ASSIGNMENT

Data structure and algorithms

Safiullah khan
BSCS 3rd D

Roll number # 14795
Submitted to Sir Jamal Abdul Ahad

Chapter No # 1

The Role of Algorithms in Computing

Exercises

1.1-1 Describe your own real-world example that requires sorting. Describe one that requires ûnding the shortest distance between two points.

Answer

An example of a real world situation that would require sorting would be if you wanted to keep track of a bunch of people's file folders and be able to look up a given name quickly. A convex hull might be needed if you needed to secure a wildlife sanctuary with fencing and had to contain a bunch of specific nesting locations.

1.1-2 Other than speed, what other measures of efficiency might you need to consider in a real-world setting?

Answer

One might measure memory usage of an algorithm, or number of people required to carry out a single task.

1.1-3 Select a data structure that you have seen, and discuss its strengths and limitations.

Answer

An array. It has the limitation of requiring a lot of copying when re-sizing, inserting, and removing elements.

1.1-4 How are the shortest-path and traveling-salesperson problems given above similar? How are they different?

Answer

They are similar since both problems can be modeled by a graph with weighted edges and involve minimizing distance, or weight, of a walk on the graph. They are different because the shortest path problem considers only two vertices, whereas the traveling salesman problem considers minimizing the weight of a path that must include many vertices and end where it began.

1.1-5 Suggest a real-world problem in which only the best solution will do. Then come up with one in which

Answer

If you were for example keeping track of terror watch suspects, it would be unacceptable to have it occasionally bringing up a wrong decision as to whether a person is on the list or not. It would be fine to only have an approximate solution to the shortest route on which to drive, an extra little bit of driving is not that bad

1.1-6 Describe a real-world problem in which sometimes the entire input is available before you need to solve the problem, but other times the input is not entirely available in advance and arrives over time.

Answer

A real-world example is processing online orders in a delivery service.

1.2-1 Give an example of an application that requires algorithmic content at the application level, and discuss the function of the algorithms involved.

Answer

A program that would pick out which music a user would like to listen to next. They would need to use a bunch of information from historical and popular preferences in order to maximize.

1.2-2 Suppose that for inputs of size n on a particular computer, insertion sort runs in 8n2 steps and merge sort runs in 64 n lg n steps. For which values of n does insertion sort beat merge sort?

Answer

We wish to determine for which values of n the inequality 8n 2 < 64n log2 (n) holds. This happens when n < 8 log2 (n), or when n ≤ 43. In other words, insertion sort runs faster when we're sorting at most 43 items. Otherwise merge sort is faster.

1.2-3 What is the smallest value of n such that an algorithm whose running time is 100n2 runs faster than an algorithm whose running time is 2 n on the same machine?

Answer

We want that 100n 2 < 2 n. note that if n = 14, this becomes 100(14)2 = 19600 > 2 14 = 16384. For n = 15 it is 100(15)2 = 22500 < 2 15 = 32768. So, the answer is n = 15.

Problems

1-1 Comparison of running times For each function f .n/ and time t in the following table, determine the largest size n of a problem that can be solved in time t, assuming that the algorithm to solve the problem takes f .n/ microseconds.

Chapter No # 2

Getting Started

Exercise

2.1-1 Using Figure 2.2 as a model, illustrate the operation of I NSERTION-SORT on an array initially containing the sequence h31; 41; 59; 26; 41; 58i.

Answer

31	41	59	26	41	58
31	41	59	26	41	58
31	41	59	26	41	58
26	31	41	59	41	58
26	31	41	41	59	58
26	31	41	41	58	59

2.1-2 Consider the procedure SUM-ARRAY on the facing page. It computes the sum of the n numbers in array A[1:n]. State a loop invariant for this procedure, and use its initialization, maintenance, and termination properties to show that the SUMARRAY procedure returns the sum of the numbers in A [1: n].

Answer

1: for j = 2 to A.length do

2: key = A[j]

3: // Insert A[j] into the sorted sequence A[1..j − 1].

4: i = j − 1

5: while i > 0 and A[i] < key do

6: A[i + 1] = A[i]

7: i = i – 1

8: end while

9: A[i + 1] = key

10: end for

2.1-3 Rewrite the I NSERTION-SORT procedure to sort into monotonically decreasing instead of monotonically increasing order.

