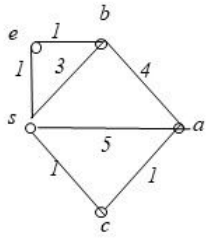


Graph G

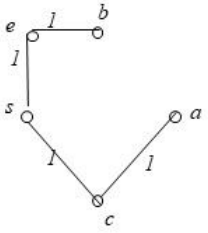


MST of weight 2

The cycle e_1 e_2 has unique smallest edge e_1. But e_1 is not part of any MST.



Graph G



MST of weight 4

The cycle s-a-b has the unique lightest edge s-b. But this graph has a unique MST of weight 4 that does not include this edge s-b.

Figure 2: A unique lightest edge of a cycle may not be part of *any* MST. Two examples are given. In Figure (a), the graph is not a simple graph, that is there are multiple edges between edges. This allows for a simpler example. In Figure (b), the graph is a simple graph.

vertex. Then, we have the recurrence relation

$$n(v) = \sum_{\substack{(w,v)\in E \\ w \neq s}} n(w) + \mathit{IsEdge}(s, w)$$

where,

$$\mathit{IsEdge}(s, w) = \begin{cases} 1 \text{ if } (s, w) \in E \\ 0 \qquad\qquad\qquad \text{otherwise.} \end{cases}$$

This can be computed by processing the vertices in increasing topological order. That way, by the time, v is reached, the set $\{n(w) \mid (w, v) \in E\}$ is computed.

1. Linearize the graph G
2. Compute G^R
3. $\forall v \in V$: $n[v] = 0$
4. **for** every vertex $v \in V$ in linearized order
5. **for** every $w \in Adj^R[v]$
6. **if** $w == s$
7. $\{ n[v] = n[v] + 1\}$
8. **else** $n[v] = n[v] + n[w]$

Steps 1 and 2 are each done in time $O(|V| + |E|)$. Step 3 is $O(|V|)$ time. Steps 4 through 8 take time $O(|V| + |E|)$.

Problem 4. Give a linear time algorithm to find an odd-length cycle in a strongly connected directed graph. (10)

Solution. The following property holds: A directed graph which is strongly connected has an odd-length directed cycle if and only if the graph is non-bipartite (when treated as an undirected graph).

The algorithm is as follows. Consider the directed graph G.

1. Choose any vertex r from V and do a BFS starting from r. As vertices are discovered for the first time, BFS labels them with a distance (level) from r, say $u.d$ for vertex u.
2. For each edge $(u, v) \in E$ compare $v.d$ and $u.d$.
   (a) If u and v have the same parity, that is, $u.d = v.d \mod 2$ then we will claim that there is an odd cycle.
   (b) To find the cycle, run DFS (BFS) from v to a to get a path.
   (c) If the length of the path $v \rightsquigarrow a$ has different parity than the $a \rightsquigarrow v$, then, $v \rightsquigarrow a \rightsquigarrow v$ is an odd cycle. Otherwise, $a \rightsquigarrow u \to v \rightsquigarrow a$ is an odd cycle (see argument and calculation below).

   Otherwise, that is, for all edges (u, v), u and v have different parity, report that the graph has no odd cycles. (This follows from the property stated at the beginning of this answer).

This computation BFS + a pass over all edges $(u, v) \in E$ take time $O(|E|) = O(|V| + |E|)$, since, $|V| = O(|E|)$ since the graph is a single strong component. An additional DFS from v takes $O(|V|+|E|)$ time.

Start a BFS from a vertex r. BFS assigns a level or distance to every vertex u denoted $u.d$, which is the shortest number of hops incurred by BFS starting from r to reach u. Suppose all edges $(u, v) \in E$ go across vertices with different parity, that is, $u.d \neq v.d$. Then, there is no odd cycle in the graph. Otherwise, there is an edge (u, v) and $u.d$ and $v.d$ have the same parity. The BFS tree has a path consisting of tree edges from r to u, and another path consisting of tree edges from r to v. Let a be the last vertex on the part of the path that is common to the BFS tree path from r to u and the BFS tree path from r to v. Let d_1 be the length of the BFS tree path P_1 from a to u and d_2 be the length of the BFS tree path P_2 from a to v. Note that d_1 and d_2 have the same parity, that is, $0 = (u.d - v.d) \mod 2 = (d_1 - d_2) \mod 2$, since the common path length in the BFS tree from r to a is subtracted from both $u.d$ and $v.d$.

Now since G is a strongly connected component, there is a simple path P_3 from v to a. Let P_3 have length d_3. If $d_3 \neq d_2 \mod 2$, then, $d_2 + d_3 = 1 \mod 2$, that is, $P_2 \cup P_3$ forms an odd cycle. If $d_3 = d_2 \mod 2$, then, $P_1 \cup (u, v) \cup P_3$ has length $d_2 + 1 + d_2 \mod 2 = 1 \mod 2$, i.e., it is an odd cycle.

