UIT2201: CS & the IT Revolution
Tutorial Set 4 (Fall 2016)

(D-Problems discussed on Friday, 02-Sep-2016)
(Q-Problems due on Tuesday, 06-Sep-2016)

Practice Problems: (not graded)

PP-problems will help you to "ease into" the materials covered in the course. *(If you have difficulties with these, please **quickly** see your classmates or the instructor for help.)*

T4-PP1: Probs 1 on page 75 (Ch2.3.3) of [SG6]. (not found in [SG3])
What part(s) of the sequential search algorithm of Figure 2.13 would need to be changed if our phone book contained 1 million names rather than 10,000?

T4-PP2: Prob 4 on p.75 (Ch2.3.3) of [SG6]. (was Prob 2 of p.66 of [SG3])
Describe exactly what would happen to the *Find-Largest* algorithm in Figure 2.14 (or Find-Max in Lecture notes) if you tried to apply it to an empty list of length n=0. Describe exactly how you could fix this problem.

T4-PP3: Prob 5 on p.75 (Ch2.3.3) of [SG6]. (was Prob 3 of p.66 of [SG3])
Describe exactly what would happen to the *Find-Largest* algorithm in Figure 2.14 (or Find-Max in Lecture notes) when it is presented with a list with exactly one item, i.e., n=1.

T4-PP4: Prob 6 on p.75 (Ch2.3.3) of [SG6] (not found in [SG3]).
Would the *Find-Largest* algorithm in Figure 2.14 (or Find-Max in Lecture notes) still work correctly if the test on Line 7 (of Figure 2.14) were changed to ($A_i \geq$ largest-so-far)? Explain why or why not.

T4-PP5: Prob 16 on p.85 (Ch2) of [SG6]. (was Prob. 16 in p.76 of [SG3]).
On Line 6 of the *Find-Largest* algorithm in Figure 2.14 (or Find-Max in Lecture notes), is an instruction that reads,

```
while (i ≤ n) do
```

Explain exactly what would happen if we changed that instruction to read as follows:
(i) `while (i ≥ n) do`
(ii) `while (i < n) do`
(iii) `while (i = n) do`

Discussion Problems: -- Prepare (individually or in groups) for tutorial discussion.

T4-D1: (Swapping X and Y)
Suppose we have two variables X and Y. We want an algorithm that swaps the value of X and Y.
For example, if X=15, Y=22 as input, then we want the end result to be X=22, Y=15.

(a) Show (via example), that the two seemingly obvious Algorithms below do NOT work (in general):

```
Algorithm Bad-1          Algorithm 2Bad-2
 1. X <-- Y;              1. Y <-- X;
 2. Y <-- X;              2, X <-- Y;
```

(b) For what special case will these algorithms *actually work*?

(c) Give a correct swap algorithm that performs the required exchange.

T4-D2: (Cyclic Shift Algorithm)
In T4-D1, we discussed the swap algorithm that performs the exchange: `X <==> Y`, where the arrows indicate that X is to be assigned the value of Y, and Y to be assigned the old value of X.
(a) Design an algorithm that makes the following exchanges: `A <== B <== C <== D <== A`
where the ==> arrows indicate that A is to be assigned the old value of B, B to be assigned the old value of C, C to be assigned the old value of D, and D to be assigned the old value of A. Namely, perform a *cyclic left-shift* of A, B, C, and D.

T4-D3: [Hamming Distance]
The Hamming distance is often used to measure *similarity* between two strings (of characters). Given two strings of length p, stored in arrays `S[1..p]` and `R[1..p]`, the *Hamming distance* between them (namely between `S[1..p]` and `R[1..p]`) is the number of characters positions where they differ.
For example, the Hamming distance between `ABCDEFG` and `AKCDSFG` is 2 since they differ at the 2nd and 5th positions. The Hamming distance between `KLMN` and `ALMA` is 2. The Hamming distance between any string and itself is always 0.

Design a simple algorithm, called **`Hamming-Dist(S,R,p)`** to compute the Hamming distance between two arrays (strings) `S[1..p]` and `R[1..p]`, each of length p. Your algorithm can assume that the two strings have already been stored in the arrays `S` and `R`.
[*Hint:* Modify from the standard iterative loop.]

T4-D4: (Two Important Processes in CS)

(a) [Repeated-Doubling] Start with the number 1. Repeatedly multiply by two until we get a number greater than or equal to *n*. How many steps will you take? Let D(n) be the number of steps.
[In a nice Table, list the value of D(n), for *n* = 1-25, 31-33, 63-65, 127-129, 10^3, 10^6, 10^9.]

