Suggested Solutions to Midterm Problems

1. Prove *by induction* that every natural number greater than or equal to 12 is a non-negative linear combination of 4 and 5, i.e., for every $n \in \mathbb{N}$, if $n \geq 12$, then there exist $a, b \in \mathbb{N}$ s.t. $n = 4a + 5b$ (where $\mathbb{N}$ is the set of all natural numbers, including 0).

   Solution. The proof is by induction on n.

   Base case ($n = 12$): in this case, $n = 12 = 4 \times 3 + 5 \times 0$. So, the problem statement is true for $n = 12$.

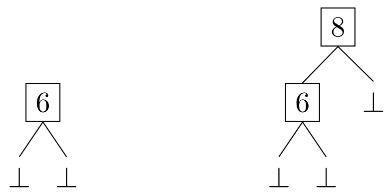
   Inductive step ($n > 12$): from the induction hypothesis we have that $n - 1 = 4a + 5b$, for some $a, b \in \mathbb{N}$. If $a > 0$, then $n = (n-1) + 1 = 4a + 5b + 1 = 4a + 5b + (5 - 4) = 4(a-1) + 5(b+1)$. Otherwise ($a = 0$), since $n > 12$ and $n - 1 > 11$, $n - 1 = 5b$ for some $b \geq 3$ and $n = 5b + 1 = 5b + (4 \times 4 - 5 \times 3) = 4 \times 4 + 5(b - 3)$. Therefore, the problem statement is also true for $n > 12$. □

2. The set of all full binary trees that store non-negative integer key values may be defined inductively as follows.

   (a) $FBT(k, \perp, \perp, 0)$, for any non-negative integer k, is a full binary tree of height 0.

   (b) If t_l and t_r are full binary trees of height h, then $FBT(k, t_l, t_r, h+1)$, for any non-negative integer k, is a full binary tree of height $h + 1$.

   Please give a similar inductive definition for the set of all complete binary trees (of the form $CBT(\cdot, \cdot, \cdot, \cdot)$) that store non-negative integer key values; you may reuse FBT in parts of your definition. For instance, $CBT(6, \perp, \perp, 0)$ is a single-node complete binary tree storing key value 6 and $CBT(8, CBT(6, \perp, \perp, 0), \perp, 1)$ is a complete binary tree with two nodes — the root and its left child, storing key values 8 and 6 repsectively. Pictorially, they may be depicted as below.



   Solution. The set of all (non-empty) complete binary trees may be defined as follows:

   (a) $CBT(k, \perp, \perp, 0)$, for any non-negative integer k, is a complete binary tree of height 0, and
   $CBT(k_1, CBT(k_2, \perp, \perp, 0), \perp, 1)$, for any non-negative integers k_1 and k_2, is a (proper) complete binary tree of height 1.

   (b) Suppose t_l and t_r are complete binary trees.

      i. If t_l is a full binary tree of height h and t_r is a complete binary tree of height h, then $CBT(k, t_l, t_r, h+1)$, for any non-negative integer k, is a complete binary tree of height $h + 1$.

ii. If t_l is a complete tree of height h and t_r is a full binary tree of height $h-1$, then $CBT(k, t_l, t_r, h+1)$, for any non-negative integer k, is a complete binary tree of height $h+1$.

Here by saying "a complete binary tree t is a full binary tree," we mean that, when every occurrence of CBT in $t = CBT(\cdot,\cdot,\cdot,\cdot)$ is replaced by FBT, it is indeed a full binary tree according to the definition of $FBT(\cdot,\cdot,\cdot,\cdot)$. (Mathematically, a CBT is a full binary tree if it is isomorphic to some FBT.)

