Algorithm 2025 Spring reference answers

1.

LCS = `001011`

2.

We assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search in T, plus 1. Then the expected cost of a search in T is $E[C_T] = \sum_{i=1}^{k}(l_i + 1) \cdot p_i = \sum_{i=1}^{k} p_i + \sum_{i=1}^{k} l_i \cdot p_i = 1 + \sum_{i=1}^{k} l_i \cdot p_i$

And the recurrence of OBST:

- $e(i,j) = \begin{cases} p_i, & i = j \\ \min\limits_{1 \le k \le n} \{e(i, k-1) + e(k+1, j) + w(i,j)\}, & i < j \end{cases}$

- $w(i,j) = \sum_{k=i}^{j} p_k$

(a) (answer 6%, explain 4%)

- In an optimal binary search tree on the ordered keys $a_1, a_2, \ldots, a_n$, we can denote it by $\mathrm{OBST}(1, n)$

- If we decide that a_k is a root, then the entire tree decomposes into two independent subtrees:

  1. Left subtree
     - Contains the keys $a_1, a_2, \ldots, a_{k-1}$.
     - It must be the OBST for those keys
     - We can denote it by $\mathrm{OBST}(1, k-1)$
  2. Right subtree
     - Similarly, we can denote it by $\mathrm{OBST}(k+1, n)$

- Graphically, the structure is:

```
            a_k
          /     \
        /         \
OBST(1,k-1)    OBST(k+1,n)
```

(b) (answer 6%, explain 4%)

Suppose in the OBST we have chosen a_k as the root, then we can denote

- $W_L = \sum_{a_i \in L} p_i = \sum_{i=1}^{k-1} p_i$
- $W_R = \sum_{a_i \in R} p_i = \sum_{i=k+1}^{n} p_i$

Let p_k be the weight of the root key a_k. Then when we splice L and R under a_k, every node in L and R moves "**one level deeper**", so each of them incurs an extra p_i in the total cost. Meanwhile a_k sits at the depth 1 and contributes p_k. Hence

- $C_T = (C_L + W_L) + p_k + (C_R + W_R)$

But note that

- $W_T = W_L + p_k + W_R$

so we can also write

- $C_T = C_L + C_R + W_T$

According to the recurrence of OBST, We can say

- $C_{OBST} = e(1, n)$
- $e(1, k-1)$ plays the role of C_L
- $e(k+1, n)$ plays the role of C_R
- $w(1, n)$ is exactly W_T

(C) (answer 6%, explain 4%)

When all weights are equally likely, $p_i = \frac{1}{n}$, the total weight of the tree on n keys is

- $W_T = \sum_{i=1}^{n} p_i = n \cdot \frac{1}{n} = 1$

Recall that if you choose a_k as the root, then

- $C_T = C_L + C_R + W_T = C_L + C_R + 1$

Since all the weights are the same, we can define

- $C_{OBST} = C(n) = \begin{cases} 0, & n = 0 \\ \min\limits_{1 \le k \le n} \{C(k-1) + C(n-k) + 1\}, & n > 0 \end{cases}$

Because the optimal split is always as balanced as possible when all p_i are the same, so we get exactly the recurrence:

- $C(n) = \begin{cases} 0, & n = 0 \\ C(\lfloor \frac{n-1}{2} \rfloor) + C(\lceil \frac{n-1}{2} \rceil) + 1, & n > 0 \end{cases}$

After solving the recurrence,

- $C(n) = \Theta(\log n)$

3.

Yes, if we sort the nodes with $O(n)$ sorting algorithms, we can build the balance tree with a $T(n) = 2T(n/2) + O(1) = O(n)$ algorithm:

```
build_tree(sorted_list):
    N = sorted_list.size()

    if N == 1:
        root.val = sorted_list[0]
        root.left = root.right = null
        return root
    if N <= 0:
        return null

    idx = floor(N/2)
    root.val = sorted_list[idx]
    root.left = build_tree(sorted_list[:idx])
    root.right = build_tree(sorted_list[idx+1:])
    return root
```

Or even better, trivial $O(n)$ algorithm to build a crooked tree:

```
build_tree(sorted_list):
    if sorted_list.size() == 0:
        return null
    root.val = sorted_list[0]
    root.left = null
    root.right = build_tree(sorted_list[1:])
    return root
```

4.