(b) [Repeated-Halving] Start with a number *n*. Repeatedly "divide by two (and throw away the remainder)" until we reach 0. How many steps will you take? Let H(n) be the number of steps.
[Add one more column to the table in (a) to list of H(n) for the same values of n.]

(c) What is the relationship between process **(a)** and **(b)** [for the same *n*]?

(d) Express the processes described in **(a)** and **(b)** as algorithms given in pseudo-code.

Problems to be Handed in for Grading by the Deadline:
(**Note:** Please submit *hard copy* to me. Not just soft copy via email.)

T4-Q1: (10 points)
(a) (5 points) [Exercising the Find-Max Algorithm]
Consider the "Find-Max" algorithm given in the lectures (or the FindLargest algorithm in Figure 2.10, p.63 of [SG]). Run the Find-Max algorithm for each of the following *problem instances* and for each, print out all the different values of the variable Max-sf during the execution of the algorithm.
(Note: Print only when the value of Max-sf *changes*.)

- n=8, A = [2, 6, 5, 2, 1, 7, 9, 8]
- n=8, A = [5, 1, 4, 2, 7, 3, 8, 6]
- n=8, A = [1, 1, 2, 2, 2, 4, 5, 6]
- n=8, A = [8, 5, 3, 1, 7, 8, 4, 7]
- n=8, A = [1, 2, 3, 5, 6, 7, 8, 9]

(b) (5 points) [Algorithm for Counting Occurrences of Num]
Design an algorithm **`Count-Occ (A, n, Num)`** to count the number of times that the number `Num` appears in a list of numbers `A[1..n]` of n numbers. (`A` is an array (or list) with n numbers.) For example, if `A[1..8] = (2, 3, 7, 7, 4, 3, 2, 7)` and `Num=7`, then `Num` appears 3 times.
(Note: You can modify the linear-scan algorithm `Array-Sum` given in class.)

T4-Q2: (10 points) [Algorithm for Minima]
(a) Write an algorithm for **`Find-Min(D,n)`** that finds the *minimum* of *n* numbers in the list `D[1..n]`.
(b) Suppose we call your algorithm with **`Find-Min(A,1)`** where `A` is the array `A[1..8] = (2, 3, 7, 7, 4, 3, 2, 7)`. Will your algorithm bomb or will it run? what will happen?

T4-Q3: (5 points) [Reversing an Array]
Design an algorithm that will reverse the elements in an array A[1..*n*] = (A[1], A[2], A[3], ... , A[*n*]).
Namely, your algorithm will transform `[4 2 7 5 1 8 3 6] --> [6 3 8 1 5 7 2 4]`.
[**Hint:** Make systematic use of the "swap" operation.]

T4-Q4: (Tennis Tournaments and Finding Maxima)
The US Open tennis tournament is now on (29-Aug to 11-Sep 2016). See http://www.usopen.org/.
The tournament is a knock-out tournament in which pairs of players play against each other (in rounds). The winner continues to the next round, while the loser is "knocked out". This continues until the final in which the last two remaining players plays to decide the final winner (champion).
(a) Explain how this "*knock-out tournament*" process be used to find the *maximum* of n numbers in a list (in your answer, use n=16). How many matches are needed altogether to find the maximum?
(b) Explain how this "*knock-out tournament*" process can be suitably modified to find the maximum of n numbers n is NOT a power of 2. In your answer, use n=13 and n=24.
(c) After finding the maximum using this process, how do we find the next largest number, *without playing the whole tournament all over again*.

(**Note:** For this problem, no need to write algorithm or pseudo-code, but describe as accurately as possible, your method, illustrated with diagrams and examples.)

A-Problems: OPTIONAL Challenging Problems for Further Exploration
A-problems are usually *advanced* problems for students who like a challenge; they are optional. There is no deadline for A-problems -- you can try them if you are interested and turn in your attempts.

A4: (How often is Max-sf updated, on average?)
Consider the Find-Max algorithm covered in class and also used in T4-Q1. Given a *random* permutation of {1,2,3,...,*n*}, determine how many times (*on average*) the variable *Max-sf* is updated.

Some notes for those who like to try this:

1. You need some probability theory to do this, but only *elementary* prob. theory is needed,
2. You can also assume that all *n*! permutations of {1,2,...,*n*} are equally likely, and
3. The answer is *not* *n*/2.

UIT2201: CS & IT Revolution; (Fall 2016); A/P Leong HW