# CS 600 HW 1
SHUCHI PARAGBHAI MEHTA
CWID: 20009083

- **1.6.7**
  **Order the list of functions by the big-Oh notation. Group together (for example, by underlining) those functions that are big-Theta of one another.**

  The ascending order is as follows,

  $\frac{1}{n}$, $2^{100}$, $\log \log n$, $\sqrt{\log n}$, $\log^2 n$, $n^{0.01}$, $\lceil \sqrt{n} \rceil$, $3n^{0.5}$, $2^{\log n}$, $5n$, $n\, log_4 n$, $6n \log n$, $\lfloor 2n \log^2 n \rfloor$, $4n^{3/2}$, $4^{\log n}$, $n^2 \log n$, $n^3$, $2^n$, $4^n$, $2^{2^n}$

  Functions which are big-Theta of each other,
  $\lceil \sqrt{n} \rceil$ and $3n^{0.5}$
  $2^{log\, n}$ and $5n$
  $n\, log_4 n$ and $6n \log n$

- **1.6.9**
  **Bill has an algorithm, find2D, to find an element x in an $n \times n$ array A. The algorithm find2D iterates over the rows of A and calls the algorithm $arrayFind$, of Algorithm 1.3.2. on each one, until x is found, or it has searched all rows of A. What is the worst-case running time of find2D in terms of n? Is this a linear-time algorithm? Why or why not?**

  Find2D has an $O(n^2)$ worst-case runtime complexity.
  This is demonstrated by looking at the worst case, in which element x is the very last element to be reviewed in the $n \times n$ array. In this instance, $arrayFind$ is called n times by the algorithm $find2D$. After that, arrayFind will have to look through each of the n elements till the last place when x is found.

  As a result, each $arrayFind$ call results in n comparisons.

  This indicates that our running time is $O(n^2)$ for n × n operations. This is not a linear time algorithm, it is quadratic.

- **1.6.22**
  **Show that n is $o(n\, log\, n)$.**

  To show that $n \in o(n\, log\, n)$, we use the definition of little $o$.
  $n \in o(n\, log\, n) \Rightarrow \forall c > 0, \exists n_0 > 0 \mid \forall n \geq n_0, n < cn\, log\, n$
  That is, $log\, n > \frac{1}{c}\ for\ n \geq n_0$. This is true when $n > 2^{\frac{1}{c}}$.
  We simply choose $n_0 = \left\lceil (1 + 2^{\frac{1}{c}}) \right\rceil$
  And we are done.

- **1.6.23**
  **Show that $n^2$ is $\omega(n)$.**

  We need to prove that $n^2$ is in $\omega(n)$. An alternative is to show that $n$ is in $o(n^2)$. In other words, if $n > \frac{1}{c}$, then for any constant $c > 0$, there is a constant $n_0 > 0$ such that n < $cn^2$.
  As a result, we select $n_0 = \left\lceil (1 + \frac{1}{c}) \right\rceil$, and we are done.

- **1.6.24**
  **Show that $n^3 \log n$ is $\Omega(n^3)$.**

  We must locate a $c > 0$ and $c \in R$, a $n_0 \geq 1$ and $n \in Z$ such that $n^3 \log n \geq cn^3$ for all $n \geq n_0$ in accordance with the definition of $\Omega$ . Since $\log n \geq 1$, choosing $c = 1$ and $n_0 = 2$ satisfies the inequality.

- **1.6.32**
  **Suppose we have a set of $n$ balls, and we choose each one independently with probability $1/n^{\frac{1}{2}}$ to go into a basket. Derive an upper bound on the probability that there are more than $3n^{\frac{1}{2}}$ balls in the basket.**

  Using the Chernoff bound, we have $\mu = E(X) = n\frac{1}{\sqrt{n}} = \sqrt{n}$
  Then for $\delta = 2$, we have

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

- **1.6.36**
  **What is the total running time of counting from 1 to $n$ in binary if 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. 0001
  2. 0010
  3. 0011
  4. 0100
  5. 0101
  6. 0110
  7. 0111
  8. 1000

  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.

  Assume that n is a power of 2, which means that for some $k$, $2^k = n$, without losing generality. The first bit then changes $n$ times, the second bit $n/2$ times, the third bit $n/4$ times, and so on as you count from 1 $to\ n$.

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

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

  Given the constraints of the question, the algorithm for counting from 1 to n is therefore of the order of $O(n)$.

- **1.6.39**
  **Consider the following recurrence equation, defining a function $T(n)$:**

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

  **Show, by induction, that $T(n) = 2^n$.**

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

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

  Which holds.

  Now that we know the equation is true for n − 1, we can demonstrate that it is true for n as well.

