CS 600 Advanced Algorithms

Homework 1

September 14, 2018

Contents

1 R-1.7 Order list of functions by the $\mathcal{O}$ notation.

The ascending order is as follows. Functions that are Θ of each other are grouped and underlined.

$\frac{1}{n}$, 2^{100}, $\log\log n$, $(\log n)^{\frac{1}{2}}$, $\log^2 n$, $n^{0.01}$, $\underline{\lceil n^{\frac{1}{2}} \rceil\ ; 3n^{0.5}}$, $\underline{2^{\log n}\ ; 5n}$, $\underline{n\log_4 n\ ; 6n\log n}$, $\lfloor 2n\log^2 n \rfloor$, $4n^{\frac{3}{2}}$, $4^{\log n}$, $n^2\log n$, n^3, 2^n, 4^n, 2^{2^n}

2 R-1.9 Complexity of the FIND2D algorithm

The worst-cast runtime complexity of FIND2D is $\mathcal{O}(n^2)$. This is seen by examining the worst case where the element x is the very last item in the $n \times n$ array to be examined. In this case, FIND2D calls the algorithm ARRAYFIND n times. ARRAYFIND will then have to search all n elements for each call until the final where when x is found. Therefore n comparisons are done for each ARRAYFIND call. This means we have $n \times n$ operations – $\mathcal{O}(n^2)$ running time. This is not a linear time algorithm, it is quadratic.

3 R-1.22 Show that $n \in o(n\log n)$

To show that $n \in o(n\log n)$ we use the definition of little o.

$$n \in o(n\log n) \implies \forall c > 0, \exists n_0 > 0 \mid \forall n \geq n_0, n < cn\log n$$

That is, $\log n > \frac{1}{c}$ for $n \geq n_0$. This is true when $n > 2^{\frac{1}{c}}$. We simply choose

$$n_0 = \lceil (1 + 2^{\frac{1}{c}}) \rceil$$

and we are done.

4 R-1.23 Show that $n^2 \in \omega(n)$

We need to show that n^2 is in $\omega(n)$. Alternatively, we can show that n is in $o(n^2)$. That is, for any constant $c > 0$, there is a constant $n_0 > 0$ such that $n < cn^2$, which is true if $n > \frac{1}{c}$. Therefore we choose $n_0 = \lceil 1 + \frac{1}{c} \rceil$ and we are done.

5 R-1.24 Show that $n^3\log n \in \Omega(n^3)$

By the definition of Ω, we need to find a $c > 0$ and $c \in \mathbb{R}$ and a $n_0 \geq 1$ and $n \in \mathbb{Z}$ such that $n^3\log n \geq cn^3$ for all $n \geq n_0$. Choosing $c = 1$ and $n_0 = 2$ satisfies the inequality since $\log n \geq 1$ in this range.

6 R-1.32 Upper bound of a probability of choosing from a set of n balls

Using the Chernoff bound, we have

$$\mu = E(X) = n\frac{1}{\sqrt{n}} = \sqrt{n}$$

Then for $\delta = 2$, we have

$$Pr(X > (1+\delta)\mu) < \Big[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\Big]^{\mu} \implies Pr(X > 3\sqrt{n}) < \Big[\frac{e^2}{3^3}\Big]^{\sqrt{n}}$$

7 C-1.4 Running time of counting from 1 to n in binary

We have the condition that the time needed to add 1 to the current number, i, is proportional to the number of bits in the binary expansion of i that must change in going from i to $i+1$.

Let's enumerate the binary expansions of the numbers from 1 to 8.

1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0

Observe that the rightmost bit changes with every number; the second bit changes every other time; the third bit changes 2 times, and the last bit changes 1 time.

Without loss of generaliity, assume n is a power of 2, that is $2^k = n$ for some k. Then in counting from 1 to n, the first bit changes n times, the second bit $n/2$ times, the third bit $n/4$ times and so on.

Give the work required to change a bit is w, the total work is

$$w\sum_{i=0}^{k}\frac{n}{2^i} = wn\sum_{i=0}^{k}\frac{1}{2^i} < 2wn \in \mathcal{O}(n)$$

Therefore the algorithm for counting from 1 to n given the constraints of the question is in the order of $\mathcal{O}(n)$.

8 C-1.7 Show by induction that $T(n) = 2^n$

The recurrence equation is

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2T(n-1) & \text{otherwise} \end{cases}$$

Consider the base case when $n = 1$. We have

$$T(1) = 2T(0) = 2$$

This holds. We now assume that the equation holds for $n - 1$ and show that it also holds for n.

$$T(n) = 2T(n-1) = 2 \times 2^{n-1} = 2^n$$

9 C-1.22 Show that $\sum_{i=1}^{n} \lceil \log_2(\frac{n}{i}) \rceil \in \mathcal{O}(n)$

Consider the following inequality

$$\sum_{i=1}^{n} \lceil \log_2(\frac{n}{i}) \rceil \leq \sum_{i=1}^{n} (1 + \log_2(\frac{n}{i})) = \sum_{i=1}^{n} (1 + \log_2 n - \log_2 i).$$

We can further see that

$$\sum_{i=1}^{n} (1 + \log_2 n - \log_2 i) < \int_{x=0}^{x=n} (1 + \log_2 n - \log_2 x)\, \mathrm{d}x = n\Big(1 + \frac{1}{\log(2)}\Big)$$

Since $n\Big(1 + \frac{1}{\log(2)}\Big) \in \mathcal{O}(n)$, we are done.

