CS 600 A - Advanced Algorithms – Homework 1

Name: Deepali Nagwade | 20013393

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

Answer:

- $1/n$
- 2^{100}
- $\log \log n$
- $\log^{.5} n$
- $\log^2 n$
- $n^{.01}$
- $n^{.5}$, $3n^{.5}$
- $2^{\log n,}\ 5n$
- $N \log_4 n$, $6\ n \log n$
- $2n \log^2 n$
- $4n^{3/2}$
- $4^{\log n}$
- $n^2 \log n$
- n^3
- 2^n
- 4^n
- $2^{2\wedge n}$

R.1.6.9 Bill has an algorithm, find2D, to find an element x in an n × n array A. The algorithm find2D iterates over the rows of A and calls the algorithm arrayFind, of Algorithm 1.12, 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?

Answer: The worst-case running time for algorithm find2D = O (n^2).
Therefore, it is not a linear time algorithm.

Let's say, there's an element x as the last value in n*n array. Here find2D would call the algorithm arrayFind n number of times and it would search the elements until x is found in the array. So, n number of comparisons are executed for every time arrayFind runs. Therefore, the running time for this algorithm O (n^2).

Here the size is n^2 which shows it is O(N) time algorithm. This algorithm is a linear time algorithm. As it is running time is same as the linear function of input size.

R 1.6.22 Show that n is O (n log n).

Answer:

f(n) = n
Let c > 0 be any constant
if n_0 = 1/c, then 1 <= c n for n > n_0
Therefore, if n >= n_0, f(n) = n <= n log n <= c n log n

R 1.6.23 Show that n2 is w(n)

Answer:

f(n) = n^2
Let c > 0 be
if n_0 = c, then for n > n_0, n > c
Therefore, if n >= n_0, f(n) = n^2 >= c n

R 1.6.24 Show that n3 is omega n3

Answer:

According to the definition of big-Omega, the real constant is supposed to be searched c>0 and integer constant $n_0 \geq 1$ such that $n^3 \log n$ is $\Omega(n^3)$.

So, when c =1 & n_0 =2 log n $\geq$ 1 in the range.

$n^3 \log n >= n^3$ for n >= 2

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

Answer:

P =1 / $n^{1/2}$

here binomial distribution is used as number of trails is n and it is independent.

So, let the probability of getting selection in the basket be p.

$P(X=x) = \binom{n}{x} p^x q^{n-x}$

$P(X=x) = \binom{n}{x} (1/n^{1/2})^x (1-(1/n)^{1/2})^{n-x}$

Using Chernoff bound $E(X) = n * (1/n^{(1/2)}) = n^{1/2}$

$P(X_i=1) = 1/n^{1/2}$

The upper bound on the probability is more than $3\,n^{1/2}$ balls in the basket

Is $P(X \geq 3\,n^{1/2}) = P((X \geq n^{1/2}(1+2))$

$= e^{-(2n^{**}(1/2)\ln 2)/2}$

C 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?

Answer:
Suppose n = 16, then when 1 to 16 is counter binary is as follow:

0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 0110 1110 1111 10000

As shown above the right bit changes as we count to 16 times, the second bit changes every 4 times, the third bit changes 4 times and the last bit changes twice. We can assume n is the power of 2 and hence $2^x = n$ so if it is 0 to n. the first bit changes n times. The second changes n/2 and so on.

$$totalTime(n) = t * totalBits(n)$$
$$= \sum_{i=1}^{n-1} bitChange(n) = \sum_{i=1}^{n-1} \begin{cases} 1 & if\ n\ is\ even \\ & if\ n\ is\ odd \end{cases}$$

Therefore, it is O(n)

C 1.6.39 Consider the following recurrence equation, defining a function T(n):

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

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

Answer:

Here n = 1, then T(1) = 2, T(0) =2 is true.

So, assume true for n-1

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

C 1.6.52

Show that the summation $\sum_{i=1}^{n} \log_2 (i)$ is O (n log n)

Answer:

We know that for all positive integers i, $\log_2(i)$ is less than or equal to $\log_2(n)$, where n is also a positive integer. This is because log functions are monotonically increasing.
Therefore, for each i from 1 to n, we have:

$\log_2(i) \leq \log_2(n)$

Now, let's sum both sides of the inequality for i from 1 to n:

$$\sum_{i=1}^{n} \log_2 (i) <= \sum_{i=1}^{n} \log_2 (n)$$

The right side of the inequality is just n times $\log_2(n)$:

$$\sum_{i=1}^{n} \log_2 (i) < n * \log_2 (n)$$

Since n * $\log_2(n)$ is a product of n and a logarithmic term, it is in the O(n log n) time complexity class.

Therefore, we have shown $\sum_{i=1}^{n} \log_2 (i)$ is indeed O(n log n) based on the simple proof that the summation is bounded by n * $\log_2(n)$.

C 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 $\sqrt{N}$ additional cells, going from capacity N to N + $\sqrt{N}$. 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.

Answer: The time taken for copying a single value from a table to the array takes O (1) time and hence the time taken to copy N elements is O (N). If the length of the array is extended to √N instead of 2N, then total time into an array = $O(N+N^{3/2})$.

The time it takes to add N elements into an array $N + \sqrt{N} = O\left(\sum^{i=N}_{i=1}(N + \sqrt{N_i})\right)$

$=O(N+\sqrt{N})$

$= O(N+N^{3/2})$.

The worst-case time complexity and the average case time complexity is $O(N+N^{3/2})$ and $O(N^{3/2})$ respectively.

A 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?

Answer: Let the length of an array be n and duplicate the values of

A[0] : A[n-1] to X[n-1] to X[0]

Have a For loop run: - for (i=0; i<n; i++)
X[n-1-i] = A[i];

The running time of this algorithm will be O(n), memory O(1)

A 1.6. 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?

Answer: The efficient algorithm for finding the shortest subarray of A that contains k 1's, the algorithm is as follows:

- Let I be the first index of the occurrence of 1 in an array.
- Find inside the array the value K 1's.
- The left side index I is removed, and scanning continues until next 1 is found.
- The right side is extended so that 1's = k
- Update the length when the window is smaller.

Time Complexity of this algorithm = O(n).