CSC-421 Applied Algorithms and Structures
Spring 2017

Instructor: Iyad Kanj
Office: CDM 832
Phone: (312) 362-5558
Email: ikanj@cs.depaul.edu
Office Hours: Monday 4:40 - 5:40 PM & Wednesday 4:45 - 6:45 PM
Course Website: https://d2l.depaul.edu/

Solution Key to Assignment #1

1. Let $N = [1..n]$ be the collection of nuts and $B[1..n]$ the collection of bolts. The idea is very similar to the Merge procedure that merges two sorted lists. We keep two pointers i and j, i points to the current position in N and j to that in B. At each step we compare $N[i]$ to $B[j]$ and we either report a match or update the pointers: if $N[i] = B[j]$ then we have a match; if $N[i]$ is smaller than $B[j]$ we increment i; otherwise, we increment j. If we reach the end of one of the arrays, we report no match. The pseudocode is given next.

```
i <-- 1; j <-- 1;

loop forever

if (i > n) or (j > n) then
   return (‘no match exists’);
if N[i] = B[j] then
   return (‘nut’ i ‘matches bolt’ j);
if (N[i] < B[j]) then
    i <-- i + 1;
else j <-- j + 1;
```

Clearly, the number of comparisons is $O(n)$.

2. (a) We use two pointers i and j, i points to the beginning of the array and j to its end. At each step we check the value $(A[i] + A[j])$. If this value is equal to t than we have found the two elements, namely $A[i]$ and $A[j]$. If this value is less than t, then we increment i; otherwise, we decrement j. We keep doing this until either we have found two elements that sum to t or i and j overlap. Clearly, this can be done in linear time, since for every comparison one element in the array is skipped. We give the pseudocode next.

```
1. Call Find_Sum(A, 1, n, t);


Find_Sum(A, i, j, t)

if i < j then
   if (A[i] + A[j] = t) then
       return('Yes');
   else if (A[i] + A[j] < t) then
           return(Find_Sum (A, i + 1, j, t));
        else
          return(Find_Sum(A, i, j - 1, t));

else
  return('No').
```

(b) We first sort the array using any sorting algorithm that runs in $O(n^2)$ time (e.g., Insertion Sort, Merge Sort, etc.). Afterwards, we iterate through the elements of the array, and for each element x, we search the remaining subarray (from that element on) for two elements y, z whose sum is $t - x$ using the algorithm in part (a); this takes $O(n^2)$ time because we apply the algorithm in (a), which runs in $O(n)$ time, $O(n)$ times. The overall running time (including the sorting) is $O(n^2)$.

3. Pinocchio is right. One way of solving the problem is to search the array for every pair of positive integers x, y whose sum is 1000 (i.e., $y = 1000 - x$). The number of such pairs is a constant (500 pairs, allowing for the possibility of x and y being equal), and is independent from the size n of the array. For each such pair x, y, we spend $O(n)$ time searching the array for x and $y = 1000 - x$ (e.g., using Linear Search). The overall running time is $500 \times O(n) = O(n)$.

4. We use the same technique used in Binary Search. At each step we pick the middle index mid of the array. We check if mid is equal to the element stored at mid. If it is we return mid and we are done. If mid is less than the element stored at mid, then since the elements are stored in a strictly increasing order, we know that no index that is greater than mid can satisfy the desired property, and our search can be restricted to the first (left) half of the array. If mid is greater

than the element stored at mid, then the search can be restricted to the second (right) half of the array. The recurrence that we get for the running time of the algorithm is exactly the same as that for Binary Search. Hence, the algorithm runs in $\Theta(\lg n)$ time. The pseudocode is given next.

```
Search(A, i, j)

if i <= j do

   mid <-- (i+j)/2;
   if A[mid] = mid then
      return(mid);
   if A[mid] > mid then
      return (Search(A, i, mid-1));
   if A[mid] < mid then
      return(Search(A, mid+1, j));

else return(Nil);
```

5. (a) Here is the recursive algorithm. The first call is FindMax(A, 1, n).

```
FindMax(A, first, last)
if first = last then
   return (A[first]);
else
   middle <-- (first+last)/2;
   m <-- FindMax(A, first, middle);
   M <-- FindMax(A, middle+1, last);
   if m <= M then return(M)
   else return(m);
```

The number of comparisons is $T(n) = 2T(n/2) + 1$ if $n > 1$, and $T(1) = 0$. Using the iteration method, we get $T(n) = n - 1 = \Theta(n)$. The form after the i-th iteration is $T(n) = 2^i T(n/2^i) + 2^{i-1} + 2^{i-2} + \ldots + 1 = 2^i T(n/2^i) + 2^i - 1$. Substituting $i = \lg n$ (the number of iterations to reach the base case) gives $T(n) = n - 1$.

(b) No. An iterative algorithm computes the largest number in A by making $n - 1$ comparisons, matching the number of comparisons of the recursive algorithm.