Suggested Solutions to Midterm Problems

1. Consider the following two-player *counting* game: given a positive integer N, player A and player B take turns counting to N. In her/his turn, a player may advance the count by 1 or 2. For example, player A may start by saying "1, 2", player B follows by saying "3", player A follows by saying "4", etc. The player who eventually has to say the number N loses the game.

   A game is *determined* if one of the two players always has a way to win the game. Prove *by induction* that the counting game as described is determined for any positive integer N; the winner may differ for different given integers. (Hint: think about the remainder of the number N divided by 3.)

   Solution. We first prove the following claim:

   > When $N = 3k + 1$ for some $k \geq 0$, player B can always win the game.

   The proof is by induction on k.

   Base case ($k = 0$, i.e., $N = 1$): player A has no other choice but say 1 and hence player B wins.

   Inductive step ($k \geq 1$, i.e., $N = 3k + 1 \geq 4$): player A starts either by "1" or "1, 2". In both cases, player B can always count to 3. At this point we have the situation analogous to where the two players are to play a game with $N = 3(k-1)+1$, in which player B can always win from the induction hypothesis.

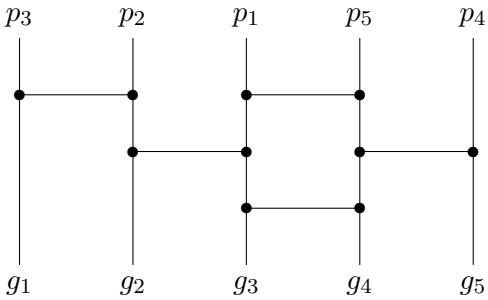
   We next prove a second claim:

   > When $N = 3k+2$ or $N = 3(k+1)$ for some $k \geq 0$, player A can always win the game.

   In the case when $N = 3k + 2$, player A starts by saying "1", while in the case when $N = 3(k + 1)$, he starts by "1, 2". After player A's first turn, we have the situation analogous to that player B is to start a game with $N = 3k + 1$, playing the role of player A (to start first in the remaining game). From the first claim, player A (playing the role of player B in the remaining game) will win the game.

   With the two proven claims put together, we have shown that, for every positive integer N, there is always a player that can win the counting game and hence the game is determined. □

2. We sometimes would use a diagram like the following to distribute n gifts (or assign n tasks) to n people. With the main part of the diagram covered, each person p_i ($1 \leq i \leq n$), without seeing the horizontal line segments, is asked to choose one of the vertical lines. After everyone has made a choice, the whole diagram is revealed. Following the vertical line chosen by p_i, go down along the line and, whenever hitting an intersection, make a turn and switch to a neighboring vertical line (to the left or right). The traced path will eventually reach a gift at the end and the gift is given to p_i.



Prove *by induction* that such a diagram (with arbitrary numbers of vertical and horizontal line segments) always produces a one-to-one mapping between the people and the gifts (whose number equals that of the vertical lines). Assume that the horizontal line segments do not intersect with one another.

Solution. The proof is by induction on the number m of horizontal line segments. Note that, as stated in the problem, the horizontal line segments do not intersect with one another; in particular, no two horizontal line segments share an intersection. For simplicity, but without loss of generality, we assume that the p_i's select the vertical lines in the order of their indices.

Base case ($m = 0$): Since there is no horizontal line segment, p_1 is mapped to g_1, p_2 to g_2, ..., and p_n to g_n, which is a one-to-one mapping between the n people and the n gifts.

Inductive step ($m \geq 1$): Given an arbitrary setting of m horizontal line segments, we remove the line segment that is highest in position; if there are several such line segments, remove one of them. From the induction hypothesis, the new setting of $m-1$ horizontal line segments defines a one-to-one mapping between the people and the gifts. Let us refer to the mapping as f, which maps p_i to $g_{f(i)}$. Suppose the removed line segment originally connected vertical lines i and $i+1$. We claim that, with the removed line segment restored, the original setting also defines a one-to-one mapping; call it f'. Clearly, $f'(i) = f(i+1)$, $f'(i+1) = f(i)$, and $f'(j) = f(j)$ for any other j. It follows that, given f is one-to-one, f' is also one-to-one. □

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 $f(n) < cg(n)$." (Note: $\neg\exists c\exists N(P(c,N)) \equiv \forall c\neg(\exists N(P(c,N))) \equiv \forall c(\forall N(\neg P(c,N)))$.) 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 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 described in class an algorithm to solve the problem. Modify the algorithm to solve a variation of the Knapsack problem where the i-th item has additionally an associated value v_i. Find a way or determine it is impossible to pack the knapsack (of size K) fully, such that the items in it have the maximal total value among all possible ways to pack the knapsack. Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will be credited for this problem. (Hint: both the values of $P(i-1,k)$ and $P(i-1,k-S[i])$ should be considered.)

