EC 504 – Fall 2020 – Homework 1
Solutions to Written Exercises

Written part due Thursday, Sept. 17, 2019 in the beginning of class. Coding problem due Monday Sept 21 at 11:59PM. Submit both in your directory `/projectnb/alg504/yourname/HW1`

Start reading Chapters 2, 3 and 4 in CLRS .

1. (20 pts)

   (a) In CRLS do Problem 3-2 on page 61.

   (b) Place the following functions in order from asymptotically smallest to largest – using $f(n) \in O(g(n)$ notation. When two functions have the same asymptotic order, put an equal sign between them.

$$n^2 + 3n\log(n) + 5\ ,\ \ n^2 + n^{-2}\ ,\ n^{n^2} + n!\ ,\ n^{\frac{1}{n}}\ ,\ n^{n^2-1}\ ,\ \ln n\ ,\ \ln(\ln n)\ ,\ 3^{\ln n}\ ,$$
$$(1+n)^n\ ,\ n^{1+\cos n}\ ,\ \sum_{k=1}^{\log n}\frac{n^2}{2^k}\ ,\ 1\ ,\ n^2+3n+5\ ,\ n!\ ,\ \sum_{k=1}^{n}\frac{1}{k}\ ,\ \prod_{k=1}^{n}(1-\frac{1}{k^2})\ ,\ (1-1/n)^n$$

Solution: Here is the order, using set notation for less than ($\subset$) or equal ($=$):

$$O(\prod_{k=1}^{n}(1-\frac{1}{k^2})) = O(0) \subset O(1) = O((1-1/n)^n) = O(n^{\frac{1}{n}}) \subset O(\ln(\ln n)) \subset O(\sum_{k=1}^{n}\frac{1}{k}) = O(\ln n)$$
$$\subset O(n^{1+\cos n}) \subset O(3^{\ln n}) \subset O(n^2 + 3n\ln n + 5) = O(n^2 + n^{-2}) = O(n^2+3n+5)$$
$$= O(\sum_{k=1}^{\log n}\frac{n^2}{2^k}) \subset O(n!) \subset O((1+n)^n) \subset O(n^{n^2-1}) \subset O(n^{n^2}+n!) = O(n^{n^2})$$

For instance,

$$\lim_{n\to\infty}\prod_{k=1}^{n}(1-\frac{1}{k^2}) = 0 \in O(0)$$

because the first term in the product is zero. If the first term were not 0, then the product would converge to a constant, as

$$\ln\prod_{k=2}^{n}(1-\frac{1}{k^2}) = \sum_{k=1}^{n}\ln(1-\frac{1}{k^2}) \approx \sum_{k=1}^{n}-\frac{1}{k^2} = \frac{-\pi^2}{6}$$

In general a very useful trick it to take `ln-exp!`of the function ($f(n) = e^{\ln(f(n))}$) followed by the large n limit. For example:

$$(1-1/n)^n = e^{n\ln(1-1/n)} \simeq e^{n(1/n+1/2n^2+\cdots)} \to e^1 = 2.718281828459$$

(I found this going to WolframAlfpha: https://www.wolframalpha.com!)
Also, recall that $e^{\ln n} = n$.
Finally the function $n^{1+\cos n}$ is tricky so we accept any reasonable placements. It doesn't have a smooth monotonic limit at large n. It oscillates between $\Theta(1)$ and $\Theta(n^2)$ getting arbitrarily close both even at integer values. Therefore strictly speaking the best bound is $n^{1+\cos n} \in O(n^2)$ but $n^\alpha \in O(n^{1+\cos n})$ implies $\alpha \le 0$ by the definition in CLRS of "Big Oh".

(c) Substitute

$$T(n) = c_1 n + c_2\, n log_2(n)$$

into $T(n) = 2T(n/2) + n$ to find the values of c_1, c_2 to determine the exact solution.
Extra Credit(5 pt): If you are eager do it more generally case for $T(n) = aT(n/b) + n^k$ with trial solution

$$T(n) = c_1 n^\gamma + c_2 n^k$$

using $b^\gamma = a$. What happens when $\gamma = k$?
Now set $\gamma = k$ but start over with the guess $T(n) = c_1 n^\gamma + c_2 n^\gamma log_2(n)$ to determine new values of c_1, c_2.
Solution: You are give a a `guess` with unknown constants c_1, c_2. To see is a good guess see if you can determine the constants by substituting the `guess` in the RHS (right hand side) and the LHS (left hand side).
First case:

$$\begin{aligned} c_1 n + c_2\, n log_2(n) &= 2[\ c_1 n/2 + c_2\ (n/2) log_2(n/2)\ ] + n \\ &= c_1 n + c_2\, n(log_2(n) - log_2(2)) + n \\ &= c_1 n + c_2\, n log_2(n) - n c_2 + n \end{aligned}$$

