i. 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.

Answer

1. 1 / n
2. 2^{100}
3. Loglogn
4. $\sqrt{logn}$
5. $log^2 n$
6. $n^{0.01}$ (positive powers are always bigger than the logarithmic function)
7. $\sqrt{n}$, $3n^{0.5}$
8. 5n, 2^{logn} (let log n = x so by definiton n = 2^x so 5n = 5 * 2^x and 2^{logn} = 2^x which is almost same)
9. $nlog_4 n$, 6nlogn
10. $2nlog^2 n$
11. $4n^{3/2}$
12. 4^{logn}(let logn = x so by definition n = 2^x so 4^{logn} = 4^x which can be simplified to 2^{2x} and n^2logn = 2^{2x} * x so we can see that 4^{logn} is smaller than n^2logn
13. n^2logn
14. n^3
15. 2^n
16. 4^n
17. 2^{2n}

ii. 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 xis 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 of find2D in terms of n is $O(n^2)$. This is not a linear-time algorithm. Because for each no of rows you are running the arrayFind algorithm and the worst-case scenario will occur if the element is not present in the n * n array.

Also, the definition of a linear time algorithms is that the algorithm is executed in O(n) time but in our case it is getting executed in $O(n^2)$

iii. Show that n is o(nlogn).

Answer.

In order to prove n is o(nlogn) we will use L'Hospital's rule which is used to calculate the limits of the indeterminate forms.

According to the rule if

$\lim_{n \to \infty} f(n) / g(n) = 0$ then we can say that f(n) is little o(g(n))

Let f(n) = n & g(n) = nlogn

According to L'Hospital's rule

$$\lim_{n \to \infty} f(n) / g(n) = \lim_{n \to \infty} f'(n) / g'(n)$$

So f'(n) = 1 & g'(n) = 1 + logn

Substitute the value of n→ ∞ in f'(n) & g'(n) we get f'(∞) = 1 & g'(∞) = ∞

Therefore f'(n) / g'(n) = 0 hence f(n) is little o(g(n)) i.e. n is o(nlogn)

iv. Show that n^2 is $little\ \omega(n)$

Answer

In order to prove n^2 is little ω(n) we will use L'Hospital's rule which is used to calculate the limits of the indeterminate forms.

According to the rule if

$\lim_{n \to \infty} f(n) / g(n) = \infty$ then we can say that f(n) is little ω(g(n))

Let f(n) = n^2 & g(n) = n

According to L'Hospital's rule

$$\lim_{n \to \infty} f(n) / g(n) = \lim_{n \to \infty} f'(n) / g'(n)$$

So f'(n) = 2n & g'(n) = 1

Substitute the value of n→ ∞ in f'(n) & g'(n) we get f'(∞) = 2 * ∞ = ∞ & g'(∞) = 1

Therefore f'(n) / g'(n) = ∞ hence f(n) is little ω(g(n)) i.e. n^2 is little ω(n)

v. Show that n^3log n is Ω(n^3)

Answer:

We can use two methods to prove the above statement.

Method 1:

n^3logn is Ω(n^3) if there exist for atleast one choice of constant c > 0 such that it satisfies this equation f(n) >=c * g(n) for all n > n0.

Now f(n) = n^3logn and g(n) = n^3

So, n^3logn >= cn^3

So for c = 1 and n0 =2 this equation is satisfied.

So for all n > 2 & c >=1 this equation is satisfied.

Hence n^3log n is Ω(n3)

Method 2:
In order to prove n^3logn is big $\Omega(n^3)$ we will use L'Hospital's rule which is used to calculate the limits of the indeterminate forms.

According to the rule if
$\lim_{n \to \infty} f(n) / g(n) = \infty$ then we can say that f(n) is big Ω(g(n))

Let f(n) = n^3logn & g(n) = n^3

f'(n) = n^2 + 3 * n^2logn f''(n) = 2n + 3 (n + 2nlogn) f'''(n) = 2 + 3 (1 + 2(1 + log(n)))

g'(n) = $3n^2$, g''(n) = 6n, g'''(n) = 6

Substitute the value of n→ ∞ in f'''(n) & g'''(n) we get f'''(∞) = ∞ = ∞ & g'''(∞) = 6

Therefore f'''(n) / g'''(n) = ∞ hence f(n) is big Ω(g(n)) i.e. n^3logn is big $\Omega(n^3)$

vi.
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:
We are given the probability that a ball is selected to go into a basket is 1 / $n^{1/2}$ which is nothing but 1 / $\sqrt{n}$ i.e. p_i.

Thus, $\mu = \sum_{i=1}^{n}$ p_i = n * 1 / $\sqrt{n}$ = $\sqrt{n}$

Using Chernoff bound for $\delta = 2$ we have
$Pr(X > (1 + \delta) * \mu) < [e^{\delta} / (1 + \delta)^{(1+\delta)}]^{\mu}$

i.e.
$Pr(X > 3\sqrt{n}) < [e^2 / 3^3]^{\sqrt{n}}$

Hence proved

vii.

