Suggested Solutions to Midterm Problems

Problems

1. Prove *by induction* that the sum of the heights of all nodes in a *full* binary tree of height h is $2^{h+1} - h - 2$ and that the sum equals $n - (h + 1)$, where n is the total number of nodes in the tree. (Note: a single-node tree has height 0.)

   Solution. (Chih-Pin Tai)

   It is sufficient to prove that the sum of the heights of all nodes in a full binary tree of height h is $2^{h+1}-h-2$. For a full binary tree of height h, it is well known that $n = 2^{h+1}-1$ and it follows that the sum equals $2^{h+1} - 1 - (h + 1) = n - (h + 1)$. The proof is by induction on the height h.

   Base case: When $h = 0$, the tree has only one node which is of height 0 and so the sum of the heights of all nodes is 0 which equals $2^{0+1} - 0 - 2$.

   Inductive step: When $h = k + 1$, we proceed as follows. A full binary tree of height $k + 1$ is composed of one root node whose height is $k + 1$ and two full binary trees of height k. The induction hypothesis states that when $h = k$, the sum of the heights of all nodes in a full binary tree of height k equals $2^{k+1} - k - 2$. Therefore, the sum for a full binary tree of height $k + 1$ equals $(k + 1) + 2 \times (2^{k+1} - k - 2)$, which equals $2^{(k+1)+1} - (k + 1) - 2$. □

2. The *Partition* procedure for the Quicksort algorithm discussed in class is as follows, where *Middle* is a global variable.

   Partition $(X, Left, Right)$;
   begin
   $pivot := X[Left]$;
   $L := Left$; $R := Right$;
   while $L < R$ **do**
   while $X[L] \leq pivot$ and $L \leq Right$ **do** $L := L + 1$;
   while $X[R] > pivot$ and $R \geq Left$ **do** $R := R - 1$;
   if $L < R$ **then** $swap(X[L], X[R])$;
   $Middle := R$;
   $swap(X[Left], X[Middle])$
   end

   Find an adequate loop invariant for the main while loop, which is sufficient to show that after the execution of the last two assignment statements the array is properly partitioned by $X[Middle]$. Please express the loop invariant as precisely as possible, using mathematical notation.

Solution. The algorithm assumes that $Left < Right$. This condition holds throughout the algorithm and we will keep it implicit. A suitable loop invariant for the main while loop is as follows:

$$\begin{array}{ll} & pivot = X[Left] \\ \wedge & \forall i(Left \leq i < L \rightarrow X[i] \leq pivot) \\ \wedge & \forall j(R < j \leq Right \rightarrow pivot < X[j]) \\ \wedge & Left \leq L \leq Right \\ \wedge & Left \leq R \leq Right \\ \wedge & (L \not< R) \rightarrow (L - 1 = R) \end{array}$$

This loop invariant is maintained before and after every iteration of the loop. Note that the inequalities $i < L$ and $R < j$ in the second and third conjuncts are strict. This is so because when the condition $L < R$ does not hold, the statement $swap(X[L], X[R])$ will not be performed. After the while loop terminates with $L \not< R$ and the following two statements are executed, we can conclude:

$$\begin{array}{ll} & pivot = X[Middle] \\ \wedge & \forall i(Left \leq i \leq Middle \rightarrow X[i] \leq pivot) \\ \wedge & \forall j(Middle < j \leq Right \rightarrow pivot < X[j]) \end{array}$$

which is the (post-)condition desired of the Partition algorithm, indicating that the algorithm is indeed correct. □

3. Find the asymptotic behavior of the function $T(n)$ defined as follows:

$$\begin{cases} T(1) = 1 \\ T(n) = T(n/2) + \sqrt{n}, \quad n = 2^i \ (i \geq 1) \end{cases}$$

You should try to solve this problem without resorting to the general theorem for divide-and-conquer relations (see the Appendix) discussed in class. The asymptotic bound should be as tight as possible. (Hint: an effective way is to guess and verify by induction. You may need to try a few choices.)

