Suggested Solutions to Midterm Problems

1. Consider the following inductive definition for the set of all binary trees that store integer key values:

   - The empty tree, denoted $\perp$, is a binary tree, storing no key value.
   - If t_l and t_r are binary trees, then $node(k, t_l, t_r)$, where $k \in \mathbb{Z}$, is also a binary tree with the root storing key value k.

   Refine the definition to include only binary *search* trees where an inorder traversal of a binary search tree produces a list of all stored key values in *increasing* order.

   To make the definition mathematically precise, you must define suitable functions $maxVal$ and $minVal$ that return respectively the maximum key value and the minimum key value stored in a given binary tree (which may be empty). You may use ∞ ($-\infty$) to denote the value that is larger (smaller) than any other value.

   Solution. The set of all binary search trees that store integer key values may be defined as follows:

   (a) The empty tree, denoted $\perp$, is a binary search tree, storing no key value.

   (b) If t_l and t_r are binary search trees, then $node(k, t_l, t_r)$, where $k \in \mathbb{Z}$, $maxVal(t_l) \leq k$, and $k \leq minVal(t_r)$, is also a binary search tree with the root storing key value k.

   The maximum key value $maxVal(t)$ and the minimum key value $minVal(t)$ stored in a binary tree t are defined as follows:

   $$maxVal(t) = \begin{cases} -\infty & \text{if } t = \perp \\ \max(maxVal(t_l), k, maxVal(t_r)) & \text{if } t = node(k, t_l, t_r) \end{cases}$$

   constructor function

   $$minVal(t) = \begin{cases} \infty & \text{if } t = \perp \\ \min(minVal(t_l), k, minVal(t_r)) & \text{if } t = node(k, t_l, t_r) \end{cases}$$

   □

2. Let n be a natural number ($n \geq 0$) and p be a prime ($p \geq 2$). Let s be the sum of the p-ary digits in the representation of n in base p. Let m be the multiplicity of the factor p in $n!$, i.e., the maximum value m such that p^m divides $n!$. For example, if $n = 6$ and $p = 2$, then $n = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 110_2$ and hence $s = 1 + 1 + 0 = 2$. Moreover, 2^4 divides $6!$ but 2^5 does not and therefore $m = 4$.

   Prove that

   $$m = \frac{n - s}{p - 1} .$$

   In the example above, $\frac{n-s}{p-1} = \frac{6-2}{2-1} = 4 = m$.

   Solution. The proof is by induction on n.

   Base case ($n = 0$): When $n = 0$, we have $s = 0$, $n! = 1$, and $m = 0$ and hence $\frac{n-s}{p-1} = \frac{0-0}{p-1} = 0 = m$.

   Inductive step ($n > 0$): Let s and m be as defined in the problem statement for n. Let s' be the sum of the p-ary digits of $n - 1$ and m' be the multiplicity of the factor p in

$(n-1)!$. We then have $m-m'$ as the multiplicity of the factor p in n, i.e., $p^{m-m'}$ divides n but $p^{m-m'+1}$ does not, which implies that n, when represented in base p, has at least $(m-m'+1)$ p-ary digits and exactly $(m-m')$ trailing 0's. Let d_i be the i-th p-ary digit of n, counting from the least significant digit (which is the 0-th digit). When represented in base p, n is in the form of $\ldots d_{m-m'+1}d_{m-m'}0\ldots0$, where $d_{m-m'} \neq 0$ and therefore $n-1$ is in the form of $\ldots d_{m-m'+1}(d_{m-m'}-1)(p-1)(p-1)\ldots(p-1)$ with $m-m'$ trailing $(p-1)$'s.

So, $s'-s = (d_{m-m'}-1)-d_{m-m'}+(m-m')(p-1) = (m-m')(p-1)-1$ and hence $s'-s+1 = (m-m')(p-1)$, $m-m' = \frac{s'-s+1}{p-1}$, and $m = \frac{s'-s+1}{p-1}+m'$. From the induction hypothesis, we have $m' = \frac{(n-1)-s'}{p-1}$. It follows that $m = \frac{s'-s+1}{p-1}+\frac{(n-1)-s'}{p-1} = \frac{s'-s+1+(n-1)-s'}{p-1} = \frac{n-s}{p-1}$. □

3. In a homework problem, to determine whether $f(n) = O(g(n))$ and/or $f(n) = \Omega(g(n))$, for a given pair of monotonically growing functions f and g that map natural numbers to non-negative real numbers, you may have claimed and used the following:

   If $f(n) = o(g(n))$, then $f(n) = O(g(n))$ and $f(n) \neq \Omega(g(n))$.