Note: as the definition is intended to exclude the empty tree as a complete binary tree, it includes as a base case the (proper) complete binary tree (with two nodes) of height 1, to avoid the need, in the inductive step, of treating the special case of a complete binary tree without a right child. □

3. Consider bounding summations by integrals. We already know that, if $f(x)$ is monotonically *increasing*, then
$$\sum_{i=1}^{n} f(i) \leq \int_{1}^{n+1} f(x)dx.$$

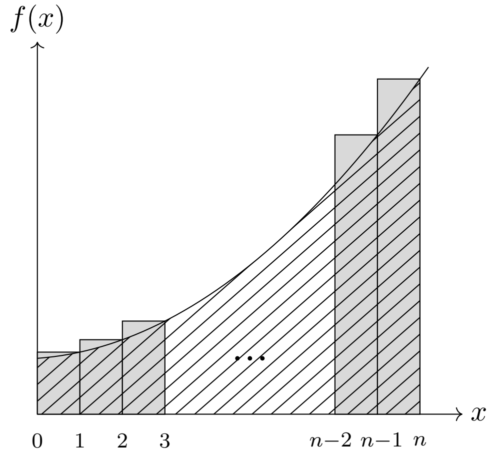
(a) The sum may also be bounded from below as follows:
$$\int_{0}^{n} f(x)dx \leq \sum_{i=1}^{n} f(i).$$

Show that this is indeed the case.

Solution. Given that $f(x)$ is monotonically increasing, we have

$$\begin{array}{rcl}
\int_0^1 f(x)dx & \leq & f(1) \\
\int_1^2 f(x)dx & \leq & f(2) \\
\int_2^3 f(x)dx & \leq & f(3) \\
& \cdots & \\
\int_{n-2}^{n-1} f(x)dx & \leq & f(n-1) \\
\int_{n-1}^{n} f(x)dx & \leq & f(n) \\
\hline
\int_0^n f(x)dx & \leq & \sum_{i=1}^{n} f(i)
\end{array}$$

So, the lower bound for the summation $\sum_{i=1}^{n} f(i)$ is correct. This is also easily seen by comparing the areas (on the $R \times R$ plane) defined by the formulae on the two sides. As shown in the following diagram, the integral $\int_0^n f(x)dx$ equals the area under the curve that is shaded with thin parallel lines. The area is apparently no larger than the total area of the vertical bars which represents $\sum_{i=1}^{n} f(i)$.



$\square$

(b) Prove, using this bounding technique, that $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. Note that $\frac{1}{i}$ actually decreases when i increases.

Solution. As $\frac{1}{i}$ is monotonically decreasing and the bounding technique cannot be directly applied, we rewrite the sum as $\sum_{i=1}^{n} \frac{1}{(n+1)-i}$. Now we have a monotonically increasing $f(x) = \frac{1}{(n+1)-x}$, for $x < n+1$. We know that $\int \frac{1}{(n+1)-x} dx = -\ln((n+1)-x)$, for $x < n+1$.

$\sum_{i=1}^{n} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{(n+1)-i} \geq \int_0^n \frac{1}{(n+1)-x} dx = -\ln((n+1)-n) - (-\ln((n+1)-0)) = \ln(n+1) \geq \ln n \geq \frac{1}{\log e} \log n$. So, $\sum_{i=1}^{n} \frac{1}{i} = \Omega(\log n)$.

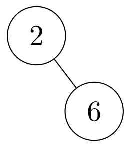
$\sum_{i=1}^{n} \frac{1}{i} = \sum_{i=1}^{n} \frac{1}{(n+1)-i} = 1 + \sum_{i=1}^{n-1} \frac{1}{(n+1)-i} \leq 1 + \int_1^n \frac{1}{(n+1)-x} dx = 1 + (-\ln((n+1)-n) - (-\ln((n+1)-1))) = 1 + \ln n \leq \frac{1}{\log e} \log n + \frac{1}{\log e} \log n \leq \frac{2}{\log e} \log n$ (for $n \geq 3$). So, $\sum_{i=1}^{n} \frac{1}{i} = O(\log n)$.