10 C-1.30 Complexity of an extendable table implementation

We will use amortization to compute the total cost of this extendable table implementation.

As an overflow occurs, the size of the array is expanded from N to $N + \lceil \sqrt{N} \rceil$. We assign one cyber dollar for the cost of inserting an element the first time in to the array. So the total cost of inserting n elements is n.

We need to distribute the cost of copying the array of size N to $N + \sqrt{N}$ in the next overflow over the new insertions from $N + 1$ to $N + \sqrt{N}$. So each such insertion will be charged an additional $\frac{N+\sqrt{N}}{\sqrt{N}} = 1 + \sqrt{N}$ cyber dollars.

Therefore, each insertion into the array is charged an extra $1 + \sqrt{N}$ for future copying, for a total of

$$\sum_{N=1}^{n} 1 + 1 + \sqrt{N} = \sum_{N=1}^{n} 2 + \sqrt{N} = 2n + \sum_{N=1}^{n} \sqrt{N}$$

There is no closed formula for this summation, but it is known that the sum is no more than

$$\frac{2}{3}n^{3/2} + \frac{1}{2}n^{1/2} - \frac{1}{6}$$

but no less than

$$\frac{2}{3}n^{3/2} + \frac{1}{2}n^{1/2} + \frac{1}{3} - \frac{\sqrt{2}}{2}$$

So the total cost of the array operation is in $\Theta(n^{3/2})$.

There is another method for approximating the sum above. The sum is bounded by $\int_0^{n-1} \sqrt{x}\,\mathrm{d}x$ and the same integral from 1 to n. These are the sum of the heights under the curve $f(x) = \sqrt{x}$ and above the curve $f(x) = \sqrt{x}$. Since, $\int \sqrt{x}\,\mathrm{d}x = x^{\frac{3}{2}}$, we are done.

11 A-1.8 Algorithm for reversing an array

We can use two pointers – one pointing to the start of the array and the other to the end of the array to perform the reversal. Let's call these *start* and *end*. While *start* is less than *end*, we swap the elements they point to, *increment* start, and *decrement* end.

Algorithm Reverse array A

Input: array A of n elements
Output: reversed array A

$start \leftarrow 0$
$end \leftarrow n - 1$
while $start < end$ **do**
    $temp \leftarrow A[start]$ ▷ Swap the elements
    $A[start] \leftarrow A[end]$
    $A[end] \leftarrow temp$
    $start \leftarrow start + 1$
    $end \leftarrow end + 1$

This algorithm is fairly standard, and has a worst-case runtime complexity of $\mathcal{O}(n)$. The space taken (excluding the array) is constant.

12 A-1.15 Algorithm for finding shortest subarray with k 1's in an array of n bits

We initialize two pointers, *lower* and *upper*, and scan A. We record the first occurence of a 1 in *lower* and scan from this position on until we have found k 1's and set *upper* to this index. We also have to record the positions of *lower* and *upper* in order to compare them later with other possible subarrays containing k 1's.

The goal now is to keep scanning further right by incrementing *lower* and *upper* while maintaining the condition that $A[lower : upper]$ contains k 1's. Each time we find such a subarray we compare it to the length of the previous shortest subarray and make changes to our information if necessary. Once we are done scanning the entire array, we return the bounds of the shortest subarray containing k 1's.

We are essentially using a sliding window. Notice that at most we scan an element twice. These scans take place after we find the first subarray containing k 1's. The runtime complexity is in the order of $\mathcal{O}(n)$. The space taken (excluding the array) is constant. In the pseudocode shown below, the variables *start* and *end* are used to denote the bounds of the shortest subarray.

Algorithm Find shortest subarray in A containing k 1's

Input: An array A of n bits, and an integer $k > 0$
Output: Bounds of the shortest subarray in A containing k 1's

```
for i ← 0 to n do
    if A[i] = 1 then
        lower ← i
        count ← i
        for j ← i + 1 to n do
            if A[j] = 1 then
                count ← count + 1
            if count = k then
                upper ← j
                start ← lower; end ← upper          ▷ Record current shortest subarray
                break
        break

while upper < n do

    lower ← lower + 1
    while A[lower] = 0 do                           ▷ Move lower to the next 1
        lower ← lower + 1

    upper ← upper + 1
    while A[upper] = 0 do                           ▷ Move upper to the next 1
        upper ← upper + 1

    if (upper − lower) < (end − start) then
        start ← lower; end ← upper
return (start, end)
```