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

- **1.6.52**
  **Show that the summation $\sum_{i=1}^{n} \lceil \log_2 i \rceil$ is $O(n \log n)$.**

  Consider the following,

  $$\log(1) + \log(2) + \ \dots + \log(n) \leq \log(n) + \log(n) + \ \dots + \log(n) \ = \ n * \log(n)$$

  We can further see that

  $$\log(1) + \ \dots + \log\left(\frac{n}{2}\right) + \ \dots + \log(n) \geq \log\left(\frac{n}{2}\right) + \ \dots + \log(n) \ \geq \log\left(\frac{n}{2}\right) + \ \dots + \log\left(\frac{n}{2}\right) = \ n/2 \ * \ log(n/2)$$

  Hence from above we can prove, $\sum_{i=1}^{n} \lceil \boldsymbol{log_2} \ \boldsymbol{i} \rceil$ **is** $\boldsymbol{O(n \log n)}$.

- **1.6.62**
  **Consider an implementation of the extendable table, but instead of copying the elements of the table into an array of double the size (that is, from $n\ to\ 2N$) when its capacity is reached, we copy the elements into an array with $\lceil \sqrt{N}\, \rceil$ additional cells, going from capacity $n\ to\ N + \lceil \sqrt{N}\, \rceil$. Show that performing a sequence of $n\ add$ operations (that is, insertions at the end) runs in $\Theta(n^{3/2})$ time in this case.**

The overall cost of implementing this expandable table will be determined using amortization. The size of the array increases from $N$ to $\mathrm{N} + \lceil \sqrt{\mathrm{N}}\, \rceil$ as an overflow takes place. For the cost of first-time array insertion of an element, we allocate one cyber dollar. Therefore, the total cost of inserting $n$ elements is $n$.

In the following overflow, we must divide the cost of copying the array of size $N$ to $\mathrm{N} + \sqrt{N}$ by the additional insertions from $N\ +\ 1\ to\ N\ +\sqrt{N}$. As a result, each such insertion will incur an extra price of $\frac{N+\sqrt{N}}{\sqrt{N}} = 1 + \sqrt{N}$ cyber dollars.

Thus, an additional $1+\sqrt{N}$ for subsequent copying is charged for each insertion into the array, 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 yet another way to approximate the sum. The sum is constrained by the integral from $1\ to\ n$ and $\int_0^{n-1} \sqrt{x}\,dx$. These represent the sum of the heights below and above the curve $f(x) = \sqrt{x}$ . As $\int \sqrt{x}\ dx = x^{\frac{3}{2}}$, we can conclude.

- **1.6.70**
  **Given an array, A, describe an efficient algorithm for reversing A. For example, if A=3,4,1,5, then its reversal is A=5,1,4,3. You can only use $O(1)$ memory in addition to that used by A itself. What is the running time of your algorithm?**

  To reverse the array, we can use two pointers, one pointing to the start and the other to the end.

  These can be known as $start$ and $end$. We switch the components they point to, increment $start$, and decrement $end$ while $start$ is less than 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 conventional algorithm has a worst-case runtime complexity of $O(n)$. The amount of space used (apart from the array) is constant.

- **1.6.77**
  **Given an integer $k > 0$ and an array, A, of $n$ bits, describe an efficient algorithm for finding the shortest subarray of $A$ that contains $k$ $1's$. What is the running time of your method?**

  We initialize the lower and higher pointers before scanning A. When a 1 appears for the first time in lower, we note it there. We then scan from that point on until we find $k$ $1's$, at which point we set upper to that index. Additionally, we must keep track of the lower and upper places so that we may later compare them to other potential subarrays containing $k$ $1's$.

  The objective at this point is to continue scanning to the right while keeping the requirement that $A[lower : upper]$ contains $k$ $1's$ by increasing $lower$ and $upper$. Every time we come across one of these subarrays, we measure its length against the preceding one to see whether any adjustments should be made to our data.
  The bounds of the shortest subarray containing $k$ $1's$ are returned once the full array has been scanned.

  Essentially, we are using a sliding window. Note that we only scan an element a maximum of twice. After we identify the first subarray having $k$ $1's$, we perform these scans. The runtime complexity is in the order of $O(n)$. The amount of space used (apart from the array) is constant.
  The variables $start$ and $end$ are used to indicate the bounds of the shortest subarray in the pseudocode displayed below.

  **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 $i \leftarrow 0$ to $n$ do
  if $A[i] = 1$ then
     $lower \leftarrow i$
     $count \leftarrow i$
     for $j \leftarrow i + 1$ to $n$ do
       if $A[j] = 1$ then
          $count \leftarrow count + 1$
       if $count = k$ then
           $upper \leftarrow j$
           $start \leftarrow lower$; $end \leftarrow upper$     ($Record\ current\ shortest\ subarray$)
       break
  break

while $upper < n$ do
 $lower \leftarrow lower + 1$
while $A[lower] = 0$ do                                      ($Move\ lower\ to\ the\ next$ 1)
  $lower \leftarrow lower + 1$
  $upper \leftarrow upper + 1$
while $A[upper] = 0$ do                                      ($Move\ upper\ to\ the\ next$ 1)
  $upper \leftarrow upper + 1$

if $(upper - lower) < (end - start)$ then
start $\leftarrow lower$; $end \leftarrow upper$
return $(start, end)$
```