Solution. (Yu-Fang Chen)

$T(n) = O(\sqrt{n})$.

We prove by induction on n that $T(n) \leq 7\sqrt{n}$.
Base case: $T(1) = 1 \leq 7 = 7\sqrt{1}$.
Inductive step: $T(2n) = T(n) + \sqrt{2n} \leq 7\sqrt{n} + \sqrt{2}\sqrt{n} = (7 + \sqrt{2})\sqrt{n} \leq 7\sqrt{2}\sqrt{n} = 7\sqrt{2n}$.

Note: How was the constant 7 determined? Suppose $T(n) \leq c\sqrt{n}$ for some c. Anticipating the proof obligation in the inductive step that $T(2n) = T(n) + \sqrt{2n} \leq c\sqrt{n} + \sqrt{2}\sqrt{n}$, which should be $\leq c\sqrt{2n} = c\sqrt{2}\sqrt{n}$, we stipulate that the constant c should be $\geq \sqrt{2}/(\sqrt{2} - 1) = 6.\cdots$; 7 is just one of such constants. But, how do we know in the first place that $T(n) = O(\sqrt{n})$ is a good guess? The guess is motivated by considering two other recurrence relations: "$T(n) = T(n/2) + 1$, $T(1) = 1$" (in which case $T(n) = O(\log n)$) and "$T(n) = T(n/2) + n$, $T(1) = 1$" (in which case $T(n) = O(n)$). □

4. Consider a max heap represented as the following array which may store a maximum of 15 elements.

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

(a) Show the resulting heap after *Insert*(14).

Solution. (Yi-Wen Chang)

Appending 14 at the end:

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

After Rearrange_Heap :

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
15	12	<u>14</u>	11	10	9	<u>13</u>	6	8	1	2	4	3	5	<u>7</u>

□

(b) Show the resulting heap after a *Remove*() operation (on the original heap).

Solution. (Yi-Wen Chang)

Swapping the head and the end elements:

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

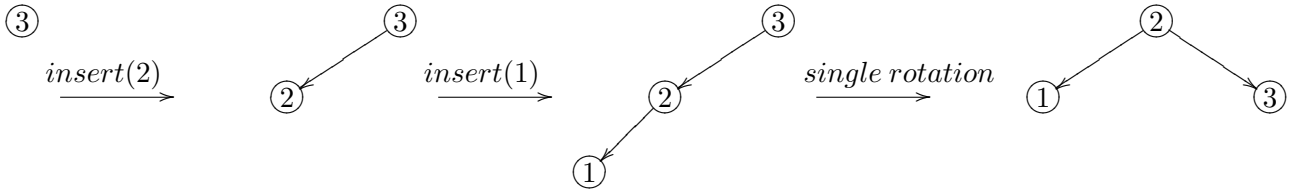
After Rearrange_Heap:

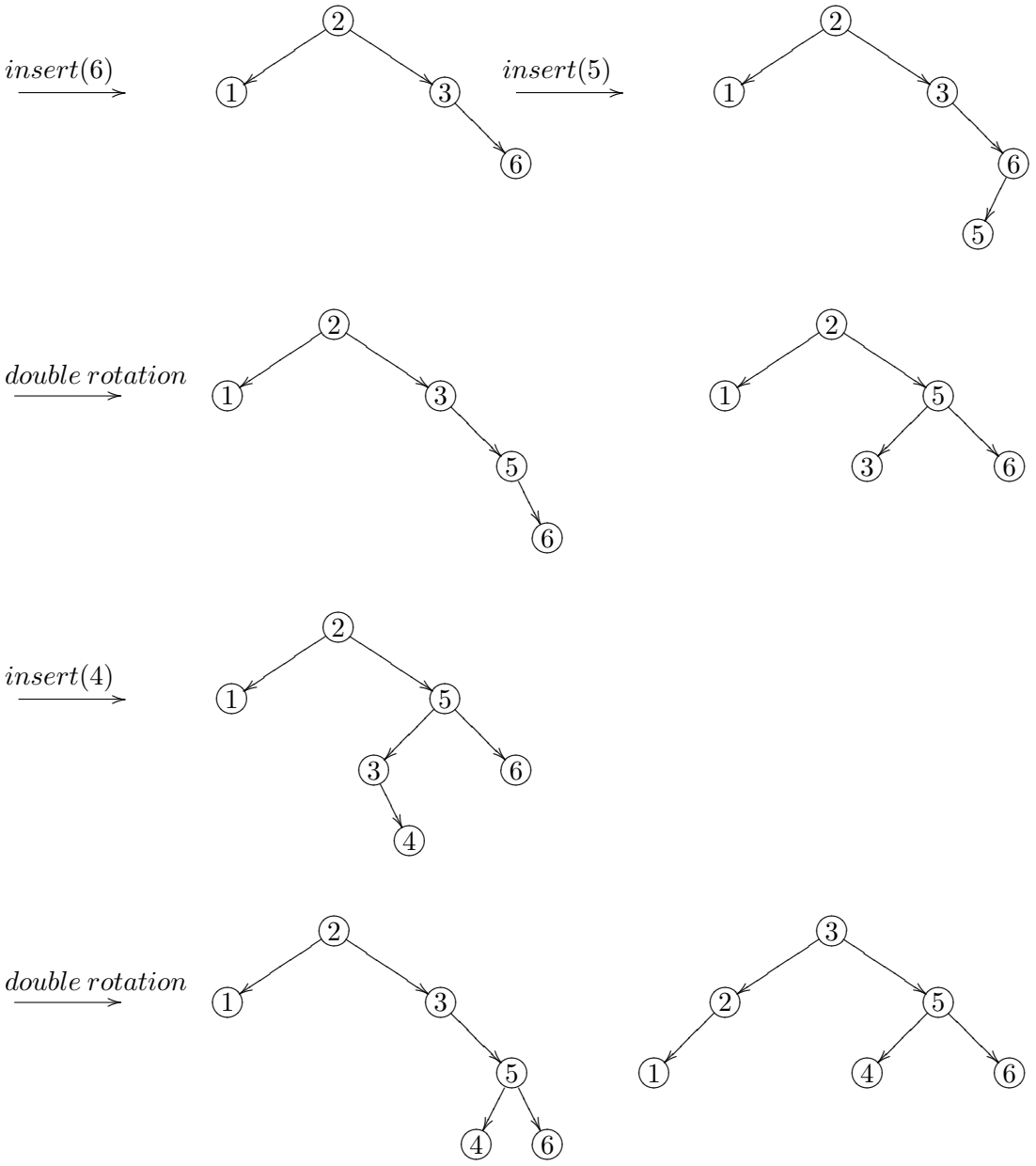
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<u>13</u>	12	<u>9</u>	11	10	<u>5</u>	7	6	8	1	2	4	3		

□

5. Show all intermediate and the final AVL trees formed by inserting the numbers 3, 2, 1, 6, 5, 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. (Chih-Pin Tai)





□

6. Let $x_1, x_2, \cdots, x_n$ be a sequence of real numbers (not necessarily positive). Design an $O(n)$ algorithm to find the subsequence $x_i, x_{i+1}, \cdots, x_j$ (of consecutive elements) such that the product of the numbers in it is maximum over all consecutive subsequences. The product of the empty subsequence is defined to be 1.

   Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary. Explain the intuition behind your algorithm so that its correctness becomes clear.