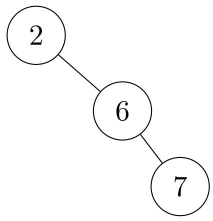
It follows that $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. $\square$

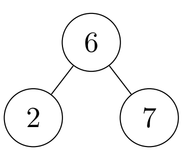
4. Show all intermediate and the final AVL trees formed by inserting the numbers 2, 6, 7, 1, 5, 3, and 4 (in this order) into an empty tree. Please use the following ordering convention: the key of an internal node is larger than that of its left child and smaller than that of its right child. If re-balancing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

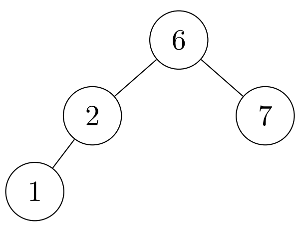
   Solution.

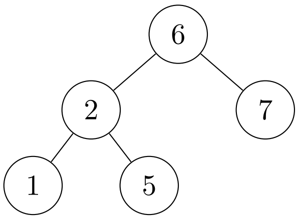
Insert 2: 

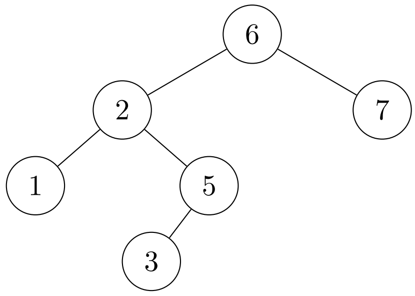
Insert 6: 

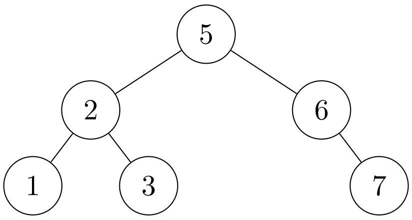
Insert 7: 

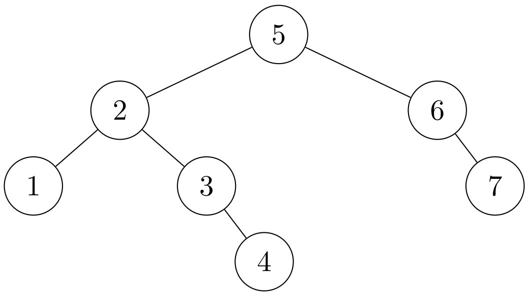
Single rotation at 6: 

Insert 1: 

Insert 5: 

Insert 3: 

Double rotation at 6: 

Insert 4: 

□

5. Below is the pseudocode of the binary search algorithm we discussed in class. Would the code still be correct if we change the assignment "$\mathit{Middle} := \lceil \frac{\mathit{Left}+\mathit{Right}}{2} \rceil$" to "$\mathit{Middle} := \lfloor \frac{\mathit{Left}+\mathit{Right}}{2} \rfloor$" for *Middle* to take instead the largest integer less than or equal to $\frac{\mathit{Left}+\mathit{Right}}{2}$? Please justify your answer. If the modified code is incorrect, what other changes must be made accordingly?

function Find $(z, \mathit{Left}, \mathit{Right}) : \mathit{integer}$;
begin
    if $\mathit{Left} = \mathit{Right}$ **then**
        if $X[\mathit{Left}] = z$ **then** $\mathit{Find} := \mathit{Left}$
        else $\mathit{Find} := 0$
    else
        $\mathit{Middle} := \lceil \frac{\mathit{Left}+\mathit{Right}}{2} \rceil$;
        if $z < X[\mathit{Middle}]$ **then**
            $\mathit{Find} := \mathit{Find}(z, \mathit{Left}, \mathit{Middle} - 1)$
        else
            $\mathit{Find} := \mathit{Find}(z, \mathit{Middle}, \mathit{Right})$
end

Algorithm Binary_Search (X, n, z);
begin
    $\mathit{Position} := \mathit{Find}(z, 1, n)$;
end