Algorithm Steps (6%)

1. Decrease x to $-\infty$
   - Set key[x] to a value smaller than every other key.
   - Then "bubble" x up to the root list.
2. Extract the minimum.
   - Now x is the minimum element in H, and it removes x from the root list.
   - Take x's children and merges them back into H.

Complexity Analysis (4%)

Let n be the size of the heap before deletion.

1. Decrease-key to $-\infty$:
   - A binomial tree of order k has height k (property 2).
   - There are at most $\lfloor \log_2 n \rfloor$ orders, i.e. $k = O(\log n)$, in the heap.
   - So the cost is $O(\log n)$.

2. Extract-min:
   Since there are at most $O(\log n)$ roots the root list, in this operation we need:
   - Scanning the root list of H.
   - Splice out and reversing the list of x' children, $O(\log k)$.
   - Merging two root-lists (original roots and children).
   - And the "linking" steps are constant time.
   - So the total cost is $O(\log n)$

5.

(a)

```
GENERIC-MST(G, w)
A = ∅
while A does not form a spanning tree
    find an edge (u, v) that is safe for A
    A = A ∪ {(u, v)}
return A
```

Theorem 23.1
Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let $(S, V - S)$ be any cut of G that respects A, and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A.

Proof Let T be a minimum spanning tree that includes A, and assume that T does not contain the light edge (u, v), since if it does, we are done. We shall construct another minimum spanning tree T' that includes $A \cup \{(u, v)\}$ by using a cut-and-paste technique, thereby showing that (u, v) is a safe edge for A.

The edge (u, v) forms a cycle with the edges on the simple path p from u to v in T, as Figure 23.3 illustrates. Since u and v are on opposite sides of the cut $(S, V - S)$, at least one edge in T lies on the simple path p and also crosses the cut. Let (x, y) be any such edge. The edge (x, y) is not in A, because the cut respects A. Since (x, y) is on the unique simple path from u to v in T, removing (x, y) breaks T into two components. Adding (u, v) reconnects them to form a new spanning tree $T' = T - \{(x, y)\} \cup \{(u, v)\}$.

We next show that T' is a minimum spanning tree. Since (u, v) is a light edge crossing $(S, V - S)$ and (x, y) also crosses this cut, $w(u, v) \le w(x, y)$. Therefore,

$$
\begin{aligned}
w(T') &= w(T) - w(x, y) + w(u, v) \\
&\le w(T) .
\end{aligned}
$$

(b)

```
function Kruskal(Graph G) is
    F:= ∅
    for each v in G.Vertices do
        MAKE-SET(v)

    for each {u, v} in G.Edges ordered by increasing weight({u, v}) do
        if FIND-SET(u) ≠ FIND-SET(v) then
            F := F ∪ { {u, v} }
            UNION(FIND-SET(u), FIND-SET(v))
    return F
```

Find minimum [edit]

To find the **minimum** element of the heap, find the minimum among the roots of the binomial trees. This can be done in $O(\log n)$ time, as there are just $O(\log n)$ tree roots to examine.[1]

By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to $O(1)$. The pointer must be updated when performing any operation other than finding the minimum. This can be done in $O(\log n)$ time per update, without raising the overall asymptotic running time of any operation.[citation needed]

Delete minimum [edit]

To **delete the minimum element** from the heap, first find this element, remove it from the root of its binomial tree, and obtain a list of its child subtrees (which are each themselves binomial trees, of distinct orders). Transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap. Since each root has at most $\log_2 n$ children, creating this new heap takes time $O(\log n)$. Merging heaps takes time $O(\log n)$, so the entire delete minimum operation takes time $O(\log n)$.[1]

```
function deleteMin(heap)
    min = heap.trees().first()
    for each current in heap.trees()
        if current.root < min.root then min = current
    for each tree in min.subTrees()
        tmp.addTree(tree)
    heap.removeTree(min)
    merge(heap, tmp)
```