   Prove that the claim is indeed true.

   Solution. When $f(n) = o(g(n))$, i.e., $\lim_{n\to\infty}\frac{f(n)}{g(n)} = 0$, it means that, for every real $\varepsilon > 0$, there exists a natural number N such that, for every natural number $n > N$, $|\frac{f(n)}{g(n)}-0| < \varepsilon$ or $f(n) < \varepsilon g(n)$ (since both $f(n)$ and $g(n)$ are non-negative). It follows that there exist a real constant c (by taking one of the ε values) and a natural constant N' (by taking $N+1$ where N is suited for the particular ε value taken) such that, for all $n \geq N'$, $f(n) \leq cg(n)$, and hence $f(n) = O(g(n))$.

   We next show that, given the same condition of $f(n) = o(g(n))$, $f(n) \neq \Omega(g(n))$, i.e., it is not the case that there exist constants c and N (both greater than 0) such that, for all $n \geq N$, $f(n) \geq cg(n)$, which is equivalent to the following statement: for every constant $c > 0$, we have "for every constant $N > 0$, there exists some $n \geq N$ and $f(n) < cg(n)$." The definition of $f(n) = o(g(n))$ implies that, for every constant $c > 0$, there exists a constant $N > 0$ such that, for every $n > N$, $f(n) < cg(n)$ and therefore, for every constant $c > 0$, we have "for every constant $N' > 0$, there exists some $n \geq N'$ and $f(n) < cg(n)$," which is the preceding statement that is equivalent to $f(n) \neq \Omega(g(n))$. □

4. The Knapsack Problem that we discussed in class is defined as follows: Given a set S of n items, where the ith 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 described in class an algorithm (see the Appendix) to solve the problem. Modify the algorithm to solve a variation of the Knapsack Problem where each item has an *unlimited* supply. In your algorithm, please change the type of $P[i,k].belong$ into integer and use it to record the number of copies of item i needed. Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will get for this problem.

   Solution.

   Algorithm Knapsack_Unlimited (S, K)**;**
   begin
   $P[0,0].exist := true$;

```
    P[0,0].belong := 0;
    for k := 1 to K do
        P[0,k].exist := false
    end for
    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 := 0
            else if k - S[i] ≥ 0 then
                    if P[i,k-S[i]].exist then
                       P[i,k].exist := true;
                       P[i,k].belong := P[i,k-S[i]].belong + 1
                    end if
                end if
            end if
        end for
    end for
end
```

From the main nested for-loops, we see that the complexity is $O(nK)$. (Note: the bound should be understood as $O(n2^{\log K})$, where $\log K$ represents the input size of the number K.) □

5. Suppose that you are given an algorithm/function called *subsetSum* as a *black box* (you cannot see how it is designed) that has the following properties: If you input any sequence X of real numbers and an integer k, *subsetSum*(X, k) will answer "yes" or "no", indicating whether there is a subset of the numbers whose sum is exactly k. Show how to use this black box to find the subset whose sum is k, if it exists. You should use the black box $O(n)$ times (where n is the size of the sequence).

   Solution. We assume that a sequence of n numbers are stored in an array of n elements, where the elements are indexed from 1 through n.

```
Algorithm Print_Subset(S,k);
begin
   if subsetSum(S,k)="no" then
      print "No suitable subset"; halt
   end if;
   print "Below is a suitable subset:";
   sum := 0.0;
   i := 1;
   while sum<k do
      this := S[i];
      S[i] := 0.0; // Remove the value of S[i] temporarily.
      if subsetSum(S,k)="no" then
         print this;
         sum := sum + this;
         S[i] := this // Restore the value of S[i].
```

```
      // If "yes", we should not restore the value of S[i]. Why?
      end if;
      i := i + 1
   end while
end
```

