1. **Initialization.**

   *Don't forget to do this.*

2. **Empirical running time.**

   16,000, \( \Theta(V^2E) \)

   *When \( V \) doubles, the running time goes up by a factor of 4, so the exponent for \( V \) is \( \log_2 4 = 2 \). When \( E \) quadruples, the running time goes up by a factor of 4, so the exponent for \( E \) is \( \log_4 4 = 1 \). Thus, the order of growth of the running time is \( \Theta(V^2E) \).*

3. **Depth-first search.**

   (a) 0 2 5 8 1 3 7 9 6 4
   
   (b) 5 1 7 9 3 6 8 2 4 0
   
   (c) yes

   *The digraph is a DAG. So, the reverse postorder provides a topological order.*

4. **Minimum spanning trees.**

   (a) 0 10 20 30 50 60 110
   
   (b) 30 0 20 50 60 10 110

5. **Shortest paths.**

   (a) 0 7 58 13 3 1
   
   (b) 0 4 5
6. Maxflows and mincuts.

(a) \[ 31 = 8 + 5 + 18 \]

(b) \[ 34 = 13 + 21 \]

(c) 31

The net flow across any cut is equal to the value of the flow.

(d) \[ A \rightarrow F \rightarrow B \rightarrow G \rightarrow H \]

(e) 3

The edge \( B \rightarrow G \) is the bottleneck.

7. Data structures.

(a) \( F \rightarrow C \) would not be inserted at index 3 with index 0 empty.

T The second one would arise if the keys were inserted in the order A B C D E.

T The third one would arise if the keys were inserted in the order B A E D C.

(b) \( (10, 10), (12, 9) \)

The constraints of the 2d-tree imply that, for any point \((x, y)\) in \(T\), we must have both \(9 \leq x < 13\) and \(8 \leq y < 14\).

(c) \( \Theta(n^2), \Theta(n) \)

In the worst case (repeatedly removing the first element), each call to \texttt{remove()} takes time proportional to number of elements remaining. This leads to a running time of \(n + (n - 1) + \ldots + 1\), which is \(\Theta(n^2)\).

In the best case (repeatedly removing the last element), each call to \texttt{remove()} takes \(\Theta(1)\) time. Also, each call to \texttt{append()} takes \(\Theta(1)\) amortized time. So, the overall running time is \(\Theta(n)\).
8. Dynamic programming.

    A C E H H L or C A E H H L

```java
int[][] opt = new int[m+1][n+1];
for (int i = 1; i <= m; i++) {
    for (int j = 1; j <= n; j++) {
        if (times[j] > i) {
            opt[i][j] = opt[i][j-1];
        } else {
            opt[i][j] = Math.max(opt[i][j-1], points[j] + opt[i - times[j]][j-1]);
        }
    }
}
```


(a) A B D

It runs Kruskal’s algorithm (using the random edge weights), adding the edges C—E, E—F, A—D, and A—B to T, until T contains exactly two connected components.

(b) 4

The edges that cross the cut are A—E, B—C, B—E, and D—E.

10. Multiplicative weights.

    T F F F F F

11. Intractability.

    N Y Y Y Y Y Y N
12. **Princeton path game.**

(a) To determine whether the orange player has already won:
   - Build an edge-weighted graph $G'$ containing only the orange edges.
   - Run BFS (or DFS) to determine whether there is a directed path from $s$ to $t$ in $G'$.
   - If such a path exists, declare orange the winner.

(b) To determine whether the black player has already won:
   - Build an edge-weighted graph $G''$ containing the orange and uncolored edges (but not the black edges).
   - Run BFS (or DFS) to determine whether there is a directed path from $s$ to $t$ in $G''$.
   - If no such path exists, declare black the winner.

(c) The game cannot end in a tie.
   Let's suppose the game continues until all edges are colored either orange or black. We'll see that exactly one player must win.
   - If there is a directed path $P$ from $s$ to $t$ containing only orange edges, then orange wins (and black cannot simultaneously win because there are no black edges in $P$).
   - Otherwise, consider the subset of vertices $S$ reachable from $s$ via orange edges, and let $T$ be the remaining vertices. Note that $s \in S$ and $t \in T$. All edges that go from $S$ to $T$ are black and every directed path from $s$ to $t$ must use one (or more) of these edges. Thus, black wins.
13. Princeton minimum spanning trees.

The main idea is to change the weight of all of the orange edges to a small value, smaller than the weight of any of the black edges. That way, the MST will prefer the orange edges to the black edges.

Step 1. Construct $G'$:

- Create an edge-weighted graph $G'$ that has the same vertices and edges as $G$.
- Let $w(e)$ and $w'(e)$ denote the weight of edge $e$ in $G$ and $G'$, respectively.
- If edge $e$ is black, set $w'(e) = w(e)$.
- If edge $e$ is orange, set $w'(e) = \min_e w(e) - 1$.

Step 2. Compute the MST $T'$ of $G'$ via Prim or Kruskal.

- If $T'$ contains all of the orange edges, then return $T'$.
- Otherwise, report no Princeton-MST exists.

Alternate solution (to determine whether Princeton MST exists).

- Create a graph $G''$ containing all of the orange edges in $G$.
- Determine whether $G''$ contains a cycle using DFS.
- If $G''$ contains a cycle, then report no Princeton-MST exists.

Alternate solution (to find MST). Create a graph $G'$ formed by contracting all of the orange edges in $G$; compute the MST in $G'$; and return the corresponding edges in $G$. Some care is needed to contract the edges efficiently, which we won’t describe here.