EC 504 – Fall 2019 – Homework 1

Due Thursday, Sept. 19, 2019 in the beginning of class. Coding problems submitted in the directory `/projectnb/alg504/yourname/HW1` on your SCC account by Friday Sept 20, 11:59PM. Reading Assigned in CLRS Chapters 2, 3 and 4.

1. (20 pts) 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.

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

Extra Credit (5 pts): 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. If you are eager do it more generally case for $T(n) = aT(n/b) + n^k$ with $k = \gamma$ and $b^\gamma = a$ using the trial solution $T(n) = c_1 n^\gamma + c_2 n^\gamma log_2(n)$ to determine new values of c_1, c_2.

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 funcion ($f(n) = e^{\ln(f(n))}$) 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 oscilates between $\Theta(1)$ and $\Theta(n^2)$ getting arbitrarily close both even at interger values. Therefore strickly speaking the best bound is $n^{1+\cos n} \in O(n^2)$ but $n^\alpha \in O(n^{1+\cos n})$ implies $\alpha \leq 0$ by the definition in CLRS of "Big Oh".

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 <= 1) return 1;
     int prod = 0;
     for (int ii = 0; ii < n; ii++)
         prod *= C((int) sqrt(n));
```

```
    return prod;
}
```

Solution: Given n, the code will call n versions of the function of size $\sqrt{n}$, and perform a linear cn amount of operations (multiplications, square roots) to combine the results.

The expression for run time is

$$T(n) = nT(\sqrt{n}) + cn$$

To solve this, draw a recursion tree starting from a node of size n, with n children of size $\sqrt{n}$, each of which has $n^{1/2}$ children of size $n^{1/4}$, etc. Note that the work in each node is linear in the size of the node. The number of children from level n is n, from level $n^{1/2}$ is $n * n^{1/2}$, at level $n^{1/4}$ is $n^{1+1/2+1/4}$, ...

Thus, the total amount of work at each level is equal to the number of nodes at that level times the amount of work at the node, which is linear in the size of the node. There is one node of size n, n nodes of size $n^{1/2}$, $n^{3/2}$ nodes of size $n^{1/4}$, ... This yields the following expression, summing across the different levels:

$$T(n) = cn + cn^{3/2} + cn^{7/4} + \ldots + cn^{\frac{2^k-1}{2^{k-1}}} = cn^2(\frac{1}{n} + \frac{1}{n^{1/2}} + \ldots + \frac{1}{n^{1/2^k}})$$

where $k \approx \log_2 n$ (the number of square roots required to get $n^{1/k} < 2$.)

Note that there are $\log_2(n)$ terms in the summation. Each of the terms in parentheses is less than 1, so we know that this is $O(n^2 \log_2(n))$ as an upper bound. Can we show that this is also $\Theta(n^2 \log_2(n))$? Consider that half the terms are greater than $\frac{1}{n^{\frac{1}{\sqrt{n}}}}$, and that

$$\lim_{n\to\infty} \frac{1}{n^{\frac{1}{\sqrt{n}}}} = 1$$

because, by taking logarithms, we have

$$\lim_{n\to\infty} \log \frac{1}{n^{\frac{1}{\sqrt{n}}}} = -\lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = 0$$

This shows that there is a lower bound on the complexity with $\frac{\log_2(n)}{2}$ terms in the summation close to 1, so the complexity is $\Theta(n^2 \log n)$.

(d)
```
int D(int n) {
    if (n == 0) return 1;
    if (n== 1) return 3;
    return D(n-1) + D(n-2)*D(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 $T(n) \in \Theta(2^n)$.

(e)
```
int E(int n) {
    if (n<= 1) return 1;
```

```
        int count = 3;
        int tmp = E(n/2);
        for (int k = 0; k < n; k++)
             for (int m = 1; m <n; m*=2)
                  if (tmp < exp(k+m))
                       count++;
        return E(n/2)*(count%2);
}
```

Solution: As before, let's derive a recursion: There are two calls to $E(n/2)$, and there are two loops, that do $O(n \log_2(n))$ operations. Hence, the recursion is

$$T(n) = 2T(n/2) + cn\log(n)$$

Note that this does not fit any of the 3 cases of the Master's theorem! We can try enumerating the recursion tree to see that, at each level, we do the following amount of work: $cn\log(n/k)$, where k counts the level from 1 (the call of size n) to $\log_2(n)$, the calls to the function of size 1. Hence, the total work is

$$\sum_{k=1}^{\log_2 n} cn(\log(n) - \log(k)) \in O(n\log^2(n))$$

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 swapoing nuts for bolts.

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

You can also do a version of quicksort, 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 quicksort 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. (20 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, $0 \le x \le 1$:

   ```
   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. In class we will discuss how to use `gnuplot` to show this behavior graphically – much more fun that a bunch of numbers!

   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 file simliar file for dictionary with "bisections" the number of
   dictionary sections
   ```

   (In the future we use this a code 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.)

5. (20 points) Average performance of $O(N^2)$ search algorithm. Standard $O(N^2)$ search algorithms involve local (nearest neighbor) exchanges of element of the given list

$$A_{list} = a[0], a[1], a[2], a[3], \cdots, a[N-1] \tag{1}$$

The result is that you need to slide (or push) each of N elements into its right position exchanging it (or `swapping it`) with all that are out of order. It is easy to imagine (and proven in class) that on average each of N elements need to exchange on 1/2 if the others so the on average the algorithm would have

$$\text{Number of Exchanges} \quad = \quad \frac{N(N-1)}{4} \tag{2}$$

This exercise to *numerically prove* by counting the exchanges averaged over a large set of permutation of the list few values of N. To do this you need to set up a list of N objects put it into a random order and count the exchanges. The code to do this is on GitHub. Place your final code in directory HW1 so that it is compiled* with `makeRanSort` .

```
(1) For one value of N make a plot of the number of swaps versus at least
100 permuations.
(2) Compare the average with the theoretical value N (N -1)/4
```

Extra Credit: This exercise illustrate two significant issues. #1. Average performance is rather than worse case if often more relevant in real applications: When and Why? #2. Counting operations is often a more interesting metric for an algorithm than time it: When and Why? Pass in with the written part one sentence comments on these issues.

*Note the usual command `make -k` will run the default `Makefile`. However in these code exercises I have two separate **named makefiles** that need to be run using $-f$ flags:

```
make -f makeFind
and
make -f makeRanSort
```

respectively.