Solution. The code would be incorrect, if just that change is made. Consider $X[1..2] = [7, 9]$, an array with two numbers 7 and 9. Suppose we invoke **Binary_Search**$(X, 2, 6)$ to find out whether 6 is in X. The call in turns invokes $Find(6, 1, 2)$, whose execution will set $Middle$ to $\lfloor \frac{Left+Right}{2} \rfloor = \lfloor \frac{1+2}{2} \rfloor = 1$. Since $z = 6 < 7 = X[1] = X[Middle]$, the execution will invoke $Find(z, Left, Middle - 1)$, i.e., $Find(6, 1, 0)$, which will result in an access to $X[0]$, an erroreous behavior.

The nested conditional statement should be modified accordingly as follows.

function Find $(z, Left, Right) : integer;$
begin
    if $Left = Right$ **then**
        ...
    else
        $Middle := \lfloor \frac{Left+Right}{2} \rfloor;$
        if $z \leq X[Middle]$ **then**
            $Find := Find(z, Left, Middle)$
        else
            $Find := Find(z, Middle + 1, Right)$
end

□

6. Given the array below as input [to the Mergesort algorithm], what are the contents of array *TEMP* after the merge part is executed for the first time and what are the contents of *TEMP* when the algorithm terminates? Assume that each entry of *TEMP* has been initialized to 0 when the algorithm starts.

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

Solution. The contents of array *TEMP* after the merge part is executed for the first time:

1	2	3	4	5	6	7	8	9	10	11	12
2	3	0	0	0	0	0	0	0	0	0	0

The contents of array *TEMP* when the algorithm terminates:

1	2	3	4	5	6	7	8	9	10	11	12
1	2	3	4	5	6	7	8	9	0	0	0

□

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;
```

```
        if A[child1]>A[child2] then maxchild := child1
        else maxchild := child2;
        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;
           if A[child1]>A[child2] then maxchild := child1
           else maxchild := child2
        end
     end
  end
```

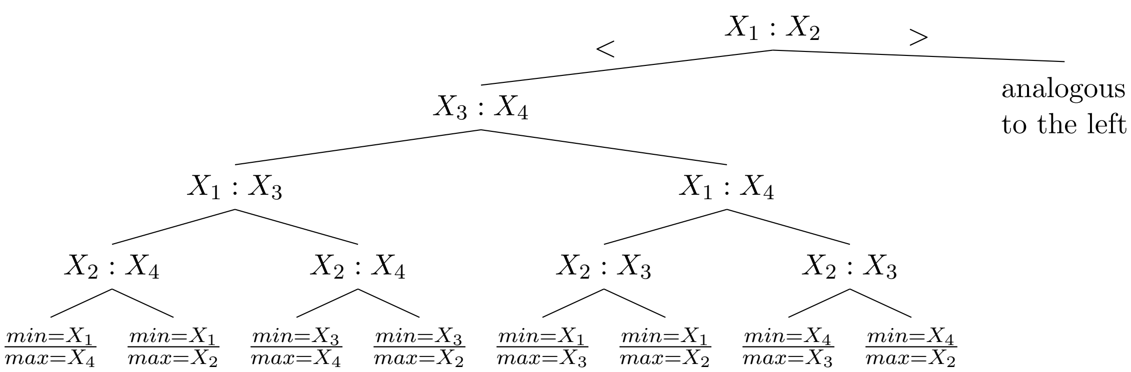
□

8. We have studied in class an algorithm, outlined again below, for finding the minimum and the maximum of a sequence of numbers.

    Compare the first two numbers (assuming the input sequence is of even length). Set *min* to be the smaller of the two and *max* the larger. Compare the next pair of numbers and then compare the smaller with *min* and the larger with *max*. Update *min* and *max* accordingly. Continue until we have exhausted the sequence.

   Draw a decision tree of the algorithm for the case of an input sequence of four *distinct* numbers. In the decision tree, you must indicate (1) which two elements of the original sequence are compared in each internal node and (2) the output (the values of *min* and *max* respectively) in each leaf. Please use X_1, X_2, X_3, X_4 to refer to the numbers (in this order) in the original input sequence.

   Solution.



□

9. Consider the text data compression problem we have discussed in class; the problem statement is given below.

Given a text (a sequence of characters), find an encoding for the characters that satisfies the prefix constraint and that minimizes the total number of bits needed to encode the text.

Prove that the two characters with the lowest frequencies must be among the deepest leaves (farthest from the root) in the final code tree. (Hint: proof by contradiction.)

Solution. Denote the characters in the text by c_1, c_2, $\cdots$, c_n and their frequencies by f_1, f_2, $\cdots$, f_n. Given an encoding E in which a bit string s_i represents c_i, the length (number of bits) of the text encoded by using E is $\sum_{i=1}^{n} |s_i| \cdot f_i$. In the code tree corresponding to E, the depth of the leaf representing character c_i equals the length of the encoding s_i for c_i. We observe that at the deepest level in the code there must be at least two leaves; otherwise, we may remove the only leaf and take its parent as a new leaf, obtaining a better code tree.

Assume toward a contradiction that one of the two characters, say c_j, with the lowest frequencies is at a level shallower than that of a character, say c_k, with a higher frequency such that $|s_j| < |s_k|$. Since $|s_j| < |s_k|$ and $f_j < f_k$, $(|s_k| - |s_j|) \cdot f_k > (|s_k| - |s_j|) \cdot f_j$ and $|s_j| \cdot f_j + |s_k| \cdot f_k > |s_j| \cdot f_k + |s_k| \cdot f_j$. It follows that

$$\sum_{i=1}^{n} |s_i| \cdot f_i > (\sum_{i=1, i \neq j, i \neq k}^{n} |s_i| \cdot f_i) + |s_j| \cdot f_k + |s_k| \cdot f_j.$$

If we swap the characters c_j and c_k, then we will get a better code tree, a contradiction. □

10. 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, so that such wasted comparisons can be avoided. Also, please give the values of the *next* table for the string $B[1..9] = abbaabbaa$, 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 five lines for optimization.
    for i := m down to 2 do
        j := next[i] + 1;
        while B[i] = B[j] and j > 0 do
            j := next[j] + 1;
        next[i] := j − 1;
end
```

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

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

The new $next$:

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

□

Appendix

- The Mergesort algorithm:

```
Algorithm Mergesort (X, n);
begin  M_Sort(1, n)  end

procedure M_Sort (Left, Right);
begin
    if Right − Left = 1 then
        if X[Left] > X[Right] then swap(X[Left], X[Right])
    else if Left ≠ Right then
            Middle := ⌈½(Left + Right)⌉;
            M_Sort(Left, Middle − 1);
            M_Sort(Middle, Right);
            // the merge part
            i := Left;  j := Middle;  k := 0;
            while (i ≤ Middle − 1) and (j ≤ Right) do
                    k := k + 1;
                    if X[i] ≤ X[j] then
                        TEMP[k] := X[i];  i := i + 1
                    else TEMP[k] := X[j];  j := j + 1;
            if j > Right then
                for t := 0 to Middle − 1 − i do
                    X[Right − t] := X[Middle − 1 − t]
            for t := 0 to k − 1 do
                X[Left + t] := TEMP[1 + t]
end
```

- The KMP algorithm (assuming $next$):

```
Algorithm String_Match (A, n, B, m);
begin
```

```
    j := 1;  i := 1;
    Start := 0;
    while Start = 0 and i ≤ n do
        if B[j] = A[i] then
            j := j + 1;  i := i + 1
        else
            j := next[j] + 1;
            if j = 0 then
                j := 1;  i := i + 1;
        if j = m + 1 then Start := i − m
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
```