CMPT 307 Homework 1
September 10, 2019
Date due: September 23, 2019 (**in the class**).

1 Practice Problems (Not to be handed in)

1. Problems (Chapter 0) 0.1, 0.2, 0.3

2 Homework Problems (To be handed in)

1. Programming Contest Problems

   Barry's Game You can find the problem statement from
   `www.cs.sfu.ca/~binay/2019/cmpt307/BarrysGame.pdf`.

   Let $(red\#, green\#, black\#)$ represent the input to the problem. In the class we have shown that Barry always wins when the input is of the type $(n, n, *)$, $(n, *, n)$ or $(*, n, n)$ where * indicates any number. This is not sufficient (thanks to a student in the class). Note that Barry will win for inputs $(1, 4, *), (7, 40, *),$ or $(123, 12, *)$. Determine the general input type where Barry always wins.

   Solution: In the class we have mentioned that Barry wins if two of the three input numbers are the same. Barry can also win when the difference of any two of the three colored balls (input numbers) is divisible by 3. Suppose the given input (a, b, c) is such that either $(a-b), (b-c)$ *or* $(c-a)$ is divisible by 3. Without any loss of generality suppose $(a-b)$ mod $3 = 0$ with $a > b$, i.e. $a - b = 3k$ for some integer $k > 0$. Consider the following operation.

   - Take one ball from the first column (red) and another ball from the third column (black).
   - Following the rules of the game, add two green balls to the second column and remove one red and green balls.

   Now the difference of red and green balls = 3k -3. In this way the difference between the number of red and green balls is reduced by 3. By repeating this process we should be able to bring the difference between the number of red and green balls to be zero (i.e. the number of green and red balls are the same). Thus Barry will win. We can therefore conclude that Barry will win if the difference of the number of any two colored balls is divisible by 3. How do you guarantee that there are enough black balls to play the game?

Star Crossed You can find the problem statement from
`www.cs.sfu.ca/~binay/2019/cmpt307/StarCrossed`.

Write an algorithm to solve the Star-Crossed problem. You must analyze the worst case step count of the most dominant steps of the algorithm. What is the space complexity?

Solution Sketch

Surprisingly, the students struggled with this. The vertices of the polygon $< v_1, v_2, , v_n >$ are read first. The edges of the polygon are $(v_1, v_2), (v_2, v_3),$ $, (v_{n-1}, v_n), (v_n, v_1)$. We now determine the total crossing number as follows:

```
total = 0;
for each edge e = (v_i,v_{i+1}) do
    determine the number n(e) of intersections with the other edges. ----(A)
    total = total + n(e);
total = total/2 /* Answer;
```

The time complexity is $O(n^2)$ since the statement labelled (A) takes O(n) operations for each edge e in the for loop.

Can we do better (i.e. sub-quadratic)? I believe it can be done. If interested let me know. We can discuss this in the class towards the end of the semester.

2. An n-degree polynomial $p(x)$ is an equation of the form

$$p(x) = \sum_{i=0}^{n} a_i x^i,$$

where x is a real number and each a_i is a constant.

(a) Describe a simple $O(n^2)$ time method to compute $p(x)$ for a given x.
Solution: The following algorithm computes $p(x)$ in $O(n^2)$ time.

```
{
poly = a0;
for i=1 to n do {
    xpower=1;
    for j= 1 to i do
       xpower = xpower * x; --------(1)
    poly = poly + ai*xpower;
    }
}
```

This algorithm is $O(n^2)$ since the statement (1) is repeated $1+2+\ldots+n$ times.

(b) $p(x)$ can be rewritten as (**Horner's method**)

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + x(a_n))\ldots))).$$

For a given x, now what is the cost of evaluating $p(x)$? How many multiplications and additions are needed?

Solution In this case $p(x)$ can be evaluated in $O(n)$ additions and multiplications. The pseudocode:

```
{
poly = an;
for i = n-1 downto 0 do{
    poly = ai + x*poly;
}
}
```

3. Problem 0.4 of the text.

Solution

(a) Easy [(b)] We can compute X^n recursively as follows:

```
function power(X,n)
  if n = 1 return X;
  if n is even then return (power(x,n/2))^2;
  if n is odd then reurn x*(power(x,floor(n/2))^2);
```

The recursion tree has height $\mathrm{O}(\log_2 n)$. Thus the number of matrix multiplications is $O(\log_2 n)$.

(c) Every time we square a matrix, we double the size of the elements. In the recursive routine each element of the matrix is doubled $\mathrm{O}(\log_2 n)$ times. Therefore, the size of each element during the recursive routine is $\mathrm{O}(2^{\log_2 n})$ which is $O(n)$.

(d) Since a matrix multiplication involves O(1) multiplications and additions (from (a)) and since the size of each element is always $O(n)$, the cost of each matrix multiplication is $O(M(n))$. Since there are $O(\log n)$ matrix multiplications, the running time is $O(M(n)\log n)$.

(e) Here we are assuming $M(n) = n^t, 1 \leq t \leq 2$. The recurrence relation of the recursive algorithm is (after the simplification)

```
T(1) = 1
T(n) = T(n/2) + M(n/2) = T(n/2) + n^t for n > 1, n = 2^k
```

From Master theorem we notice that $a = 1, b = 2, d = t$. From $\frac{a}{b^\alpha} = 1$ we notice that $\alpha = 0 < 1$. Applying the Master theorem we can say that $T(n) = O(n^t) = O(M(n))$. Here the merging cost dominates the splitting cost.