CSC-421 Applied Algorithms and Structures
Winter 2019

Instructor: Iyad Kanj
Office: CDM 832
Phone: (312) 362-5558
Email: ikanj@cs.depaul.edu
Office Hours: Monday & Wednesday 4:00-5:30
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. The idea is to sort the points by their x-coordinate first, and whenever two points have the same x-coordinate, to sort them by their y-coordinate. After sorting the points as described, identical points must appear contiguously/adjacently in the sorted array; so now we can scan the array, checking if any two adjacent points are identical, which can be done in $O(n)$ time. We describe how to sort the points as described above in $O(n \lg n)$ time. To do so, we slightly tweak any

$O(n \lg n)$-time sorting algorithm, say **Merge Sort** for example, as follows. In the procedure **Merge** (in **Merge Sort**), whenever two elements $A[i]$ and $A[j]$ are compared via the comparison "$A[i] \leq A[j]$", we replace the comparison with the following. (Note that $A[i]$ and $A[j]$ contain points.) Suppose $A[i]$ contains point $p_i(x_i, y_i)$ and $A[j]$ contains point $p_j(x_j, y_j)$. We replace "$A[i] \leq A[j]$" with: "$(x_i < x_j)$ or $(x_i = x_j)$ and $(y_i \leq y_j)$". The running time of **Merge**, and hence, of **Merge Sort**, remains the same because the modified comparison still takes constant time. The overall running time of the algorithm is $O(n \lg n)$ (sorting) plus $O(n)$ (scanning the array afterwards), which is $O(n \lg n)$.

5. To show that HAM-PATH is in NP, it suffices to give a polynomial-time algorithm $A(\cdot, \cdot)$ that verifies it. The algorithm A takes an input an instance $x = \langle G, u, v \rangle$ of HAM-PATH and a certificate y, where y is to be interpreted as a sequence of vertices in G. Since y is a sequence of vertices, its length is polynomial in that of x. The algorithm A needs to verify the following: (1) the first vertex in the sequence y is vertex u and the last vertex is v; (2) every vertex in G appears exactly once in y; and (3) there is an edge in G between every two consecutive vertices in y. Clearly, each of conditions (1)-(3) is checkable in polynomial time, and hence A runs in polynomial time. Moreover, if x is a yes-instance of HAM-PATH, then G has a Hamiltonian path P between u and v, and if we pass P as y to A then A will return YES. On the hand, if there is no Hamiltonian path between u and v in G then there is no certificate y on which the algorithm will return YES. It follows that A verifies HAM-PATH in polynomial time, and HAM-PATH is in NP.

6. A formula F in Disjunctive Normal Form (DNF) is satisfiable if and only if one of the conjunctive terms (i.e., a term consisting of "AND's" of literals) separated by an OR in F is satisfiable. In turn, a conjunctive term in F is satisfiable if and only if it does not contain two opposite literals. Clearly, based on the above, it can be decided in polynomial time whether or not a formula F in DNF is satisfiable.

7. Suppose (hypothetically) that A is a polynomial-time algorithm that decides SAT. Let F be a formula in CNF over variable $\{x_1, \ldots, x_n\}$. First, we run A on F; if A rejects F (i.e., returns NO) we return that there is no assignment that satisfies F. Suppose now that A returns YES when run on F, and we describe how to find a truth assignment τ satisfying F (we know in this case that such an assignment must

exist). We start by picking variable x_1, and we try to figure out the value of x_1 in some satisfying assignment to F. To do so, we set $x_1 = 0$, and substitute the value $x_1 = 0$ in F to obtain another CNF formula F_0. We then run A on F_0. If A returns YES on F_0, then we know that there exists a satisfying assignment to F that sets $x_1 = 0$. So we set the value of x_1 to 0 in τ, and we repeat the above process on F_0 —- whose variables are $\{x_2, \ldots, x_n\}$ — by considering variable x_2 next. On the other hand, if A returns NO on F_0, then we know that no satisfying assignment to F sets $x_1 = 0$. Therefore, we set $x_1 = 1$ in τ, substitute the value $x_1 = 1$ in F to obtain a CNF formula F_1, and repeat the above process on F_1 — whose variables are $\{x_2, \ldots, x_n\}$ — by considering variable x_2. We continue the above process until we finally stop when we have finished considering the last variable x_n, and at that point we have found a satisfying assignment τ to F. To analyze the running time of the algorithm described above, notice that the algorithm makes at most $n + 1$ calls to A (one call per variable plus the initial call). Since A is assumed to run in polynomial time, and since substituting the value of a variable in a formula to obtain the resulting formula can clearly be done in polynomial time, it follows that the overall running time of this algorithm is polynomial.