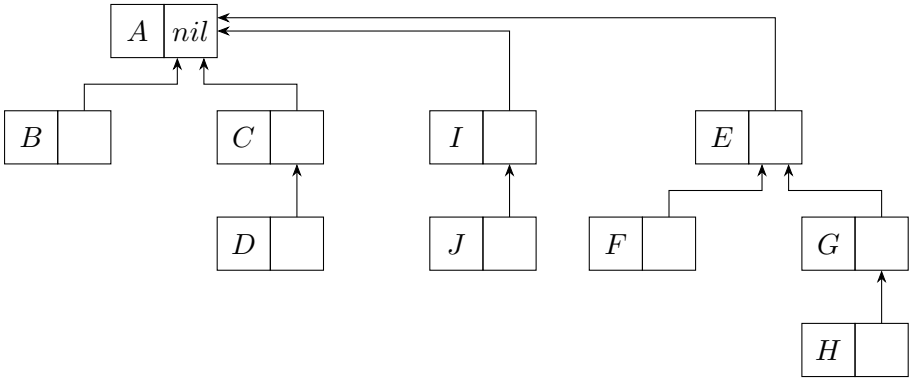
□

6. Consider the solutions to the union-find problem discussed in class. Suppose we start with a collection of ten elements: A, B, C, D, E, F, G, H, I, and J.

   (a) Assuming the balancing, but not path compression, technique is used, draw a diagram showing the grouping of these ten elements after the following operations (in the order listed) are completed: union(A,B), union(C,D), union(E,F), union(G,H), union(I,J), union(A,D), union(F,G), union(D,J), union(J,H).

   In the case of combining two groups of the same size, please always point the second group to the first.

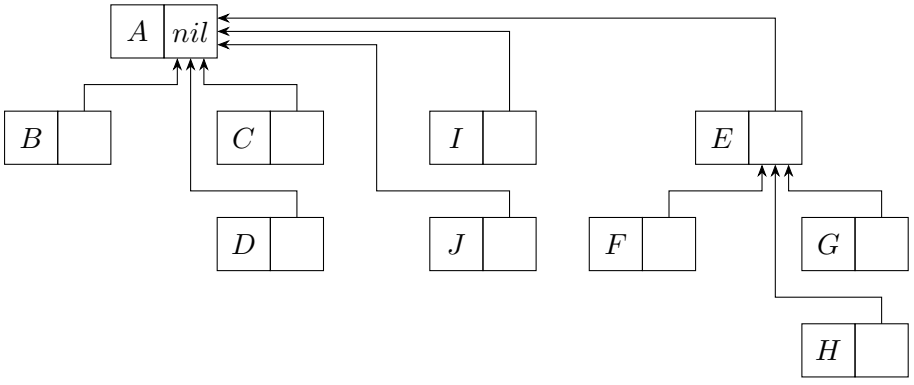
   Solution.



□

   (b) Repeat the above, but with both balancing and path compression.

   Solution.



□

7. Please present in suitable pseudocode the algorithm (discussed in class) for rearranging an array $A[1..n]$ of n integers into a max heap using the *bottom-up* approach.

   Solution.

```
Algorithm Build_Heap(A,n);
begin
   for i := n DIV 2 downto 1 do
      parent := i;
      child1 := 2*parent;
      child2 := 2*parent + 1;
      if child2 > n then child2 := child1 end if;
      if A[child1]>A[child2] then maxchild := child1
      else maxchild := child2 end if;
      while maxchild<=n and A[parent]<A[maxchild] do
         swap(A[parent],A[maxchild]);
         parent := maxchild;
         child1 := 2*parent;
         child2 := 2*parent + 1;
         if child2 > n then child2 := child1 end if;
         if A[child1]>A[child2] then maxchild := child1
         else maxchild := child2 end if
      end while
   end for
end
```

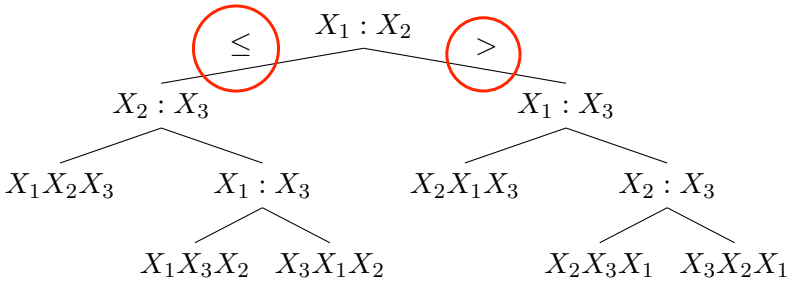
□

8. Below is a variant of the insertion sort algorithm.

Algorithm Insertion_Sort (A, n);
begin
  for $i := 2$ **to** n **do**
    $a := A[i]$;
    $j := i$;
    while $j > 1$ and $A[j-1] > a$ **do**
      $A[j] := A[j-1]$;
      $j := j - 1$;
    end while
    $A[j] := a$;
  end for
end

