CS5800: Algorithms Spring 2018 Assignment 1

Saptaparna Das

January 18, 2018

1. $\sqrt{n}, n\sqrt{\log n}, 2^{\sqrt{\log n}}, (\log n)^2$

- for$\sqrt{n}$ vs $2^{\sqrt{\log n}}$:
  We can express $\sqrt{n}$ as $\sqrt{n}^{\log_2 2}$, which is equivalent to $2^{\log_2 \sqrt{n}}$.
  So, applying limit on both sides, $\lim_{n\to\infty} \frac{\sqrt{n}}{2^{\sqrt{\log n}}} = \lim_{n\to\infty} \frac{2^{\log_2 \sqrt{n}}}{2^{\sqrt{\log n}}} = \lim_{n\to\infty} 2^{(\log\sqrt{n} - \sqrt{\log n})}$.(Dropping the base, since it doesn't matter)
  Now,substituting $\log n$ by u, the expression can be written as $\lim_{u\to\infty} 2^{(\frac{u}{2}-\sqrt{u})}$. Now, $\frac{u}{2} - \sqrt{u} = \sqrt{u}(\frac{\sqrt{u}}{2} - 1) = \infty$
  Hence, $\sqrt{n} >= 2^{\sqrt{\log n}}$

- for$\sqrt{n}$ vs $n\sqrt{\log n}$:
  Applying limit, $\lim_{n\to\infty} \frac{\sqrt{n}}{n\sqrt{\log n}} = \lim_{n\to\infty} \frac{1}{n\sqrt{\log n}} = 0$
  So, $n\sqrt{\log n} >= \sqrt{n}$
  Example: say $n = 2^{16}$, $\sqrt{n} = 2^8$ and $n\sqrt{\log n} = 2^{18}$

- for$(\log n)^2$ vs $2^{\sqrt{\log n}}$:
  Say, $(\log n)^2 = a$, then a can be written as $a^{\log_2 2}$, which is equivalent to $2^{\log_2 a}$.
  Now, $\log{(\log n)^2} = 2\log\log n$. So, $(\log n)^2$=$2^{2\log\log n}$
  So, applying limit on both sides of original problem, $\lim_{\ n} \to \infty \frac{2^{2\log\log n}}{2^{\sqrt{\log n}}}$. Putting $u = \log n$, the expression becomes, $\lim_{u\to\infty} 2^{(2\log u - \sqrt{u})}$. Cleary $\log u$ is smaller than $\sqrt{u}$. (Since, $\lim_{u\to\infty} \frac{\log u}{\sqrt{u}} = \lim_{u\to\infty} \frac{2}{\sqrt{u}} = 0$). So the expression $(2\log u - \sqrt{u})$ evaluates to $2^{-\infty}$=0.
  Hence, $(\log n)^2 <= 2^{\sqrt{\log n}}$

Compiling all the results, we can arrange the following functions in order from the slowest growing function to fastest growing function as:

$(\log n)^2$,$2^{\sqrt{\log n}}$,$\sqrt{n}$,$n\sqrt{\log n}$

2.(a) if $f(n) = \Omega(h(n))$ and $g(n) = O(h(n))$, then $f(n) = \Omega(g(n))$

$f(n) = \Omega(h(n))$ implies that for a positive constant c_0, $f(n) >= c_0 h(n)$. We can also say, $-f(n) <= -c_0 h(n)$
$g(n) = O(h(n))$, implies that for a positive constant c_1, $g(n) <= c_1 h(n)$.

Hence, we can deduce, $-\frac{f(n)}{g(n)} <= -\frac{c_0}{c_1}$ or $\frac{f(n)}{g(n)} >= \frac{c_0}{c_1}$
So, we can have a positive constant k, for which $f(n) >= kg(n)$ or $f(n) = \Omega(g(n))$

Alternatively, $f(n) = \Omega(h(n))$ and $g(n) = O(h(n))$ implies $h(n) = \Omega(g(n))$ [Transpose symmetry]
so, we can $f(n) = \Omega(g(n))$[Transitivity]
Hence, this statement is **True**

2(b) if $f(n) = O(g(n))$, then $3^{f(n)}$ is $O(3^{g(n)})$.
Example, say $f(n) = 3n$, We know, then $f(n) = O(n)$. So, $3^{f(n)} = 3^{3n}$, which should be $O(3^n)$, if the above statement is true. But 3^{3n} is not same as 3^n. Hence, this statement is **False**.