- Time Complexity: $O(ElogE)$

(c)

```
function Prim(vertices, edges) is
    for each vertex in vertices do
        cheapestCost[vertex] ← ∞
        cheapestEdge[vertex] ← null

    explored ← empty set
    unexplored ← set containing all vertices

    startVertex ← any element of vertices
    cheapestCost[startVertex] ← 0

    while unexplored is not empty do
        // Select vertex in unexplored with minimum cost
        currentVertex ← vertex in unexplored with minimum cheapestCost[vertex]
        unexplored.remove(currentVertex)
        explored.add(currentVertex)

        for each edge (currentVertex, neighbor) in edges do
            if neighbor in unexplored and weight(currentVertex, neighbor) < ch
                cheapestCost[neighbor] ← weight(currentVertex, neighbor)
                cheapestEdge[neighbor] ← (currentVertex, neighbor)

    resultEdges ← empty list
    for each vertex in vertices do
        if cheapestEdge[vertex] ≠ null THEN
            resultEdges.append(cheapestEdge[vertex])

    return resultEdges
```

- Maintain cheapestCost with binomial heap

  Decrease key [edit]

  After **decreasing** the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated. Each binomial tree has height at most $\log_2 n$, so this takes $O(\log n)$ time.[1] However, this operation requires that the representation of the tree include pointers from each node to its parent in the tree, somewhat complicating the implementation of other operations.[3]

- Time Complexity: $O(VlogV) + O(ElogV) = O(ElogV)$

6.

Algorithm setup

- We maintain a parent pointer $p[x]$ for each node x.
  - If $p[x] = x$, x is a root.

    - Otherwise, $p[x]$ points towards its parent in the tree.
- For each root r we keep $size[r]$ = the number of nodes in r's tree.
- To union two trees rooted at r1, r2.
    1. Compare sizes: assume $size[r_1] \le size[r_2]$.
    2. Make r_1's parent point to r_2: $p[r_1] \leftarrow r_2$
    3. Update $size[r_2] \leftarrow size[r_1] + size[r_2]$

Claim: Total cost of n–1 unions is $O(n)$ (3%)

- Starting from n singleton trees, you need exactly n–1 unions to merge them all into one tree. Since each union is just one constant-time pointer assignment ($p[r_1] \leftarrow r_2$), the total work is $(n-1) \times O(1) = O(n)$

Claim: Height bound: $h \le \lfloor \log n \rfloor$ (7%)

- https://hackmd.io/@KentLee/Syshq-trJx (https://hackmd.io/@KentLee/Syshq-trJx)

7.

When k is a constant, the problem is easy; otherwise, it is difficult since if it's easy, we can make $k = |V|$ can solve the Hamiltonian Cycle problem in polynomial time.
Assuming $P \neq NP$, this is contradictory.
(Hamiltonian Cycle problem can be reduced to k cycle problem)

8.

Yes. (2%)

Explaination

Because if a directed graph has no cycles, we can use dynamic programming (e.g., Bellman-Ford style) along with the max-version of the RELAX operation to find the maximum distance from s to t.

Alternatively, we can also use a greedy approach based on the **topological ordering** of the vertices to find our solution.

Example Algorithm (8%)

1. Topological sort the vertices of G.

2. Initialize
   - $dist[v] := \begin{cases} 0, & v = s \\ -\infty, & v \neq s \end{cases}$
   - $pred[v] := \text{NIL}$

3. Relax edges in topo-order:

```
for each u in topological order:
    for each v in outgoing(u):
        if dist[v] < dist[u] + w(u,v):
            dist[v] = dist[u] + w(u,v)
            pred[v] = u
```

4. Outputs:
   - Longest-path length = dist[t]
     - Setting it to infinity if t is unreachable.
   - Longest-path:
     - follow $pred$ backward from t to s and then reverse (if t is reachable from s).