R-1.7

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 6n log n | 2­­­100 | loglog n | log2 n | 2log n |  |
| O(n log n) | O(1) | O(log n) | O(n2) | O(n) | O() |
|  | n0.01 | 1/n | 4n3/2 | 3n0.5 | 5n |
| O(n0.5) | O(0.01n) | O(1/n) | O((3/2)n) | O(n0.5) | O(n) |
|  | 2n | nlog4 n | 4n | n3 | n2log n |
| O(n3) | O(2n) | O(n log n) | O(4n) | O(n3) | O(n2log n) |
| 4log n |  |  |  |  |  |
| O(4log n) | O(n0.5) |  |  |  |  |

Time in ascending order:

1/n, 2100, loglog n, , log2 n, n0.01, , 3n0.5, 2log n, 5n, nlog4 n, 6nlog n, , 4n3/2, 4log n, n2log n, n3, 2n, 4n, .

Followings are big-Theata for each other: and 3n0.5; 2log n and 5n; nlog4 n and 6nlog n.

R-1.9

The worst case is that the item is at (n, n). Thus the algorithm has to search the whole matrix to find it, which is n \* n items. Thus, the worst-case running time is n2, which is quadratic and not linear.

R-1.22

We need to prove that for any constant c >0, there is a constant n0 >0 such that n <= c\*n log n for n >= n0.

Assume n <= c n log n, then 1/c <= log n, n >= 21/c. Let n0 = 21/c, then n >=21/c. c \* n log n >= c\* n \*1/c= n. Thus n <= n log(n) when n >= n0 proved.

R-1.23

To show that n2 is , let any c > 0. If n0 = 1/c, then for n>=n0=1/c, c \* n2 >=c \* n \* 1/c = n. Since cn2 >= n, proved.

R-1.24

We need to find a real constant c > 0 and an integer constant n0 >= 1such that n3logn >= c\*n3, for n >= n0. If c = 1 n0 = 2, then n3logn: cn3 => logn : c => logn >= 1 when n >= n0 = 2.

R-1.32

According to textbook Page 46- Chernoff bound Theorem,

. From and 3n1/2 we know that In conclusion we can write

C-1.4

If right-most bit changes 4n times, then 2nd right-most bit changed 2n times and so on.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 0 | 1 | 10 | 11 | 100 | 101 | 110 | 111 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

Assume there are n numbers, then the right-most bit changes n times, 2nd right-most bit changes n/2 times, then n/(2^2), n/(2^3) and so on. Suppose nth number has k bits. Time

C-1.7

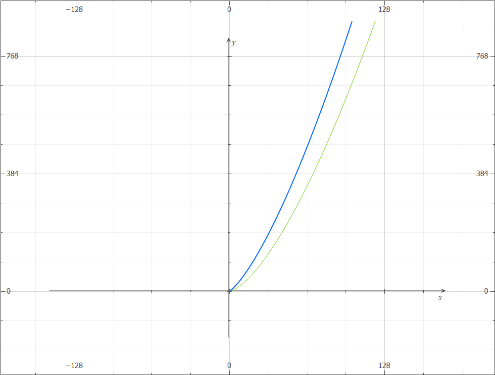
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| n=0 | n=1 | n=2 | n=3 | n=4 |
|  | 2T(0) | 2T(1) | 2T(2) | 2T(3) |
| 1 | 2 | 4 | 8 | 16 |

Assume it’s working for n-1, then T(n)=2T(n-1)=2\*2n-1=2n.

C-1.22

The equation becomes:

C-1.30

To insert n elements, the total cost in Cyber$ is $n without considering copying. Say the initial size of array is N, then the expansion would be N+. For elements newly inserted, they would be charged $ . Then each element’s insertion would be charged for future copying plus 1 for insertion, the total cost is . As we can see, f(n) is obviously O(n3/2), to prove it’s Θ(n3/2), we also need to prove f(n) is Ω(n3/2). Thus we need to prove there is a real constant c>0 and an integer constant n0>=1 such that f(n) >= c\*n3/2 for n>=n0. When c = 2/3, as shown in the figure, blue line is f(n) and green line is 2/3\*n3/2. Then f(n) >= c\*n3/2 proved. Thus, the total running time is Θ(n3/2).

A-1.8

The algorithm needs to go over the whole array once to assign new pointer to each element, thus the running time is O(n).

A-1.15

To form a subarray, we need at least two pointers say (i, j). Move ‘j’ to first 1 then move ‘i’ to j. Then scan through the rest of the array to find the next 1 and move ‘j’. Repeating until we have included k 1s in this subarray. Then we still need to scan through the whole array with ‘j’ and move ‘i’ accordingly to maintain k 1s and save each length. Thus, we need to do at most 2n operations. Thus the total running time is deduct to O(n).