Note that the cycle is not necessarily simple.

Problem 5. *Modular Arithmetic* Given a positive number $n > 1$ and numbers a, b and c, design an efficient algorithm that finds a solution x to the equation $ax + b = c \mod n$, if one exists and returns *NO SOLN* if no such x exists. Analyze its complexity. (10)

Solution.. *Note:* In some halls, the correction suggested was $ax = b \mod n$. In this case, in the following analysis, please consider $c - b$ as b–the rest of the analysis is the same.

Notation We use the notation $x|y$ to denote that x divides y. Note that there is a solution to $ax = e \mod n$ iff $\gcd(a, n)|e$. *Proof:* Suppose $\gcd(a, n) = g$ and $g|e$. Then, there exist integers x, y such that $ax' + ny' = g$ and $e = gd$. Multiplying $ax' + ny' = g$ both sides by d, we get the equation

$$adx' + ndy' = e$$

Taking $\mod n$ both sides, we get

$$ax = e \mod n$$

where, $x = dx'$. This also gives the algorithm for finding x.

Conversely, suppose $g \not| e$. Let $A = \{ax' + ny' \mid x', y' \in \mathbb{Z}\}$. As proved in the class, every member of A is a multiple of $\gcd(a, n)$ (the proof is repeated below). Since $g \not| e$, $e \notin A$. Hence, there is no solution of the form $ax' + ny' = e$. On the contrary, if there was a solution to the equation $ax = e \mod n$, then there is a y such that $ax + ny = e$ (i.e., $n|ax - e$), which would be a contradiction. Hence there is no solution to $ax = e \mod n$.

So, we can now design a simple algorithm for solving $ax + b = c \mod n$. Let $e = c - b$. Use Extended Euclid's algorithm to find x', y' and $g = \gcd(a, n)$ such that $ax' + ny' = g$. Now check if $g|e$. If so, let $e = dg$ and a solution for $ax = e \mod n$ is given by $x = d \cdot x' \mod n$. If $g \not| e$, then there is no solution, as proved above and the algorithm reports *NO SOLN*.

The complexity is one call to Extended-Euclid algorithm (cubic in bit length representation $O((\max(\log a, \log n))^3)$) followed by a constant number of checks of the form whether $g|e$ etc., which can be done in quadratic time (i.e., $O(\log |g| + \log |e|)$, etc..

Problem 6. *Counting Significant Inversions.* (10)
Let $A[1, \ldots, n]$ be an array of n distinct integers. The number of *significant inversions* in A is the number of pairs (i, j) such that $1 \leq i < j \leq n$ and $A[i] > 2 \cdot A[j]$. For example, the number of inversions in the array $\boxed{8\ 5\ 2\ 1}$ is 4 corresponding to the "significant out-of-order" numbers $(8, 2)$, $(8, 1)$, $(5, 2)$ and $(5, 1)$. Given an $O(n \log n)$ time algorithm for counting the number of significant inversions of a given array A. (*Note*: A $\Theta(n^2)$ algorithm will get only 2 marks.)

Solution Outline. This is a direct modification of the counting inversions problem. As before, given $A[1 \ldots n]$ we will sort A in place and return a count of significant inversions. This is done by overloading Merge-Sort routine, particularly, the Merge routine. Consider the routine $Merge(A, p, q, r)$. This assumes that $A[p \ldots q]$ is sorted and $A[q+1, \ldots, r]$ is sorted. The routine not only merges $A[p \ldots r]$ into a sorted array, but also counts the number of significant cross inversions, namely, $|\{(i, j) \mid p \leq i \leq q \text{ and } q+1 \leq j \leq r \text{ and } A[i] \geq 2A[j]\}|$ The only point to note is the following. As we are merging the lists, we keep an index i of the left list and j of the right list, as usual, where, $A[p \ldots, i-1] \cup A[q+1 \ldots j-1]$ have been copied into the merged array. i is the smallest index of $A[p \ldots q]$ not yet merged and j is the smallest index of $A[q+1 \ldots r]$ not yet merged. We also keep an index k, $i \leq k \leq q+1$ and maintain the following invariant: k is the smallest index satisfying $A[k] > 2A[j]$.

Due to the sorted nature of the lists $A[p \ldots q]$ and $A[q+1 \ldots r]$, if $A[k] > 2A[j]$, then, $A[k+1] > 2A[j]$ and so on. So when $A[j]$ is copied into its proper position in the merged array, that is, $A[j] < A[i]$, it should contribute $q - k + 1$ to the count of significant cross inversions, which is added to a running counter.