   Solution. (Wen-Chin Chan, and modified by Jinn-Shu Chang)

```
Algorithm Maximum_Consecutive_Subsequence(X,n)
begin
    Global_Max := 1;
    Suffix_Max := 1;
    Suffix_Min := 1;
    for i := 1 to n do
      if X[i] > 0 then
          if Suffix_Max × X[i] > Global_Max then
              Global_Max := Suffix_Max × X[i];
          Suffix_Max := Suffix_Max × X[i];
          Suffix_Min := Suffix_Min × X[i];
          if Suffix_Max < 1 then
              Suffix_Max := 1;
          if Suffix_Min ≥ 0 then
              Suffix_Min := 1;
      else if X[i] < 0 then
          if Suffix_Min × X[i] > Global_Max then
              Global_Max := Suffix_Min × X[i];
          Suffix_Max := Suffix_Max × X[i];
          Suffix_Min := Suffix_Min × X[i];
          swap(Suffix_Max, Suffix_Min);
          if Suffix_Max < 1 then
              Suffix_Max := 1;
      else /* X[i] = 0 */
          Suffix_Max := 1;
          Suffix_Min := 1;
end
```

□

7. The Knapsack Problem 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.

   Now consider a variant where we want the subset to be as large as possible (i.e., to be with as many items as possible). How will you adapt the algorithm (see the Appendix) that we have studied in class? Your algorithm should collect at the end the items in one of the best solutions if they exist. Please present your algorithm in an adequate pseudo code and make assumptions wherever necessary (you may reuse the code for the original Knapsack Problem). Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will get for this problem.

   Solution. (Yi-Wen Chang)

To find the largest possible subset of items, we modify the *Knapsack* algorithm in the Appendix to obtain *Knapsack_ForMaxSubset*, as shown below. Each element $P[i,k]$ in the result array P now contains a new integer variable *size* which memorizes the size of the current largest subset for k.

Algorithm *Knapsack_ForMaxSubset*(S,K)**;**
begin
    $P[0,0].exist := true$;
    $P[0,0].size := 0$;
    for $k := 1$ **to** K **do**
        $P[0,k].exist := false$;
        $P[0,k].size := 0$;
    for $i := 1$ **to** n **do**
        for $k := 0$ **to** K **do**
            $P[i,k].exist := false$;
            $P[i,k].size := 0$;
            if $k - S[i] \geq 0$ and $P[i-1,k-S[i]].exist$ **then**
                if $P[i-1,k].exist$ and $P[i-1,k].size \geq P[i-1,k-S[i]].size + 1$ **then**
                    $P[i,k].exist := true$;
                    $P[i,k].belong := false$;
                    $P[i,k].size := P[i-1,k].size$;
                else
                    $P[i,k].exist := true$;
                    $P[i,k].belong := true$
                    $P[i,k].size := P[i-1,k-S[i]].size + 1$;
            else if $P[i-1,k].exist$ **then**
                $P[i,k].exist := true$;
                $P[i,k].belong := false$;
                $P[i,k].size := P[i-1,k].size$;
    if $\neg P[n,K].exist$ **then**
       print "no solution"
    else $i := n$;
        $k := K$;
        while $k > 0$ **do**
            if $P[i,k].belong$ **then**
                print i;
                $k := k - S[i]$;
            $i := i - 1$;
end

The complexity remains the same, which is $O(nK)$. When a solution (a subset of items

whose sizes sum to exactly K) exists, the printed result will be the largest among such subsets.

□

8. Consider an alternative algorithm for partition in the Quicksort algorithm:

```
Partition (X, Left, Right);
begin
    pivot := X[Left];
    i := Left;
    for j := Left + 1 to Right do
        if X[j] < pivot then i := i + 1;
                             swap(X[i], X[j]);
    Middle := i;
    swap(X[Left], X[Middle])
end
```

How does this algorithm compare to the algorithm we discussed in class? Please point out the advantages and the disadvantages of this alternative with adequate justification.