3(a) The problem says, we have to compute a childs percentile BMI from last years data. So, input of the function will be a list of BMIs of all kids last year and BMI of current kid. Since, the percentile is calculated as the ratio of the number of 5-year old patients last year whose BMI is lesser than it, and the total number of patients in last years record, we may assume that last years record is maintained in nondescending order, we need to find out all elements which are lesser than current element. We can use binary search technique here, to get the exact index where current element can be inserted, so that the whole list remains sorted.The time complexity of the search would be $O(\log n)$, because we divide the array in two parts and recursively call the same method.

3(b) Pseudo code:

```
//Assume we have a sorted (in non−descending order) array of BMIs of last year
//The data is sorted while updating records, at the end of every year
//Since the data is large, standard merge sort may be used.
//The array is zero indexed.

calculateBMIPercentile(bmiList[o...n−1], currBmi){

        size=bmiList.length
        //Initially calls binarySearch function with left index as 0 and
        //right index as n−1
        result = binarySearch(bmiList,0,n−1,currBmi)

        //"result" will have the index before which all elements
        // will be less than currBmi

        print ((result+1)/size)
        }

binarySearch(bmiList[0...n−1], left, right, currBmi){
        if (right>=left){
                //Assume mid takes floor value of the expression
                mid = (left + right)/2

                // If the element matches middle element
                if (bmiList[mid] == currBmi)
                        return mid

                // If element is smaller than middle element,
                // then should be present in left subarray
                if (bmiList[mid] > x)
                        return binarySearch(bmiList, left, mid−1, currBmi)

                // Else the element should be present in right
                // subarray
                return binarySearch(bmiList, mid+1, right, currBmi)
                }

                // return left index when element is not found in list
                return left
        }
```

3(c) Proof of correctness:

The invarient here, is the difference between right and left index, which becomes smaller in every call. To prove correctness, let's assume that binarySearch works correctly for inputs where righ-left+1 = n. If we can prove that this is true for all n, then we know that binarySearch will work for all arguments.

Base Case: when n=1, i.e. there is only one element in array, we know left=right=mid. Since, our code returns either mid or left(when element not present) index, therefore the function will return 0 or 1. if currBmi is same as the element present in array, then it will return mid=0. if it's greater than element present, then it will call binarySearch(bmiList,mid+1,right,currBmi), which will return 1. Similarly if it is less than element present, then it will call binarySearch(bmiList,left,mid-1,currBmi) which returns 0. Hence, it works correcly for single element.
Inductive step: We assume the binarySearch method works for all the values of inputs from 0 to k (where $right - left + 1 = k$). We need to prove it works for k+1. There are three cases:
1. currBmi=bmiList[mid]: Already proven, it works correctly.
2. $currBmi < bmiList[mid]$: Since the array is sorted in non-desecending order, that currBmi index must be between bmiList[left] and bmiList[mid-1].So, number of elements in array = mid - 1 - left + 1 = $\left\lfloor \frac{(left+right)}{2} \right\rfloor - left$.
If left + right is odd, then $n = (left + right - 1)/2 - left = (right - left - 1)/2$ which is smaller than right-left or

k.
If left + right is even, then $n = (left + right)/2 - left = (rightleft)/2$, which is also smaller than right - left because right - left or k.

3. $currBmi > bmiList[mid]$: Similarly,$n = right - (mid + 1) + 1 = right - mid = right - \lfloor(left + right)/2\rfloor$.
if right+left is even, then it is (right-left)/2 , which is less than right-left(k). If right+left is odd, this is $right - (left + right - 1)/2 = (right - left + 1)/2$ which is again less than right-left or k.

So, we can say binarySearch works for all inputs.

Efiiciency analysis: Say T(n) is the time taken by the program. So T(n) can be written as, $T(n) = T(n/2) + c$, where c is a constant time for doing operations independent of n.
T(n/2) is written because the binarySearch function calls itself recursively and every time the size of the array becomes half.
So, by substitution, we can write, $T(n) = [T(n/4) + c] + c = [T(n/2^2) + 2.c]$. We can observe a pattern here and we can deduce $T(n) = [T(n/2^i) + i.c]$
If n is perfect power of 2, the $n = 2^i$ or $i = \log_2 n$. if its not perfect square, it will differ by some positive constant
Hence, the total time becomes $T(1) + c.\log_2 n$=$c_1 + c.\log_2 n$, which is $O(\log n)$