CSE 321 - Introduction to Algorithm Design
Homework 03
Deadline: 23:55 November 14th, 2016

1. a) If we measure the size of an instance of the problem of computing the greatest common divisor of m and n by the size of the second parameter n, by how much can the size decrease after one iteration of Euclid's algorithm?

[SOLUTION]

a. Since the algorithm uses the formula $\gcd(m, n) = \gcd(n, m \bmod n)$, the size of the new pair will be $m \bmod n$. Hence it can be any integer between 0 and $n-1$. Thus, the size n can decrease by any number between 1 and n.

b) Prove that an instance size will always decrease at least by a factor of 2 after two successive iterations of Euclid's algorithm. Let r = m mod n. Investigate two cases of r's value relative to n's value.

[SOLUTION]

b. Two consecutive iterations of Euclid's algorithm are performed according to the following formulas:

$$\gcd(m, n) = \gcd(n, r) = \gcd(r, n \bmod r) \quad \text{where } r = m \bmod n.$$

We need to show that $n \bmod r \leq n/2$. Consider two cases: $r \leq n/2$ and $n/2 < r < n$. If $r \leq n/2$, then

$$n \bmod r < r \leq n/2.$$

If $n/2 < r < n$, then

$$n \bmod r = n - r < n/2,$$

too.

2. An algorithm which ensures that each new permutation is created by exchanging only two neighboring elements is called a minimal change algorithm. Design a decrease-by-one algorithm for generating permutations.

[SOLUTION]

Generating Permutations

Almost all work on generating permutations assumes that we are permuting the integers 1..*n*. When not, we use the integers as keys into a sequence of non-integer values.

There is a very simple bottom-up decomposition of `permute(n)`: insert *n* at all possible locations of all the members of `permute(n-1)`.

```
                1

         21              12

    321 231 213     312 132 123
```

This works, but the output is less than ideal. Notice the change between 213 and 312: the numbers in different positions are two spots apart. On larger sequences, the gap can, of course, be even larger. But algorithms that *use* the permutations can sometimes benefit from having the changed values in successive elements always be in consecutive positions.

An algorithm which ensures that each new permutation is created by exchanging only two neighboring elements is called a **minimal change** algorithm. We can add a small detail to our approach that makes it satisfy the minimal change requirement:

```
INPUT: n, an integer

p   ← permute(n-1)
end ← left
for each item in p
    start at end and insert n in all possible positions
    toggle end [left ↔ right]
```

If we do this in the example above, we would insert the 3s into 21 starting on the left, and into 12 starting on the right:

```
                1

         21              12

    321 231 213     123 132 312
```

Notice: every change in the bottom row swaps consecutive values.

How efficient is the minimal-change approach? Time-wise, we can't do much better. But think about space. The algorithm has to generate *and store* all the permutations for *n*-1, *n*-2, ..., down to 1. That is expensive.

How about this, which may not look like a decrease-by-1 algorithm but which is very much in the same spirit:

- Generate one permutation.
- Try to generate another unique permutation.

This creates an implicit sequence permutations. The new idea is to *morph* one element repeatedly until all possiblilities have been generated. We can "seed" the process with the trivial [1..*n*] permutation.

The **Johnson Trotter algorithm** embodies this idea. It uses two new definitions:

- Each number has a *direction*, all initialized 'left'.
- A number is **mobile** if its arrow points to an adjacent smaller number.

For example, consider this sequence in the permutation of [1..4]:

```
 3 2 4 1
 → ← → ←
```

3 and 4 are mobile; 1 and 2 are not.

Here is the algorithm:

```
ALGORITHM: johnson-trotter(n)
INPUT    : integer n

initialize A = [1 2 3 ... n]
initialize D = [← ← ← ... ←]

while there exists a mobile element
    k ← the largest mobile integer in A
    swap k and the element it points to
    reverse the direction of all elements in A larger than k
```

3. Outline an algorithm for deleting a key from a binary search tree. Consider separately three cases: (1) the key's node is a leaf; (2) the key's node has one child; (3) the key's node has two children.

a) Would you classify this algorithm as a variable-size-decrease algorithm?

[SOLUTION]

a. This is an important and well-known algorithm. Case 1: If a key to be deleted is in a leaf, make the pointer from its parent to the key's node null. (If it doesn't have a parent, i.e., it is the root of a single-node tree, make the tree empty.) Case 2: If a key to be deleted is in a node with a single child, make the pointer from its parent to the key's node to point to that child. (If the node to be deleted is the root with a single child, make its child the new root.) Case 3: If a key K to be deleted is in a node with two children, its deletion can be done by the following three-stage procedure. First, find the smallest key K' in the right subtree of the K's node. (K' is the immediate successor of K in the inorder traversal of the given binary tree; it can be also found by making one step to the right from the K's node and then all the way to the left until a node with no left subtree is reached). Second, exchange K and K'. Third, delete K in its new node by using either Case 1 or Case 2, depending on whether that node is a leaf or has a single child.

This algorithm is not a variable-size-decrease algorithm because it does not work by reducing the problem to that of deleting a key from a smaller binary tree.

b) What is the time efficiency class of your algorithm?

[SOLUTION]

b. Consider, as an example of the worst case input, the task of deleting the root from the binary tree obtained by successive insertions of keys $2, 1, n, n-1, ..., 3$. Since finding the smallest key in the right subtree requires following a chain of $n-2$ pointers, the worst-case efficiency of the deletion algorithm is in $\Theta(n)$. Since the average height of a binary tree constructed from n random keys is a logarithmic function (see Section 5.6), we should expect the average-case efficiency of the deletion algorithm be logarithmic as well.

4. Suppose that an array contains n numbers, each of which is −1, 0, or 1. Then, the array can be sorted in O(n) time in the worst case. Prove or disprove that statement.

[SOLUTION]

Solution: **True.** We may use counting sort. We first add 1 to each of the elements in the input array such that the precondition of counting sort is satisfied. After running counting sort, we subtract 1 from each of the elements in the sorted output array.

A solution based on partitioning is as follows. Let $A[1\,..\,n]$ be the input array. We define the invariant

- $A[1\,..\,i]$ contains only -1,
- $A[i+1\,..\,j]$ contains only 0, and
- $A[h\,..\,n]$ contains only $+1$.

Initially, $i = 0$, $j = 0$, and $h = n + 1$. If $h = j + 1$, then we are done; the array is sorted. In the loop we examine $A[j+1]$. If $A[j+1] = -1$, then we exchange $A[j+1]$ and $A[i+1]$ and we increase both i and j with 1 (as in partition in quicksort). If $A[j + 1] = 0$, then we increase j with 1. Finally, if $A[j + 1] = +1$, then we exchange $A[j + 1]$ and $A[h - 1]$ and we decrease h by 1.

5. Given the array A[1...n] of sorted distinct integers, design a divide and conquer algorithm that finds and index i such that A[i] = i. Your algorithm should run in O(log n) time.

```
Search(A,i,j,key){
    int mid=(i+j)/2;
    if(A[mid]==key && mid==key)  then  return mid;
    else if(A[mid] < key)  then  return Search(A,i,mid-1,key);
    else  then  return -1;
}
```

$$T(n) = 2T(n-1) + 1$$

If we solve the recurrence, $T(n) = O(logn)$