Solution. Let us first try to understand this Partition algorithm. The basic idea is to maintain three contiguous regions in the array, using the two indices i and j. From left to right, the first region contains elements that are known to be less than or equal to the pivot. The second region contains elements that are known to be greater than the pivot. The third region contains elements that are yet to be processed, i.e., to be put into the first or the second region. Index i marks the right end of the first region, j marks the left end of the third region, and in between is the second region. In each iteration of the for loop, depending on the value of the first element (pointed to by j, when the execution reaches the if statement) of the third region, one of the following two changes takes place:

- When $X[j] < pivot$, the first region is expanded by one element to the right by swapping $X[i+1]$ and $X[j]$, causing the second region to be shifted also by one element to the right.
- When $X[j] \geq pivot$, the second region is expanded by one element to the right (via simply incrementing j).

We now compare this algorithm with the Partition algorithm discussed in class, which we refer to as the "Sides-to-Middle" algorithm.

- Advantages: In this algorithm, *pivot* is compared against $n-1$ array elements, while in "Sides-to-Middle" $n+1$ such comparisons are made. This algorithm is simpler in structure and perhaps less prone to mistake for implementation, but this is rather subjective.

- Disadvantages: A swap is needed in this algorithm whenever $X[j] < pivot$. So, the total number of swaps equals the number of elements that are smaller than *pivot*. In contrast, "Sides-to-Middle" usually needs less swaps. For example, $5, 6, 1, 2, 3, 4$ will need 4 swaps in this algorithm, but only 1 swap in "Sides-to-Middle". Also, in an extreme case, where $i + 1$ equals j (i.e., when the second region is still empty), the swapping of elements is not necessary, but is performed in this algorithm.

□

9. Apply the Quicksort algorithm to the following array. Show the contents of the array after each partition operation. If you use a different partition algorithm (from the one discussed in class), please describe it.

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

Solution.

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

□

10. Your task is to design an *in-place* algorithm that sorts an array of numbers according to a prescribed order. The input is a sequence of n numbers $x_1, x_2, \cdots, x_n$ and another sequence $a_1, a_2, \cdots, a_n$ of n distinct numbers between 1 and n (i.e., $a_1, a_2, \cdots, a_n$ is a permutation of $1, 2, \cdots, n$), both represented as arrays. Your algorithm should sort the first sequence according to the order imposed by the permutation as prescribed by the second sequence. For each i, x_i should appear in position a_i in the output array. As an example, if $x = 23, 9, 5, 17$ and $a = 4, 1, 3, 2$, then the output should be $x = 9, 17, 5, 23$.

    Please describe your algorithm as clearly as possible; it is not necessary to give the pseudo code. Remember that the algorithm must be in-place, without using any additional storage for the numbers to be sorted. Give an analysis of its time complexity. The more efficient your algorithm is, the more points you will get for this problem.

    Solution. Suppose that array X holds the first sequence and array A the second. Sort A increasingly according to the position values that it stores. Every time when two elements in A, say a_i and a_j, are swapped, we also swap the corresponding elements in X, i.e., x_i and x_j. Once the sorting of A is completed, the elements in X are also sorted as prescribed

by A. Any in-place sorting algorithm may be used for sorting A. If we use Heapsort, the complexity is $O(n \log n)$.

In fact, there is a much simpler and faster (linear-time) algorithm. In this algorithm, we scan array A form left to right. Whenever $A[i] \neq i$, we swap x_i and $x_{A[i]}$ and also $A[i]$ and $A[A[i]]$. This is repeated until $A[i] = i$ and we then proceed to the next element in array A. Each swap of x_i and $x_{A[i]}$ brings one element in X to its final destination. So, we ever need to do such swaps at most n times. The corresponding swaps for A are also performed at most n times. Therefore, this algorithm runs in $O(n)$ time. □

Appendix

- The solution of the recurrence relation $T(n) = aT(n/b) + cn^k$, where a and b are integer constants, $a \geq 1$, $b \geq 2$, and c and k are positive constants, is as follows.

$$T(n) = \begin{cases} O(n^{\log_b a}) & \text{if } a > b^k \\ O(n^k \log n) & \text{if } a = b^k \\ O(n^k) & \text{if } a < b^k \end{cases}$$

- Below is an algorithm for determining whether a solution to the 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
```