Now to see if `RHS = LHS` $c_1 = c_2$ you need to match the n and the $n log_2(n)$ term

$$c_1 = c_1 - c_2 + 1 \quad \text{and} \quad c_2 = c_2 \tag{1}$$

so it works $c_2 = 1$ and any c_1. To determine this you need to base provide a base case of the recursion (e.g. $T(1) = c_1$)).
Second case: With $\gamma \neq k$ the general solution works. Again LHS vs RHS

$$c_1 n^\gamma + c_2 n^k = a[c_1 (n/b)^\gamma + c_2 (n/b)^k] + n^k \tag{2}$$

Two conditions to match term:

$$c_1 = c_1 a/b^\gamma \quad \text{and} \quad c_2 = a/b^k c_2 + 1 \tag{3}$$

Again it works for any c_1 but we need $c_2 = 1/(1 - a/b^k)$. Trouble if $k = \gamma$ because now $c_2 = \infty$ (i.e. it fails!)
So need to start over with something larger by a log. Again try to match LHS and RHS

$$c_1 n^\gamma + c_2 n^\gamma log_2(n) = a[c_1 (n/b)^\gamma + c_2 (n/b)^\gamma \log_2(n/b)] + n^\gamma \tag{4}$$

Now the leading term c_2 works and the lower power is determined

$$c_2 = c_2 \quad \text{and} \quad c_1 = c_1 - c_2 log_2(b) \tag{5}$$

lower term is again determined relative to it. This is general the larger term matches as n goes to infinity and the smaller needs a base number of $T(1)$.

2. (20 pts) Below you will find some functions. For each of the following functions, please provide:

- A recurrence $T(n)$ that describes the worst-case runtime of the function in terms of n *as provided* (i.e. without any compiler optimizations to avoid redundant work).
- The tightest asymptotic upper and lower bounds you can develop for $T(n)$.

(a)
```
int A(int n) {
      if (n == 0) return 1;
      else return A(n-1) * A(n-1);
}
```

Solution: Let $T(n)$ be the asymptotic time for the instance of size n. Then,

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

Using the method of annihilators, or substitution recognizing this as a constant coefficient difference equation, we look for solutions of the form z^n for some root z. We find that $z = 2$, so the asymptotic upper bound is

$$T(n) \in O(2^n)$$

.

We see that this is also an asymptotic lower bound, since there is no smaller number of operations to be done. Hence, $T(n) \in \Theta(2^n)$.

(b)
```
int B(int n) {
     if (n == 0) return 1;
     if (B(n/2) >= 10)
          return B(n/2) + 10;
     else
          return 10;
}
```

Solution: Note that $B(0) = 1$, and $B(1) = 10$. For $n > 1$, we always have $B(n/2) >= 10$, so we have to follow the top branch of the if statement.

Thus, we have the recursion:

$$T(n) = 2 * T(n/2) + c$$

for some constant c. The 2 happens because we have to evaluate $B(n/2)$ once for the if condition, and another time for the return.

This fits the Master theorem model, we have that the solution is $T(n) \in \Theta(n)$.

(c)
```
int C(int n) {
     if (n == 0) return 1;
     if (n== 1) return 3;
     return C(n-1) + C(n-2)*C(n-2);
}
```

Solution: This code is recursive. A recursion for the asymptotic run time is

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

This is a constant coefficient difference equation, so we look for solutions of the form z^n for some roots z. Substituting into the equation, we find the roots must satisfy $z^2 - z - 2 = 0$, so the possible vales are $z = 2, z = -1$. The root with largest magnitude is 2, so $T(n) \in \Theta(2^n)$.

3. (20 pts) You are given n nuts and n bolts, such that one and only one nut fits each bolt. Your only means of comparing these nuts and bolts is with a function TEST(x, y), where x is a nut and y is a bolt. The function returns +1 if the nut is too big, 0 if the nut fits, and -1 if the nut is too small.

   Design and analyze an algorithm for sorting the nuts and bolts from smallest to largest using the TEST function, such that the worst case performance of the algorithm has asymptotic complexity $O(n^2)$.

   Pseudo-code.