The invariant is easily maintained: initially, a scan is done starting at p through q to find the earliest index k such that $A[k] > 2A[q+1]$. If in any iteration, if $A[j]$ has been copied into the merged array, then, j is incremented by 1. So for the invariant to hold again, from the current value of k ($i \leq k \leq q+1$), we may have to advance k as follows:

```
// j has been incremented by 1
if  j > r
    return; // Merge is over, no new cross inversions.
else
    while  k ≤ q and A[k] < 2A[j]
        k = k + 1
```

The sum of the complexity of advancing k for $j = q+1 \ldots r$ is $O(q - p + 1 + r - (q+1) + 1) = O(r - p + 1)$, since, k only advances forwardfrom the previous position. Thus, the Merge function has cost $O(r - p + 1)$, which is linear. This gives the Merge-Sort complexity as $T(n) = 2T(n/2) + O(n)$, or that $T(n) = O(n \log n)$.

The number of inversions in the left list and the number of inversions in the right list are recursively computed while calling Merge-Sort on the left and right lists.

Problem 7. *Maximum contiguous subarray sum.* (10)
You are given an array $A[1 \ldots n]$ consisting of positive and/or negative numbers. Given indices $1 \leq i \leq j \leq n$, the contiguous sub-array sum $A[i \ldots j]$ is defined as $\sum_{k=i}^{j} A[k]$. Design an $O(n)$ dynamic programming algorithm that computes the maximum contiguous sub-array sum and also returns a contiguous sub-array with this maximum sum. (Explain your recurrence relation. *Note*: An $O(n \log n)$ time divide and conquer algorithm that solves the problem completely will get only 50% credit. A $\Theta(n^2)$ algorithm will get only 2 marks.)

Solution. For $1 \leq j \leq n$, let $L[j]$ denote the maximum contiguous sub-array sum ending at index j. That is, $L[j] = \max_{1 \leq k \leq j} \sum_{k=1}^{j} A[k]$. The index position $S[j]$ denotes an index k such that the contiguous sub-array sum of $A[k \ldots j]$ is the maximum and equals $L[j]$. $S[j]$ is the start of the maximum contiguous sub-array sum ending in j. We have to derive the recurrence relation for $L[j]$. Suppose we have computed $L[1 \ldots j]$ and have to compute $L[j+1]$. Let $A[k \ldots j+1]$ be the maximum contiguous subarray sum ending in $j+1$. Either $k = j+1$ or $k < j+1$. If $k = j+1$, a new segment is being started and its sum is $A[j+1]$. If $k < j+1$, then, the contiguous subarray is $A[k \ldots j+1]$ whose sum is $A[j+1] + \sum_{i=k}^{j} A[i]$. The latter part $\sum_{i=k}^{j} A[i]$ must be the maximum contiguous subarray sum ending in j, otherwise, this sum and the sum of $A[k \ldots j+1]$ can be improved by starting from the index $S[j]$. Hence, we have the recurrence relation

$$L[j+1] = \max(L[j] + A[j+1], A[j+1])$$

To find the maximum contiguous subarray sum, we take the maximum of the $L[j]$, $j = 1 \ldots n$. The program is as follows.

```
MaxContigSubArraySum(A, n)
1.   L[1] = A[1]
2.   S[1] = 1
3.   for j = 2 to n
4.        L[j] = max(L[j − 1] + A[j], A[j + 1])
5.        if  L[j − 1] + A[j] ≥ A[j + 1]
6.             S[j] = S[j − 1]
7.        else
8.             S[j] = j
// now find the maximum L[j]
9.   maxindex = 1
10.  maxval = L[1]
11.  for  j = 2 to n
12.       if  L[j] > maxval
13.            maxval = L[j]
14.            maxindex = j
15.  return (S[maxindex], maxindex, L[maxindex])
```

Problem 8. *Sequence of Valid Words.* (15)

You are given a string of n characters $s[1 \ldots n]$ which is believed to be a corrupted text document in which all punctuation has vanished (so that it looks like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function `dict`$(\cdot)$, that for any string w,

$$\texttt{dict}(w) = \begin{cases} \texttt{true} & \text{if } w \text{ is a valid word} \\ \texttt{false} & \text{otherwise.} \end{cases}$$

(a) Give a dynamic programming algorithm that determines whether the string $s[1 \ldots n]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to `dict` take unit time.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

Solution. We will first solve part (a). Initialize $V[0] = 1$. For $1 \leq j \leq n$, let $V[j]$ be 1 if the prefix string $s[1 \ldots j]$ can be constituted as a sequence of valid words, and $V[j]$ is 0 otherwise. For $1 \leq j \leq n$,

$$V[j] = \begin{cases} 1 & \exists k, 0 \leq k \leq j-1 \text{ s.t. } V[k] = 1 \text{ and } \texttt{dict}(s[k+1 \ldots j]) = \texttt{true} \\ 0 & \text{otherwise.} \end{cases}$$

To also help solve part (b), compute an array $P[1 \ldots n]$, (P for Partition indices). In the first case of the recurrence relation, $P[j]$ is set equal to k and otherwise, $P[i]$ is set to -1. We write the loop below. Assume `true` is 1 and `false` is 0.

