EC 504 – Fall 2021 – Homework 1

Due Friday Sept. 24, 2021 by 11:59PM as a pdf for the written part and coding problem Submit both in your directory `/projectnb/alg504/yourname/HW1`

Start reading Chapters 1, 2, 3 and 4 in CLRS . They give a very readable introduction to Algorithms.. Also glance at Appendix A for math tricks which we will use from time to time.

1. (25 pts)

   (a) In CRLS do Exercise 3.1-2 on page 52, Problem 3-2 on page 61, Problem 3.2-7 on page 60.

   Solution: Problem 3.1-3:

   if $fn) = (n+a)^b$ there are many ways to show it is $\Theta(n^a)$. For example compute the limit of ratio $\lim_{n\to\infty} f(x)/n^b = 1$ using the binomial theorem (e.g. a Taylor series)

$$f(x)/n^b = (1+a/n)^b = 1 + b(a/n) + \frac{b(b-1)}{2!}(a/n)^2 + \cdots$$

   Problem 3-2:

A	B	O	o	Ω	ω	Θ	Comment
$ln^k n$	n^ϵ	yes	yes	no	no	no	logs never beat powers
n^k	c^n	yes	yes	no	no	no	powers never beat exponents
$\sqrt{n}$	$n^{\sin(n)}$	no	no	no	no	no	$\sin(n)$ fills set $\{\sin(n) \in (-1,1)\}$
2^n	$2^{n/2}$	no	no	yes	yes	no	Since $2^{n/2} = (\sqrt{2})^n$
$n^{\ln c}$	$c^{\ln n}$	yes	no	yes	no	yes	Since $x^{\log(y)} = y^{\log(x)}$ is an identity
$\lg(n!)$	$log(n^n)$	yes	no	yes	no	yes	$\log(a^b) = b\log(a)$ and Eq. 3.20

   Problem 3.2-7:

   Note $\phi = (1+\sqrt{5})/2$ and $\hat{\phi} = (1-\sqrt{5})/2 \equiv -1/\phi$ This induction needs two starting value.

$$F_0 = (\phi^0 - \hat{\phi}^0)/\sqrt{5} = 1 \quad , \quad F_0 = [(1+\sqrt{5})/2 - (1-\sqrt{5})/2]/\sqrt{5} = 1$$

   for the induction to F_{k+1} from F_k and F_k

$$\begin{aligned} F_{k+1} &= F_k + F_{k-1} = (\phi^k - \hat{\phi}^k)/\sqrt{5} + (\phi^{k-1} - \hat{\phi}^{k-1})/\sqrt{5} \\ &= [(\phi^k + \phi^{k-1})) - (\hat{\phi}^k + \hat{\phi}^{k-1}))]/\sqrt{5} = [\phi^{k-1}(\phi+1) - \hat{\phi}^{k-1}(\hat{\phi}+1))]/\sqrt{5} \\ &= [\phi^{k+1} - \hat{\phi}^{k+1}]/\sqrt{5} \quad , \quad Q.E.D \end{aligned}$$

   where the last line uses the useful identity $\phi^2 = \phi + 1$ for **both** roots.

(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.

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

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 $\alpha \leq 0$ by the definition in CLRS of "Big Oh".

(c) Substitute

$$T(n) = c_1 n + c_2\ nlog_2(n)$$

into $T(n) = 2T(n/2)+n$ to find the values of c_1, c_2 to determine the exact solution.

(d) **Extra Credit(10 pt):** Generalize this to the 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: On both (c) and (d) you are given a `guess` with unknown constants c_1, c_2. To see if is ok see if substituting the guess is possible to to match exactly the e RHS (right hand side) and the LHS (left hand side).

Part (c):
Substitution gives:

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

Now to try set `RHS = LHS` you match both the n and the $n log_2(n)$ term

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

Indeed the match works for $c_2 = 1$ and any c_1. This is to be expected since the recursion need to start with a value $T(1)$ to fix $c_1 = T(1)$.

For part (d):
With $\gamma \neq k$ the general solution works matching the LHS vs RHS

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

gives to two conditions:

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

satisfied for any c_1 and fixed $c_2 = 1/(1 - a/b^k)$.

But it fails for $k = \gamma$ since now $c_2 = \infty$
Starting over we try a power plus log enhancement with the matching condition:

$$\begin{aligned} c_1 n^\gamma + c_2 n^\gamma log_2(n) &= a[c_1 (n/b)^\gamma + c_2 (n/b)^\gamma \log_2(n/b)] + n^\gamma \\ &= a[c_1 (n/b)^\gamma + c_2 (n/b)^\gamma (\log_2(n) - \log_2(b))] + n^\gamma \end{aligned}$$

Again using $a/b^\gamma = 1$, not that without a log term $(c-2)$ the power fails $c_1 \neq c_1 + 1$ but the log term saves this with the conditions

$$c_1 = c_1 \quad \text{and} \quad 1 - c_2 \log_2(b) = 0$$

so again c_1 is not determined but $c_2 = 1/log_2(b)$ is. This is general observation. The larger term is determined as n goes to infinity and the smaller needs a base number of $T(1) = c_1$ to determine c_1.

2. (25 pts) Here is a classical problem that is often part of interviews. 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 $O(n^2)$. This is quite easy. Note that given a bolt you can find the number of nuts that are smaller and the number that is larger and which matches. We will show later that this partitioning into small and large sets can be a modified the recursive quick sort algorithm to get a $O(nlog(n))$ solution to this problem!

   "Pseudo-code" – any clear and concise description of the algorithm even with a drawing if you want – *a picture is worth a thousand words?*

   Solution:

```
Procedure nutboltsort(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:

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

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 $O(n^2)$.

3. (50 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.

   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. 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.
   ```

   Next for extra credit do your best to modify the code. (This modified code does not have to be turned in but if you do turn it in give it a different name. In class let's discuss together what how you did this. This code should be used to graph the performance as function of N to see this scaling behavior. – much more fun that a bunch of numbers! To extend the range of sizes there is a code `makeSortedList.cpp` that can generate more sorted input lists any size N. For example you might 6 sizes: $N = 10^2, 10^4, 10^5, 10^6, 10^7, 10^8$. Then collect on Time and OpCount and graph them as function of N. 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** `GeneralComputingInfo/Plotting_and_Fitting` **)**, Any hack is ok to get some rough graphs. You will find there is an art to measuring these slow growth rates! If you want to automate the size you can integrate `makeSortedList.cpp` as subroutine into your code.

(In the future exercises, we will develop thes basic procedure to do performance analysis by running for many values of N, averaging over many cases (e.g. *keys*) with to get mean values for Time and OpCount even including error bars and fitting routines to determine average scaling. This is an important art of engineering analysis. This require more instruction and experimentation with how to plot data and do curve fitting and even error analysis. These skills may also be useful later in the class project phase depending on what you topic you choose.)