   Solution.

```
Algorithm Knapsack_Valued (S, v, K);
begin
    P[0,0].exist := true;
    P[0,0].belong := 0;
    P[0,0].value := 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;
            P[i,k].value := 0;
            if P[i-1,k].exist then
               P[i,k].exist := true;
               P[i,k].belong := 0;
               P[i,k].value := P[i-1,k].value
            end if
            if k - S[i] ≥ 0 then
               if P[i-1,k-S[i]].exist then
                  P[i,k].exist := true;
                  if P[i-1,k-S[i]].value + v[i] > P[i,k].value then
                     P[i,k].belong := 1;
                     P[i,k].value := P[i-1,k-S[i]].value + v[i]
                  end if
               end if
            end if
        end for
    end for
end
```

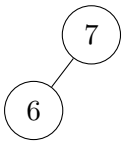
   From the main nested for-loops, we see that the time complexity is $O(nK)$. (Note: the bound should be understood as $O(n2^{\log K})$, where $\log K$ represents the input size of the number K.) □

5. Show all intermediate and the final AVL trees formed by inserting the numbers 7, 6, 5, 2, 1, 3, and 4 (in this order) into an empty tree. Please use the following ordering convention: the key of an internal node is larger than that of its left child and smaller than that of its right child. If re-balancing operations are performed, please also show the tree before re-balancing and indicate what type of rotation is used in the re-balancing.

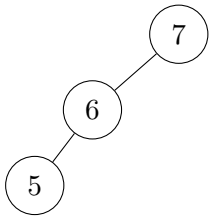
Solution.

Insert 7:

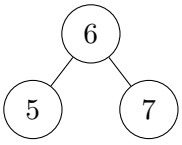
Insert 6:



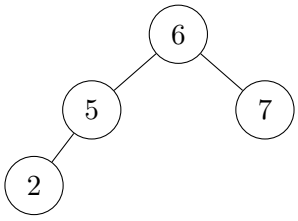
Insert 5:



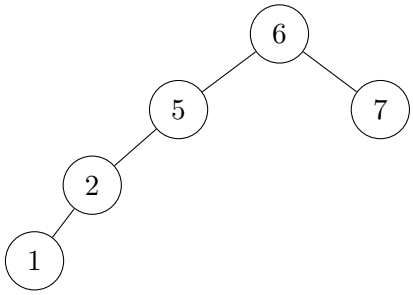
Single rotation at 7:



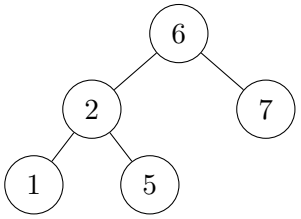
Insert 2:



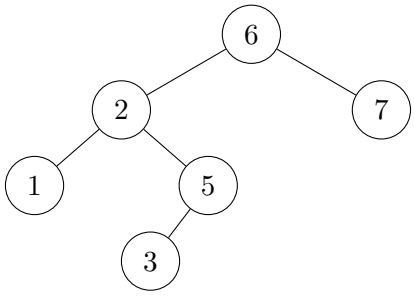
Insert 1:



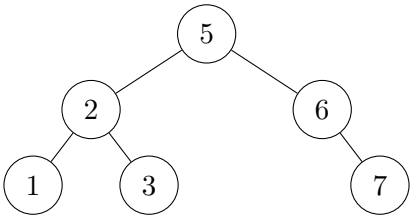
Single rotation at 5:

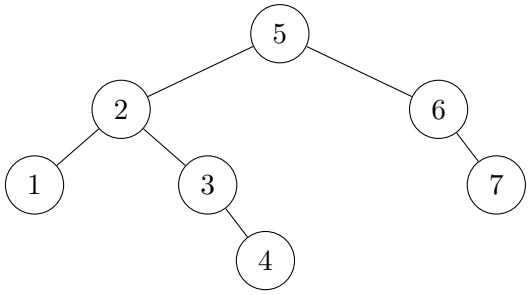


Insert 3:



Double rotation at 6:





Insert 4:

□

6. When analyzing the average-case time complexity of the Quicksort algorithm, we started with the following equation:

$$T(n) = n + 1 + T(i-1) + T(n-i), \text{ where } n \geq 2,$$

assuming the i-th smallest element of the input array is selected as the pivot.

