

Final

Note

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

Problems

1. Design an algorithm that, given a set of integers $S = \{x_1, x_2, \dots, x_n\}$, finds a nonempty subset $R \subseteq S$, such that

$$\sum_{x_i \in R} x_i \equiv 0 \pmod{n}.$$

Please present your algorithm in adequate pseudocode and make assumptions wherever necessary. Give also an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem.

Before presenting your algorithm, please argue why such a nonempty subset must exist. (Hint: think about the sums x_1 , $x_1 + x_2$, $x_1 + x_2 + x_3$, \dots , $x_1 + x_2 + \dots + x_{n-1}$, and $x_1 + x_2 + \dots + x_{n-1} + x_n$; the difference between any two of these is also a sum.)

2. Please give a binary de Bruijn sequence of 2^4 bits, which is a *cyclic* sequence of 16 bits $a_1 a_2 \dots a_{16}$ such that each binary sequence of size 4 appears somewhere in the sequence (wrapped around if necessary). Explain how you can systematically produce the de Bruijn sequence.
3. Let $G = (V, E)$ be a connected weighted undirected graph, where the costs of all edges are distinct, and $T = (V, E')$ be a spanning tree of G . Prove that, T is a minimum-cost spanning tree (MCST) of G *if and only if*, for every edge of G that is not in T , the edge must have the highest cost among the edges in the cycle created when it is inserted into T .
4. Below is the algorithm discussed in class for determining the strongly connected components (SCCs) of a directed graph using DFS. To explore the neighbors of a particular node v on which the SCC procedure is invoked, the algorithm visits each neighbor w of v via some edge (v, w) . Relative to the DFS tree (which is implicit in the algorithm), the edge (v, w) may be classified as a tree edge, forward edge, back edge, or cross edge. How does the algorithm handle these different cases? Please explain by referring to the pseudocode and pointing out the action (or no action) taken for each case.

Algorithm Strongly_Connected_Components(G, n);
begin

```

for every vertex  $v$  of  $G$  do
     $v.DFS\_Number := 0$ ;
     $v.component := 0$ ;
 $Current\_Component := 0$ ;  $DFS\_N := n$ ;
while  $v.DFS\_Number = 0$  for some  $v$  do
     $SCC(v)$ 
end

procedure  $SCC(v)$ ;
begin
     $v.DFS\_Number := DFS\_N$ ;
     $DFS\_N := DFS\_N - 1$ ;
    insert  $v$  into  $Stack$ ;
     $v.high := v.DFS\_Number$ ;
    for all edges  $(v, w)$  do
        if  $w.DFS\_Number = 0$  then
             $SCC(w)$ ;
             $v.high := \max(v.high, w.high)$ 
        else if  $w.DFS\_Number > v.DFS\_Number$ 
            and  $w.component = 0$  then
                 $v.high := \max(v.high, w.DFS\_Number)$ 
    if  $v.high = v.DFS\_Number$  then
         $Current\_Component := Current\_Component + 1$ ;
        repeat
            remove  $x$  from the top of  $Stack$ ;
             $x.component := Current\_Component$ 
        until  $x = v$ 
end

```

5. In the determination of SCCs by DFS, groups of vertices are discovered as SCCs one after another. The order of discovery depends on the vertex from which the search starts and the order via which the edges are considered from a vertex currently being visited. Those vertices getting 1 as their component number are considered discovered first, those getting 2 are second, and so on.

Draw three weakly connected directed graphs, each with four SCCs and as few vertices and edges as possible, such that one may produce *exactly* one, two ($= 2!$), and six ($= 3!$) different orders of discovery of the SCCs respectively for the three graphs. A directed graph is weakly connected if, when the orientations of all the directed edges are ignored, the graph is connected.

What if the directed graphs are not required to be weakly connected?

6. Below is an algorithm, based on the dynamic programming approach, for solving the single-source shortest paths problem.