   Solution:

```
Procedure quicknutboltsort(nut_array,bolt_array,1,n):

Sorted_nut[1:n] = -1;
Sorted_bolt[1:n] = -1;
For bolt i = 1 to n:
     smaller = 0;
     fit_k = -1;
     For nut k = 1 to n:
          ans = Test(k,i)
          if ans < 0,
               smaller++;
          else if ans == 0,
               fit_k = k
     Sorted_nut[smaller] = fit_k;
For nut k = 1 to n:
     smaller = 0;
     fit_i = -1;
     For bolt i = 1 to n:
          ans = Test(k,i)
          if ans < 0,
               smaller++;
          else if ans == 0,
               fit_i = i
     Sorted_bolt[smaller] = fit_i;
```

   Pseudo code counts the number for a give bolt the number of smaller nuts to determine the sorted position for bolts and then repeats swapping nuts for bolts.

   Of course the average case performance of this is also $O(n^2)$.

   You can also do a version of quick-sort, by picking a bolt and pivoting the nut array, and similarly picking a nut and pivoting the bolt array.

   Solution:

```
nut_array[1:n]
bolt_array[1:n]
new_nut_array[1:n] = -1;
new_bolt_array[1:n]=-1;
```

```
Random pick a bolt i in  1 to n
position = 1
lastposition = n
best_fit = 0;
For k in 1 to n,
     if Test(k,i) < 0,
          new_nut_array[position]=nut_array[k];
          position ++;
     else if Test(k,i) == 0,
          best_fit = nut_array[k];
     else
          new_nut_array[lastposition] = nut_array[k];
new_nut_array[position] = best_fit;

// Now pivot the bolts against the best_fit nut:
position = 1
lastposition = n
k = best_fit;
best_fit = 0;
For i in 1 to n,
     if Test(k,i) < 0,
          new_bolt_array[position]=bolt_array[k];
          position ++;
     else if Test(k,i) == 0,
          best_fit = bolt_array[k];
     else
          new_bolt_array[lastposition] = bolt_array[k];
bolt_nut_array[position] = best_fit;

// now recur:
quicknutboltsort(new_nut_array, new_bolt_array, 1, position-1);
quicknutboltsort(new_nut_array,new_bolt_array,position+1,n);
```

This use a bolt as a pivot for quick-sort of nuts and the corresponding nut as the pivot for bolts. An average case of $O(nlog(n))$ but worst case, this is also $O(n^2)$.

4. (40 pts) **Binary search** of a large sorted array is a classic devide and conquer algorithms. Given a value called the `key` you search for a match in an array `int a[N]` of N objects by searching sub-arrays iteratively. Starting with `left = 0` and `right = N-1` the array is divides at the middle `m = (right + left)/2`. The routine, `int findBisection(int key, int *a, int N)` returns either the index position of a match or failure, by returning `m = -1`. First write a function for bisection search. The worst case is `O(log N)` of course. Next write a second function, `int findDictionary(int key, int *a, int N)` to find the `key` faster, using what is called, **Dictionary search**. This is based on the assumption of an almost uniform distribution of number of in the range of `min = a[0]` and `max = a[N-1]`. Dictionary search makes a better educated search for the value of `key` in the interval between `a[left]` and `a[right]` using the fraction change of the value, $0 \leq x \leq 1$:

```
x = double(key - a[left])/(double(a[right]) - a[left]);
```

to estimate the new index,

```
m =  int(left + x * (right - left)); // bisection uses x = 1/2
```

Write the function `int findDictionary(int key, (int *a, int N)` for this. For a uniform sequence of numbers this is with **average** performance: `(log(log( N))`, which is much faster than `log(N)` bisection algorithm. Graph the data to see this scaling behavior. – much more fun that a bunch of numbers! You may use any graphing routine you like but in class we will discuss how to use `gnuplot` which is a basic unix tool. **(See gnuplot instructions on GitHub at** `CourseInformation/Plotting_and_Fitting` **)**,

Implement your algorithm as a C/C++ functions. On the class GitHub there is the main file that reads input and writes output the result. You only write the required functions. Do not make any changes to the `infield` reading format or the `outfile` writing format in `main()` . Place your final code in directory HW1. The grader will copy this and run* the `makeFind` to verify that the code is correct. There are 3 input files `Sorted100.txt , Sorted100K.txt, Sorted1M.txt` for $N = 10^2, 10^5, 10^6$ respectively. You should report on the following:

```
(1)The code should run for each file on the command line.
(2)The code should print to a file a table of the 100 random  keys,
execution time and the number of bisections.
(3)And another simliar file for dictionary with "bisections" the number of
dictionary sections
```

(In the future we use this basic procedure to some data analysis by running it for many values of N and many values of *keys* and to plot it to see as best we can the claimed scaling of `log(N)` and `(log(log( N))` respectively. This require more instruction in how to plot data and do curve fitting and even error analysis. These skills will be useful later in the class project phase.)