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 ithat must change in going from i to i+1?

Answer:
Let us assume n where n = 2^x where x is the power. It is a bit easier to understand
Let x = 4 so its binary representation from 1 to 16 is

0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
10000

So from the given representation of a binary numbers from 1 to 16 we can see that the 0th bit is changing n times, 1st bit is changing n / 2 times, 2nd bit is changing n /4 , 3rd bit is changing n / 8 times and 4th bit is changing n / 16 times

Therefore , in mathematical representation we can express it as

$$\sum_{i=0}^{x} n / 2^i = n * \sum_{i=0}^{x} 2^{-i} < 2n$$

So from the given expression we can see that the total running time of counting from 1 to n is O(2n) which is nothing but O(n).

viii.

Answer:

It is given that T(n) = 1 when n = 1 otherwise it is 2 * T(n - 1)

Lets put n = 2

$T(2) = 2 * T(1) = 2 * 1 = 2^1$
$T(3) = 2 * T(2) = 2 * 2 = 2^2$
$T(4) = 2 * T(3) = 2 * 2^2 = 2^3$

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

And for n = k + 1
$T(k + 1) = 2 * T(k) = 2 * 2^{k-1} = 2^k$

which came out to be true.
ix.

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

Answer:
f(n) is Og(n) if f(n) <= c * g(n)

Now $\sum_{i=1}^{n} \log_2 i$ can be expanded as $\log_2 1 + \log_2 2 + \log_2 3 + ... + \log_2(n-1) + \log_2(n)$

Also $\sum_{i=1}^{n} \log_2 i$ is equivalent to log(n!)

Now nlogn = log(n) + log(n) + … + log(n) = nlog(n)

so
$\log_2 1 + \log_2 2 + \log_2 3 + ... + \log_2(n-1) + \log_2(n) <= c(\log(n) + \log(n) + \ldots + \log(n))$

So for c = 1 & n >= 2 the condition is satisfied.

Hence $\sum_{i=1}^{n} \log_2 i$ is O(nlogn)

x.
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.

Answer:

We will use cyber dollar method.

We will overcharge the cheapest operation(insert) and normal charge the costly operation (copy).

We know that insert opearion takes O(1) time for 1 item and the copy operation takes O(n + $\sqrt{n}$) $for\ n\ items$

Since we have mentioned the time to insert one element is O(1). So the time to insert n elements is O(N)

For copy the run time is O(N + $\sqrt{N}$) for n items

So for each item insertion and copy the time would become (N + $\sqrt{N}$) / $\sqrt{N}$ = 1 + $\sqrt{N}$

So for the total operation would become,

$$\sum_{i=1}^{n} 1 + \sqrt{N} + 1 = 2 + N * \sqrt{N} = 2N + N^{3/2}$$

This is the worst case

Hence the average case will be $\theta(N^{3/2})$.

xi.

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:

Algorithm reverseArray(A, n):
Input: An array A storing n >=1 integers
Output: The reverse of an array A
middlePoint ← n / 2
for i ← 0 to middlePoint - 1 do
        temp ← A[i]
        A[i] ← A[n - i - 1]
        A[n - i - 1] ← temp
return A

This is a linear time algorithm running in O(n) time. No additional space is required.

Steps:
1. We will be using array data structure to store the n elements. Declare an array A and store the n elements.
2. Set the start variable as 0 and end variable as n - 1.
3. Run a while loop
4. Inside a while loop declare a temp variable and store the value of A[start].
5. Copy the value of A[end] into A[start].

6. Copy the value of temp into A[end]
7. Repeat Step 4 - 6 until start is not equal to end.
8. Return or print the Array A.
9. Stop

xii.

Given an integer k>0and an array, A, of n bits, describe an efficient algorithm for finding the shortest subarray of Athat contains k1's. What is the running time of your method?

Answer:

1. We will be using array data structures to store the n elements. Declare an array A and store the n elements.
2. Declare a variable named i and store its value as 0.
3. Find the first occurance of 1 in an array from i to n - 1 and store its index in i.
4. Declare a variable j and keep on scanning from i to n - 1 until you find k 1’s.
5. Once you find k number of 1’s, the length of the shortest subarray is calculated as j - i + 1.
6. Discard the left most one present at index i and keep on scanning until you find next 1.
7. Keep on scanning from j to n - 1 until you find k 1’s.
8. If the new length of the subarray is lesser than update it i.e new_length = j - i + 1.

This is a linear time algorithm. We are exploring each element in an array twice so the complexity of an algorithm is O(2n) which is nothing but O(n).

We are not using any extra space, so the space complexity is O(1)