Algorithm Single_Source_Shortest_Paths($length$);
begin

```

 $D[v] := 0;$ 
for all  $u \neq v$  do
    if  $(v, u) \in E$  then
         $D[u] := \text{length}(v, u)$ 
    else  $D[u] := \infty;$ 
for  $l := 2$  to  $n - 1$  do
    for all  $u \neq v$  do
        for all  $u'$  such  $(u', u) \in E$  do
            if  $D[u'] + \text{length}[u', u] < D[u]$  then
                 $D[u] := D[u'] + \text{length}[u', u]$ 
end

```

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).

In the for loop with index l iterating from 2 to $n - 1$, it is possible that, for certain $l = k$, $D[u]$ acquires the value of $D^{k'}(u)$, where $k < k'$. Why? Please explain with an example.

7. Consider designing by dynamic programming an algorithm that, given as input a sequence of distinct numbers, determines the length of a longest increasing subsequence in the input sequence. For instance, if the input sequence is 1, 3, 11, 5, 12, 14, 7, 9, 15, then a longest subsequence is 1, 3, 5, 7, 9, 15 whose length is 6 (another longest subsequence is 1, 3, 11, 12, 14, 15).
 - (a) Formulate the solution using recurrence relations.
 - (b) Present the algorithm in suitable pseudocode, based on the previous recursive formulation. What is the time complexity of your algorithm?
8. A string s may be *scrambled* into another string s' , if s may be changed into s' by recursively dividing and exchanging (or not exchanging) the order of two resulting substrings, or more precisely, by applying the following recursive scrambling procedure.
 - If the input string s is of length 1, then stop.
 - Divide the input s (which is of length at least 2) arbitrarily into two nonempty substrings x and y such that the concatenation of x and y (denoted by $x \cdot y$) equals s , scramble x into x' and y into y' (using this procedure), and then compose either $x' \cdot y'$ or $y' \cdot x'$.

So, a string may be scrambled into possibly many other strings, but only of the same length. Any string clearly may be scrambled into itself, by not exchanging the order of substrings. For a less trivial example, “abcd” may be scrambled into “dcab”, because we can first divide “abcd” as “ab”·“cd”, scramble “cd” into “dc” and concatenate “dc” with “ab” to obtain “dcab”. However, “abcd” may never be scrambled into “bdac”.

Design using dynamic programming an algorithm that, given two strings (arrays of symbols/characters) A and B of length n , determines whether A may be scrambled into B .

- (a) Formulate the solution using recurrence relations.

(Hint: let $S(i, j, k)$ denote the answer, a Boolean value, to the question of whether the substring of A of length k starting from index i may be scrambled into the substring of B of the same length starting from index j , i.e., whether $A[i..(i+k-1)]$, the substring of A indexed from i through $i+k-1$, may be scrambled into $B[j..(j+k-1)]$. Assuming the elements of an array are indexed from 1, $S(1, 1, n)$ then represents the final answer. You may use function \max or \vee over a set of Boolean values to represent their disjunction and \min or \wedge to represent conjunction.)

- (b) Present the algorithm in suitable pseudocode, based on the previous recursive formulation. What is the time complexity of your algorithm?

9. 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 (in 3CNF) that will be obtained from the following Boolean expression:

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

- (b) The original Boolean expression is satisfiable. As a demonstration of why the reduction is correct, please use the resulting Boolean expression (in 3CNF) to show that it is indeed the case.

10. Solve one of the following two problems about NP-completeness. (Note: if you try to solve both problems, I will randomly pick one of them to grade.)

- (a) The double satisfiability problem is as follows.

Given a Boolean expression in conjunctive normal form, determine whether it has two different satisfying assignments.

Prove that the double satisfiability problem is NP-complete. (Hint: reduction from SAT; you may introduce new Boolean variables and add new clauses in the input conversion.)

- (b) 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 $\leq S$ and whose total value is $\geq V$?

Prove that the knapsack problem is NP-complete.

Appendix

- The satisfiability (SAT) problem: given a Boolean expression in conjunctive normal form, determine whether it is satisfiable (i.e., it has a satisfying truth assignment to its Boolean variables).

The SAT problem is NP-complete.

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

The partition problem is NP-complete.