CSI 2110 Fall 2014
Computer Science University of Ottawa

Assignment #1 (20 points, weight 5%)
Due: Saturday October 10, 11:55PM
The assignment must be uploaded on Virtual Campus by the deadline
Late assignments are accepted for a maximum of 24 hours and they will receive a 30% penalty.
You may type or write by hand legibly and scan it; always submit a single PDF file.

---

1. (4 points) Suppose that a given algorithm takes 500ms in the worst case to process an input of size N. How much time will it take to process an input with twice this size, i.e. size $2N$, in the following situations:

   (a) the worst-case running time of the algorithm is constant, that is $T(N) = c$, for a constant $c > 0$.

   (b) the worst-case running time of the algorithm is linear, that is $T(N) = cN$, for a constant $c > 0$.

   (c) the worst-case running time of the algorithm is quadratic, that is $T(N) = cN^2$, for a constant $c > 0$.

   (d) the worst-case running time of the algorithm is exponential with base 2, that is $T(N) = c\ 2^N$, for a constant $c > 0$.

2. (5 points) Formally prove the following statements (i.e. for the positive statements, find suitable constants c and n_0 according to the definitions of classes of functions big-Oh, Theta and Omega).

   (a) Prove that $(n+1)^5$ is $O(n^5)$.

   (b) Prove that 3^{n+2} is $O(3^n)$.

   (c) Prove that n^2 is $\Omega(n \log n)$.

   (d) Prove that n^3 is not $O(n^2)$.

   (e) Prove that $\log(n^{10} + n^2)$ is $O(\log n)$.

(continue in the next page)

3. (5 points) Next we give an algorithm to generate all permutations of distinct numbers contained in an array. Give the exact number of swaps (calls to the method `swap`) as well as the big-Oh of this algorithms as a function of the array size n. Justify well your answer.

```
/* recursive function to generate permutations */
void perm (int v[], int n, int i) {
 /* this function generates the permutations of the array
 * from element i to element n-1 */
 int j;
 /* if we are at the end of the array, we have one permutation
 * we can use (here we print it; you could as easily hand the
 * array off to some other function that uses it for something else) */
 if (i == n) {
     for (j=0; j<n; j++) print(v[j]);
 } else
    /* recursively explore the permutations starting
    * with the element at index i going through index n-1 */
    for (j=i; j<n; j++) {
        /* consider the array with i and j switched */
        swap (v, i, j);
        /* recursively call "perm" to permute elements from positions i+1 to n-1
        perm (v, n, i+1);
        /* swap the elements back the way they were */
        swap (v, i, j);
      }
}
```

4. (6 points) You have available two stacks `P1` and `P2` and a queue `F`. Letters `a, b, c, d, e,` are added to stack `P1` in this order, with `e` being the top element. `P2` and `F` are initially empty.

   (a) (1pt) What will be the contents of the stack `P1` after the following sequence of operations?

   ```
   for i = 1 to 5
        P2.push(P1.pop())
   for i = 1 to 5
        P1.push(P2.pop())
   ```

   (b) (1pt) Consider again stack `P1` with its initial contents `a,b,c,d,e,` with `e` being the top element. What will be the contents of the stack `P1` after the following sequence of operations?

   ```
   for i = 1 to 5
        F.enqueue(P1.pop())
   for i = 1 to 5
        P1.push(F.dequeue())
   ```

   (c) (4 pt) Note how the two previous pieces of code manipulate the elements without using temporary variables. Also without using any temporary variables, write an algorithm that will swap arbitrary elements i and j in the stack `P1` possibly using `P2` and `F` as intermediate data structures. Assume $0 \leq i < j < N$, with N being the number of elements in `P1` and 0 being the position of the element on top of the stack. After running your algorithm, the stack `P1` must contain the same elements in the same order except for elements i and j that will have switched places.