What does the term $n+1$ account for? Please explain why it is $n+1$ (not any other value).

Solution. The term $n+1$ accounts for the total number of comparisons between two input numbers performed during the partition procedure, given an input array of size n (≥ 2). In some special cases, only n comparisons are needed.

For an array X of n numbers, entries indexed from 1 through n, the first number $X[1]$ is selected as the *pivot*, which is assumed to be the i-th smallest among the n numbers. Suppose $n \geq 3$, $2 \leq i < n$, and the numbers are all distinct so that both pointers L and R will stay within the range $[2..n]$; the extreme cases (such as $i = 1$ or n) may be reasoned in an analogous way just with a bit more care on the boundaries. The left pointer L goes from 2 towards the right (the larger indices), while the right pointer R goes from n towards the left (the smaller indices).

The two pointers will eventually cross each other, i.e., when $L \leq R$ becomes *false* ($L > R$ becomes *true*), and the main while loop terminates. At that moment, $L = R+1$ $(= i+1)$ or $R = L-1$ $(= i)$. This is the case, as $X[L-1] \leq pivot$ and $X[L] > pivot$, after the last iteration (if any) of the while loop for advancing L, and the last iteration (if any) of the while loop for advancing R brings R to its current value (which becomes less than L for the first time), with $X[R+1 = L] > pivot$ and $X[R = L-1] \leq pivot$.

The input number in $X[j]$, for $2 \leq j \leq L$ and also for $R \leq j \leq n$ or $L-1 \leq j \leq n$, is compared (before it has possibly been swapped with another number) against the pivot (which is an input number). In total, $(L-2+1) + (n-(L-1)+1) = n+1$ comparisons between two input numbers are performed. □

7. Consider rearranging the following array into a max heap using the *bottom-up* approach.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5	2	8	4	1	15	7	6	3	11	10	12	13	14	9

Please show the result (i.e., the contents of the array) after a new element is added to the current collection of heaps (at the bottom) until the entire array has become a heap.

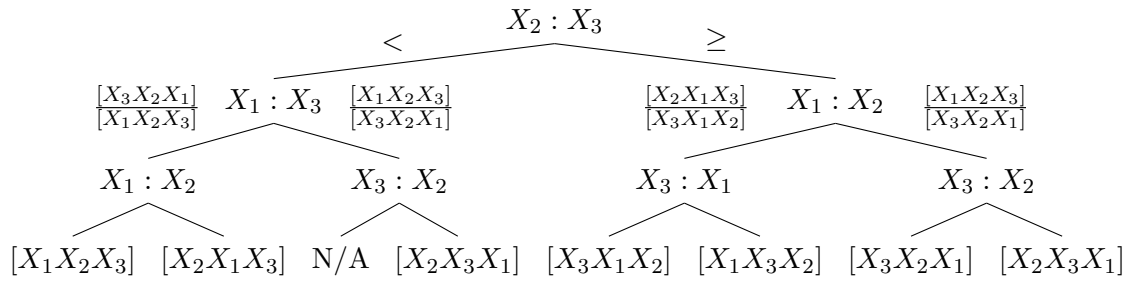
Solution.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5	2	8	4	1	15	7	6	3	11	10	12	13	14	9
5	2	8	4	1	15	$\underline{14}$	6	3	11	10	12	13	$\underline{7}$	9
5	2	8	4	1	15	14	6	3	11	10	12	13	7	9
5	2	8	4	$\underline{11}$	15	14	6	3	$\underline{1}$	10	12	13	7	9
5	2	8	$\underline{6}$	11	15	14	$\underline{4}$	3	1	10	12	13	7	9
5	2	$\underline{15}$	6	11	$\underline{13}$	14	4	3	1	10	12	$\underline{8}$	7	9
5	$\underline{11}$	15	6	$\underline{10}$	13	14	4	3	1	$\underline{2}$	12	8	7	9
$\underline{15}$	11	$\underline{14}$	6	10	13	$\underline{9}$	4	3	1	2	12	8	7	$\underline{5}$

□

8. Draw a decision tree of the Heapsort algorithm (in increasing order) 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 A.

