Yash Avinash Patole

Course : CS600

Homework 1

Exercise no. 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.

6*n* log *n* 2100 log log *n* log2 *n* 2log *n*
22*n* _
√
***n*_ *n*0.01 1/*n* 4*n*3/2**
3*n*0.5 5*n* 2*n* log2 *n*_ 2*n* *n* log4 *n*
4*n* *n*3 *n*2 log *n* 4log *n*
√
log *n*
***Hint:* When in doubt about two functions fn and gn, consider log fn and log gn or 2fn and 2gn.**

Answer:

- $1/n$
- 2^{100}
- $\log \log n$
- $\sqrt{\log n}$
- $\log^2 n$
- $n^{0.01}$
- $\underline{\lceil\sqrt{n}\rceil, 3\, n^{0.5}}$
- $\underline{2^{\log n}, 5\, n}$
- $\underline{n \log_4 n, 6\, n \log n}$
- $|\_2\, n \log^2 n\_|$
- $4\, n^{3/2}$
- $4^{\log n}$
- $n^2 \log n$
- n^3
- 2^n
- 4^n
- 2^{2n}

Exercise 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 1.3.2, 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?

Answer:

The Algorithm find2D has the worst-case running time of O (n^2).

Assume that element X is the n*n array's final value. When X was not found in the array, find2D would make n calls to the arrayFind algorithm to search each element. As a result, each time arrayFind is executed, n comparisons are made. This method's execution time is therefore O(n2), or n * n. The process requires O(N) time since in this case the magnitude of N of A is n2. This algorithm therefore operates in linear time. As it is, the running duration and input size have a linear relationship.

Exercise 1.6.22

Show that n is o (n log n).

Answer:
F(n) = o g (n)

F(n)/ g(n)$\longrightarrow$0 for n$\rightarrow \infty$

F(n) /g(n) = n/ n log n =0 for n $\rightarrow \infty$

Therefor n = o (n log n).

Or

According to the little o, ther should be a constant c>0, where n_0 >0 | n < c n log n.

The log n is > 1/c for n ≥ n_0 is true if n >$2^{1/c}$. So, n_0 = ceiling of (1 + $2^{1/c)}$.

Exercise 1.6.23

Show that n^2 is ω(n).

Answer:

$f(n) = n^2$
Let $c > 0$ be
if $n_0 = c$, then for $n > n_0$, $n > c$
$F(n)/ g(n) \rightarrow 0$ for $n \rightarrow \infty$
$F(n) /g(n) = n^2/ n$
Therefore, if $n >= n_0$, $f(n) = n^2 >= cn$

Exercise 1.6.24

Show that n3 log n is Ω(n3).

Answer:
The idea is to look for the real constant c>0 and the integer constant n0 1 such that n3 log n is (n3) referring to the definition of big-omega.
Since, $n^3 \log n >= n^3$ for $n >= 2$
Hence, $c = 1$ and $n_0 = 2\log n >= 1$ in the range.

Exercise 1.6.32

Suppose we have a set of n balls and we choose each one independently with probability 1/n1/2 to go into a basket. Derive an upper bound on the probability that there are more than 3n1/2 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.
$P (X = x) = (^n_x)\ p^x\ q^{n-x}$
$P (X = x) = (^n_x)\ (1 / n^{1/2})^x\ (1 – (1 / n)^{1/2})^{n-x}$
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 $P(X \geq 3\ n^{1/2}) = P((X \geq n^{1/2}(1+2))$

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

Creativity

Exercise 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:

Considering n=16, then for 1 to 16 the counter binary is as follows:

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

Seeing the above counter binary every right bit changes as we count to 16 times, while the second bit changes every 4 times, also the third bit changes 4 times and the last bit changes twice. Assuming that the power of n is 2 and therefore, $2^x = n$ for 0 to n. Here the first bit changes n times. While the second bit changes n/2 and the others as follows.

Working out the equation that is required,

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

Considering work as w where work required to change a bit,

$W * \sum_{i=0\ to\ x} n/2^i = w * n * \sum n/2^i < 2 * n * 2 = O(n)$

Which shows that it is O(n).

Exercise 1.6.39

Consider the following recurrence equation, defining a function Tn:
Tn=1 if n=0 2T(n−1) otherwise,
Show, by induction, that T(n)=2^n.
Answer:
Now,
$T(0) = 1 = 2^0$
$T(1) = 2T(0) = 2 = 2^1$
$T(2) = 2T(1) = 4 = 2^2$
So, let's assume true for n-1.
T(n) = 2T(n-1) = 2 * 2^{n-1} = 2^n

Exercise 1.6.52
Show that the summation ∑i=1 to n log2 i is O(n log n).

Answer:
Now, in this case,
F(n) is O g(n) if f(n) <= c* g(n)
So,
$\sum_{i=0}^{n} log_2 i$ can be expanded as $\log_2 1 + \log_2 2 + \log_2 3 + \ldots + \log_2(n-1) + \log_2(n)$
Also, $\sum_{i=0}^{n} log_2 i$ is equal to log(n!)
Now nlogn = log(n) +log(n) + ….. + log(n) = n log n
Hence, $log_2 1 + log_2 2 + log_2 3 + \ldots + log_2(n-1) + log_2(n) <= c(log(n) + log(n) + \ldots + log(n))$
Therefore, for c = 1 and n >= 2 the condition is satisfied.
Hence, $\sum_{i=0}^{n} log_2 i$ is O(n log n)

Exercise 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 root of N additional cells, going from capacity n to N+√N. Show that performing a sequence of n add operations (that is, insertions at the end) runs in Θ(n3/2) time in this case.

Answer:
Because it takes O(1) time to transfer a single value from a table to an array, while the time taken to copy N elements is O(N). Total time into an array equals O(N+N3/2) if the array's length is extended from 2N to √N.

The time taken to add N elements into an array will be:

$$N + \sqrt{N} = O\left(\sum^{i=N}_{i=1}(N + \sqrt{N_i})\right)$$

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

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

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

Application

Exercise 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 O1 memory in addition to that used by A itself. What is the running time of your algorithm?

Answer:

Have a For loop run from i = 0 to A.length/2. Create a temp variable to hold the value of A[i]. Set the value of A[i] to A[A.length - 1 – i].

Set the value of A[A.length - 1 – i] to the temp variable. The loop terminates and the array is reversed. O(1) memory, O(n) time

Consider the length of the array as n while duplicating the values of A[0] : A[n-1] to X[n-1] to X[0]

Pseudocode to execute the loop

```
for (i=0; i<n; i++) {
```

X[n-1-i] = A[i]

}

Therefore, the running time of this particular algorithm will be O(n).

Exercise 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?

Answer:

To find the shortest subarray of A that contains k 1's an efficient algorithm is required which can be as follows:

- The first index of the occurrence of 1 in an array be I.
- Set I variable equal to 0.
- Find inside the array the value K 1's.
- Loop that iterates from 0 to the length of the array.
- Then left side index I is removed and scanning continues until next 1 is found.
- When right side is extended Checks for I == k, if so, break. Check for A[i] = 1, if so, increment temp, else I = 0. Return [i-k, i-1]
- The length is updated when the window is smaller.
- Time complexity of this algorithm is n time traveled so; it is O(n).