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/

Solutions to Assignment #2

1. (a) This problem is solvable in polynomial time. We first determine how many coins of denomination x and how many coins of denomination y we have. Suppose we have n_x coins of denomination x and n_y coins of denomination y, where $n_x + n_y = n$. We try each possible split of the n_x coins into two parts, one containing i coins and the other $n_x - i$ coins, and of the n_y coins into two parts, one containing j coins and the other containing $n_y - j$ coins. For each such split, we check if $i \times x + j \times y = (n_x - i) \times x + (n_y - j) \times y$. If we get equality for any split, we return 'Yes'. Otherwise, we return 'No'. Clearly, the running time is polynomial in n.

   (b) This problem is NP-complete. It is easy to see that it is in NP (the proof is similar to that of Subset Sum). To show its NP-hardness, we give a polynomial-time reduction from Subset Sum. Given an instance $X = \{x_1, \ldots, x_n\}$ and $t \in \mathbb{N}$ of Subset Sum, we create an instance of the given problem as follows. Let $x = x_1 + \cdots + x_n$. Define a collection of checks C as follows. Add a check of value x_i, for each $i = 1, \ldots, n$, to C. If $t < x - t$, add a check of value $x - 2t$ to C; if $t > x - t$, add a check of value $2t - x$ to C. The additional check ensures that X has a subset S that sums to t (and hence a subset $X - S$ that sums to $x - t$) if and only if C can be partitioned into two parts of equal value. Clearly, the reduction that takes X, t and produces C runs in polynomial time. It follows that the problem in question is NP-complete.

   (c) This problem can be solved in polynomial time by enumerating every subset of 5 vertices in G, and checking if it is an independent set. If any such subset passes the test, we return 'Yes'. Otherwise, we return 'NO'. There are $\binom{n}{5} = O(n^5)$ many such subsets, where n is the number of vertices in G. Clearly, the running time is polynomial.

2. I will skip this one because we did a similar example in class.

3. I will skip this one because we did a similar example in class.

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) Each sublist has k elements and can be sorted in time $\Theta(k^2)$ using Insertion-Sort. Since we have n/k lists, all the lists can be sorted in time $(n/k)\Theta(k^2) = \Theta(nk)$.

   (b) We do the merging in levels. At the bottom level, level 0, we have n/k sublists, each of size k. We group the sublists into groups each containing two sublists, and we merge them using the same merge subroutine that we used in Merge-Sort. Merging two sublists takes time linear in the total number of elements in the two sublists. The total time taken by the algorithm at the bottom level is $O(n)$. The number of sublists to be merged at level 1 becomes $n/2k$, we again apply the same strategy grouping the sublists into groups each containing two sublists and we merge them. The time spent to merge the sublists at level 1 is also $O(n)$. We keep repeating this step until we have one whole list. Since at each level the number of sublists is reduced by half, it will take us $\lg(n/k)$ levels to reach the top level where we have only one list. At each level the time spent is $O(n)$. The total running time is $O(n \lg(n/k))$.

(c) $k = \Theta(\lg n)$.