ValidPartition(s, n)
1. $V[0] = 1$ //empty string is valid
2. $P[0] = -1$ // no partition for empty string
3. **for** $j = 1$ to n
4. $\quad$ *isvalid* = `false`
5. $\quad$ $k = j - 1$
6. $\quad$ **while** `not` *isvalid* and $k \geq 0$
7. $\quad\quad$ **if** $V[k] == 1$ and `dict`$(s[k+1 \ldots j]) ==$ `true`
8. $\quad\quad\quad$ $V[j] = 1$
9. $\quad\quad\quad$ $P[j] = k$
10. $\quad\quad\quad$ *isvalid* = `true`
11. $\quad\quad$ **else** $k = k - 1$

The running time for the inner while loop is in the worst case $O(j)$, and the outer loop runs from $j = 1$ to n, giving a total time of $O(n^2)$.

We can use the $P[1 \ldots n]$ array to output the reconstituted sequence. Let $Q[1 \ldots r]$ a backwards index array, where, r is the number of words in the reconstituted sequence. Q is the sequence constructed from P going backwards.

ConstructSequence(s, n)
1. **if** $V[n] == 0$ **return** "Sequence cannot be reconstituted"
2. **else**
3. $\quad$ create array $Q[1 \ldots n]$
4. $\quad$ $k = n$
5. $\quad$ $Q[1] = n$ // $Q[1]$ is initialized to n. Now going backwards.
6. $\quad$ $p = 2$ // p is the current partition number
7. $\quad$ **while** $k \geq 1$
8. $\quad\quad$ $Q[p] = P[k]$
9. $\quad\quad$ $p = p + 1$
10. $\quad\quad$ $k = P[k]$
11. $\quad\quad$ **if** $k == -1$ **return** "Error" // should not occur
12. $\quad$ $Q[p] = 0$
13. $\quad$ **for** $q = p$ *downto* 2

14. print $s[Q[p]+1 \ldots Q[p+1]]$
15. print a blank character

Problem 9. *A light spanning tree with special properties.* (15)
You are given as input an undirected graph $G = (V, E)$, edge weights w_e and a subset of vertices $U \subset V$. *Output*: The lightest spanning tree in which the nodes of U are leaves (there might be other leaves in this tree as well). Give an algorithm for this problem that runs in time $O(|E| \log |V|)$. Note that the answer isn't necessarily a minimum spanning tree.

Solution.Consider the lightest spanning tree T satisfying the properties given. Since all nodes of U are leaves, suppose we delete these nodes in U from T and the incident edge from $u \in U$ to its parent in T. Consider the depleted sub-tree T'. To satisfy the lightest spanning tree property, T' must be the MST of the subgraph G' obtained as a sub-graph of$G$ induced by removing all vertices U and its adjacent edges from G.

Pre-process the graph so that we obtain G' to be the sub-graph of G consisting of the vertex set $V - U$ and the edges of G that connect vertices of $V - U$. This can be done in time $O(|V| + |E|)$. Consider the set E' of edges of the form $\{u, w\}$, such that one end $u \in U$ and the other end is in $V - U$.

After the pre-processing, the algorithm is the following.

1. First find the MST T' of G' using Prim's or Kruskal's algorithm in time $O(|E| \log |V|)$.
2. Run a Prim's like algorithm by starting with T' and adding edges of E' one at a time, by selecting the minimum edge and adding it to T'. This is detailed below.

Construct the graph $G_1 = (V, E)$, where, the only edges are from vertices in U to vertices in $V - U$. This can be computed in time $O(|V| + |E|)$. Let the Adjacency List of G_1 be denoted as Adj_{G_1}.

1. **for** every vertex $u \in U$
2. **if** $Adj_{G_1}[u] = \phi$
3. **return** "No lighest spanning tree exists as per requirement" 4. **else**
5. Let $v \in Adj_{G_1}$ be the vertex adjacent to u with the minimum weight.
6. Add $\{u, v\}$ to T'.

This latter runs in $O(|E| + |U|)$ times. Overall, the algorithm runs in time $O(|E| \log |V|)$ time.