   Solution. The Heapsort algorithm starts by building a (max) heap. For an input array of size 3, no matter the top-down or bottom-up approach is used, X_2 and X_3 are compared first and the larger will be compared against X_1. After two comparisons, the heap is built. Subsequently, the top element is swapped with the last (third) element and then the third (last) comparison is performed between the current top and its left child to turn the array into a heap (of 2 elements). The remaining steps do not need any comparison between input numbers.



   Note: two or more of X_1, X_2, and X_3 may be equal. □

9. Design an algorithm that, given a set of integers $S = \{x_1, x_2, \ldots, x_n\}$, 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, $x_1 + x_2$, $x_1 + x_2 + x_3$, $\ldots$, $x_1 + x_2 + \cdots + x_{n-1}$, and $x_1 + x_2 + \cdots + x_{n-1} + x_n$; the difference between any two of these is also a sum.)

Solution. Consider the n sums: x_1, $x_1 + x_2$, $x_1 + x_2 + x_3$, ..., $x_1 + x_2 + \cdots + x_{n-1}$, and $x_1 + x_2 + \cdots + x_{n-1} + x_n$. If one of them is divisible by n, i.e., $\sum_{i=1}^{i=j} x_i \equiv 0 \pmod{n}$ for some j, $1 \leq j \leq n$, then we have the nonempty subset we look for. Otherwise, two of these sums must have the same remainder when divided by n, as there are n sums and only $n-1$ possible nonzero remainders. The difference of the two sums, in the form of $x_{j+1} + x_{j+2} + \cdots + x_k$ for some j and k, $1 \leq j < k \leq n$, will be the sum of the subset we look for. So, the existence of the nonempty subset is guaranteed and, from the argument, we also have the algorithm to find it.

In the following pseudocode, $R[m]$ records the right-most index (namely j) of the sum $x_1 + x_2 + \cdots + x_j$ first encountered whose remainder is m when divided by n. Upon seeing a second sum $x_1 + x_2 + \cdots + x_k \equiv m \pmod{n}$, the algorithm prints two index values $R[m] + 1$ and k, $1 \leq R[m] < k \leq n$, indicating that $x_{R[m]} + x_{R[m]+1} + \cdots + x_k = 0 \pmod{n}$.

```
Algorithm FindSubset (S, n);
begin
    for m := 1 to n − 1 do
        R[m] := 0
    end for;
    Sum := 0;
    for k := 1 to n do
        Sum := Sum + S[k];
        m := Sum % n;
        if m = 0 then
            Print 1, k;
            Halt
        end if;
        if R[m] ≠ 0 then
            Print R[m] + 1, k;
            Halt
        else
            R[m] := k
        end if;
    end for;
end
```

The main for-loop determines the time complexity, which is clearly $O(n)$. □

10. Consider the *next* table as in the KMP algorithm (the version presented in class) 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 so that such wasted comparisons can be avoided. Also, please give, for a new string $B[1..9] = bbbabbbaa$, the values of the original *next* table and those of the new *next* table according to the adaptation.

Solution.

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**
            $j := next[j] + 1$;
        $next[i] := j$;
    // Add the following lines for optimization.
    for $i := 2$ **to** m **do**
        $j := next[i] + 1$;
        if $j > 0$ and $B[i] = B[j]$ **then**
            $next[i] := next[j]$;
    /* Alternatively, perhaps clearer but less efficient
    for $i := m$ **down to** 2 **do**
        $j := next[i] + 1$;
        while $j > 0$ and $B[i] = B[j]$ **do**
            $j := next[j] + 1$;
        $next[i] := j - 1$;
    */
end

For $B[1..9] = bbbabbbaa$, the original *next*:

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

The new *next*:

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

□

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
```

- Below is the Partition procedure we studied for the Quicksort algorithm:

```
Algorithm Partition(X, Left, Right);
begin
    pivot := X[Left];
    L := Left + 1;  R := Right;
    while L ≤ R do
        while L ≤ Right and X[L] ≤ pivot do L := L + 1;
        while R ≥ Left and X[R] > pivot do R := R - 1;
        if L < R then
            swap(X[L], X[R]);
            L := L + 1;
            R := R - 1;
    Middle := R;
    swap(X[Left], X[Middle])
end
```

- The algorithm for computing the *next* table in the KMP algorithm:

```
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] ≠ B[j] and j > 0 do
            j := next[j] + 1;
        next[i] := j
end
```