Final

Note

This is a closed-book exam. Each problem accounts for 10 points, unless otherwise marked.

Problems

1. Consider the Knapsack (Subset Sum) Problem: Given a set S of n items, where the i-th item has an integer size $S[i]$, and an integer K, find a subset of the items whose sizes sum to exactly K or determine that no such subset exists.

   We have discussed in class two approaches to implementing a solution that we designed by induction: one uses dynamic programming (see the Appendix), while the other uses recursive function calls.

   Suppose there are 5 items, with sizes $2, 3, 4, 6, 7$, and we are looking for a subset whose sizes sum to 14. Assuming recursive function calls are used, please give the two-dimension table P whose entries are filled with -, O, I, or left blank when the algorithm terminates. Which entries of $P[n, K]$ are visited/computed more than once? Please mark those entries in the table.

2. Consider the *next* table as in the KMP algorithm for string $B[1..9] = abaababaa$.

1	2	3	4	5	6	7	8	9
a	b	a	a	b	a	b	a	a
-1	0	0	1	1	2	3	2	3

   Suppose that, during an execution of the KMP algorithm, $B[6]$ (which is an a) is being compared with a letter in A, say $A[i]$, which is not an a and so the matching fails. The algorithm will next try to compare $B[next[6] + 1]$, i.e., $B[3]$ which is also an a, with $A[i]$. The matching is bound to fail for the same reason. This comparison could have been avoided, as we know from B itself that $B[6]$ equals $B[3]$ and, if $B[6]$ does not match $A[i]$, then $B[3]$ certainly will not either. $B[5]$, $B[8]$, and $B[9]$ all have the same problem, but $B[7]$ does not. Please adapt the computation of the *next* table, reproduced below, so that such wasted comparisons can be avoided.

```
Algorithm Compute_Next ($B, m$);
begin
    $next[1] := -1$;  $next[2] := 0$;
    for $i := 3$ to $m$ do
        $j := next[i-1] + 1$;
        while $B[i-1] \neq B[j]$ and $j > 0$ do
            $j := next[j] + 1$;
        $next[i] := j$
end
```

3. Given as input a connected undirected graph G, a spanning tree T of G, and a vertex v, design an algorithm to determine whether T is a valid DFS tree of G rooted at v. In other words, determine whether T can be the output of DFS under some order of the edges starting with v. Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary. Explain why the algorithm is correct and give an analysis of its time complexity. The more efficient your algorithm is, the more points you get for this problem.

4. Consider Dijkstra's algorithm for single-source shortest paths as shown below. You may find in the literature two bounds, namely $O(|V|^2)$ and $O((|E| + |V|) \log |V|)$, for its time complexity. Why is this so? What does this difference imply?

```
Algorithm Single_Source_Shortest_Paths($G, v$);
begin
    for all vertices $w$ do
        $w.mark := false$;
        $w.SP := \infty$;
    $v.SP := 0$;
    while there exists an unmarked vertex do
        let $w$ be an unmarked vertex s.t. $w.SP$ is minimal;
        $w.mark := true$;
        for all edges $(w, z)$ such that $z$ is unmarked do
            if $w.SP + length(w, z) < z.SP$ then
                $z.SP := w.SP + length(w, z)$
end
```

5. Let $G = (V, E)$ be a connected weighted undirected graph and T be a minimum-cost spanning tree (MCST) of G. Suppose that the cost of one edge $\{u, v\}$ in G is *increased*; $\{u, v\}$ may or may not belong to T. Design an algorithm either to find a new MCST or to determine that T is still an MCST. The more efficient your algorithm is, the more points you will be credited for this problem. Explain why your algorithm is correct and analyze its time complexity.

6. Finding a small vertex cover for an arbitrary undirected graph is difficult, but is much easier for trees; a *vertex cover* of a graph G is a set of vertices such that every edge in G is incident to at least one of these vertices. Design an efficient algorithm to find a minimum-size vertex cover for a given tree. Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary. The more efficient your algorithm is, the more points you will be credited for this problem. Explain why your algorithm is correct and give an analysis of its time complexity.

7. Below is a solution to the single-source shortest path problem using the dynamic programming approach, which we have discussed in class:

Denote by $D^l(u)$ the length of a shortest path from v (the source) to u containing *at most* l edges; particularly, $D^{n-1}(u)$ is the length of a shortest path from v to u (with no restrictions).

$$D^1(u) = \begin{cases} \mathit{length}(v,u) & \text{if } (v,u) \in E \\ 0 & \text{if } u = v \\ \infty & \text{otherwise} \end{cases}$$

$$\begin{aligned} D^l(u) = \; & \min\{D^{l-1}(u), \min_{(u',u)\in E}\{D^{l-1}(u') + \mathit{length}(u',u)\}\}, \\ & 2 \le l \le n-1 \end{aligned}$$

Please explain why the solution allows edges with a negative weight (as long as there is no cycle with a negative weight). How is this different from Dijkstra's algorithm? Please explain.

8. In the proof (discussed in class) of the NP-hardness of the 3SAT problem by reduction from the SAT problem, we convert an arbitrary boolean expression in CNF (input of the SAT problem) to a boolean expression in 3CNF (where each clause has exactly three literals).

   (a) Please illustrate the conversion by giving the boolean expression that will be obtained from the following boolean expression:

   $$(\overline{w} + x + y + \overline{z}) \cdot (v + w + x + \overline{y} + z) \cdot (\overline{v} + y).$$

   (b) The original boolean expression is satisfiable. As a demonstration of how the reduction works, please use the resulting boolean expression to show that it indeed the case.

9. To prove that "P = NP" (which seems unlikely though), it suffices to show that some NP-complete problem is in P. Why? Please explain.

10. The (standard) knapsack problem is as follows.

    Given a set X, where each element $x \in X$ has an associated size $s(x)$ and value $v(x)$, and two other numbers S and V, is there a subset $B \subseteq X$ whose total size is $\le S$ and whose total value is $\ge V$?

    Prove that the knapsack problem is NP-complete. (Hint: by reduction from the partition problem.)

Appendix

- Below is an algorithm for determining whether a solution to the Knapsack (Subset Sum) Problem exists.

```
**Algorithm Knapsack** $(S, K)$;
**begin**
    $P[0,0].exist := true$;
    **for** $k := 1$ **to** $K$ **do**
        $P[0,k].exist := false$;
    **for** $i := 1$ **to** $n$ **do**
        **for** $k := 0$ **to** $K$ **do**
            $P[i,k].exist := false$;
            **if** $P[i-1,k].exist$ **then**
                $P[i,k].exist := true$;
                $P[i,k].belong := false$
            **else if** $k - S[i] \geq 0$ **then**
                    **if** $P[i-1,k-S[i]].exist$ **then**
                        $P[i,k].exist := true$;
                        $P[i,k].belong := true$
**end**
```

- The partition problem: given a set X where each element $x \in X$ has an associated size $s(x)$, is it possible to partition the set into two subsets with exactly the same total size?

  The partition problem is NP-complete.