Draw a decision tree of the algorithm for the case of $A[1..3]$, i.e., $n = 3$. In the decision tree, you must indicate (1) which two elements of the original input array are compared in each internal node and (2) the sorting result in each leaf. Please use X_1, X_2, X_3 (not $A[1]$, $A[2]$, $A[3]$) to refer to the elements (in this order) of the original input array.

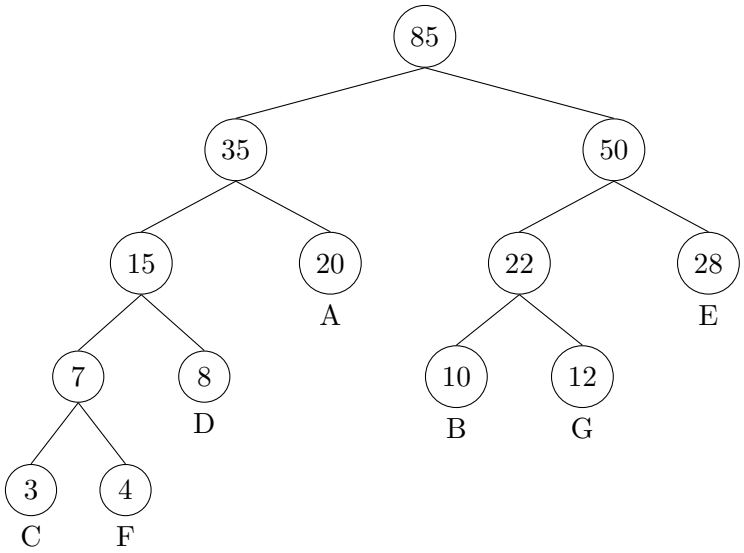
Solution.



□

9. Construct a Huffman code tree for a text composed from seven characters A, B, C, D, E, F, and G with frquencies 20, 10, 3, 8, 28, 4, and 12 respectively. And then, list the codes for all the characters according to the code tree.

   Solution.



Character	Frequency	Code
A	20	01
B	10	100
C	3	0000
D	8	001
E	28	11
F	4	0001
G	12	101

□

10. Given two strings $A = bbaaa$ and $B = bbbaba$, what is the result of the minimal cost matrix $C[0..5, 0..6]$, according to the algorithm discussed in class for changing A character by character into B? Aside from giving the cost matrix, please show the details of how the entry $C[4, 5]$ is computed from the values of $C[3, 4]$, $C[3, 5]$, and $C[4, 4]$.

    Solution.

		b	b	b	a	b	a
	0	1	2	3	4	5	6
b	1	0	1	2	3	4	5
b	2	1	0	1	2	3	4
a	3	2	1	1	1	2	3
a	4	3	2	2	1	2	2
a	5	4	3	3	2	2	2

$$C[4,5] = \min \left\{ \begin{array}{ll} C[3,5]+1 = 2+1 = 3 & (\text{deleting } A_4), \\ C[4,4]+1 = 1+1 = 2 & (\text{inserting } B_5), \\ C[3,4]+1 = 1+1 = 2 & (A_4 \neq B_5) \end{array} \right\} = 2$$

□

Appendix

- The notions of O, Ω, and o are defined as follows.
  - A function $f(n)$ is $O(g(n))$ for another function $g(n)$ if there exist constants c and N such that, for all $n \geq N$, $f(n) \leq cg(n)$.
  - A function $f(n)$ is $\Omega(g(n))$ if there exist constants c and N such that, for all $n \geq N$, $f(n) \geq cg(n)$.
  - A function $f(n)$ is $o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.
- Below is the algorithm discussed in class for determining whether a solution to the (original) Knapsack 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] ≥ 0 then
                    if P[i-1,k-S[i]].exist then
                       P[i,k].exist := true;
                       P[i,k].belong := true
end
```