Answer

On each iteration of the loop body, the invariant upon entering is that there is no index k < j so that A[k] = v. In order to proceed to the next iteration of the loop, we need that for the current value of j, we do not have A[j] = v. If the loop is exited by line 5, then we have just placed an acceptable value in i on the previous line. If the loop is exited by exhausting all possible values of j, then we know that there is no index that has value j, and so leaving NIL in i is correct.

2.1-4 Consider the searching problem:

Input: A sequence of n numbers ha1; a2; : : : ; ani stored in array A[1:n] and a value x.

Output: An index i such that x equals A[1:n] or the special value NIL if x does not appear in A. Write pseudocode for linear search, which scans through the array from beginning to end, looking for x. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulûlls the three necessary properties.

Answer

1: i = NIL

2: for j = 1 to A.length do

3: if A[j] = v then

4: i = j

5: return i

6: end if

7: end for

8: return i

2.1-5 Consider the problem of adding two n-bit binary integers a and b, stored in two n-element .....?

Answer

def ADD_BINARY_INTEGERS(A, B, n):

Initialize

carry to 0 carry = 0

Initialize the result array

C with size n + 1 C = [0] * (n + 1)

Perform bitwise addition from LSB to MSB

for i in range(n):

Sum of the current bits in A and B, and the carry

sum_bit = A[i] + B[i] + carry

Update the result bit at position i

C[i] = sum_bit % 2 # Modulo 2 gives the binary digit at this position

Update carry (1 if sum_bit is 2 or 3, otherwise 0)

carry = sum_bit // 2 # Integer division gives the carry

If there's a carry left after the last addition, set it to the MSB of C C[n] = carry

return C

2.2-1 Express the function n 3=1000 C 100n2 ? 100n C 3 in terms of ,-notation

Answer

n 3/1000 − 100n 2 − 100n + 3 ∈ Θ(n 3)

2.2-2 Consider sorting n numbers stored in array ACE1 W n€ by ûrst ûnding the smallest element of ACE1 W n€ and exchanging it with the element in ACE1€. Then ûnd the smallest element of ACE2 W n€, and exchange it with ACE2€. Then ûnd the smallest element of ACE3 W n€, and exchange it with ACE3€. Continue in this manner for the ûrst n ? 1 elements of A. Write pseudocode for this algorithm, which is known as selection sort. What loop invariant does this algorithm maintain? Why does it need to run for only the ûrst n?1 elements, rather than for all n elements? Give the worst-case running time of selection sort in ,-notation. Is the best-case running time any better?

Answer

Input: An n-element array A.

Output: The array A with its elements rearranged into increasing order. The loop invariant of selection sort is as follows: At each iteration of the for loop of lines 1 through 10, the subarray A[1..i − 1] contains the i − 1 smallest elements of A in increasing order. After n − 1 iterations of the loop, the n − 1 smallest elements of A are in the first n − 1 positions of A in increasing order, so the n th element is necessarily the largest element. Therefore we do not need to run the loop a final time. The best-case and worst-case running times of selection sort are Θ(n 2). This is because regardless of how the elements are initially arranged, on the i th iteration of the main for loop the algorithm always inspects each of the remaining n−i elements to find the smallest one remaining

2.2-3 Consider linear search again (see Exercise 2.1-4). How many elements of the input array need to be checked on the average, assuming that the element being searched for is equally likely to be any element in the array? How about in the worst case?

Answer

1: for i = 1 to n − 1 do

2: min = i

3: for j = i + 1 to n do

4: // Find the index of the i th smallest element

5: if A[j] < A[min] then

6: min = j

7: end if

8: end for

9: Swap A[min] and A[i]

10: end for

2.2-4 How can you modify any sorting algorithm to have a good best-case running time?

Answer

To modify a sorting algorithm to have a good best-case running time, we can add a preliminary check to see if the array is already sorted. If it is, the algorithm can return immediately without performing further operations, achieving an optimal best-case time of O(n)O(n)O(n) (where nnn is the number of elements in the array) for cases where the array is already sorted.

2.3-1 Using Figure 2.4 as a model, illustrate the operation of merge sort on an array initially containing the sequence h3; 41; 52; 26; 38; 57; 9; 49i.

Answer

3	41	52	26	38	57	9	49
3	41	26	52	38	57	9	49
3	26	41	52	9	38	49	57
3	9	26	38	41	49	52	57

2.3-2 The test in line 1 of the MERGE-SORT procedure reads r, then the subarray A{p : r] is empty. Argue that as long as the initial call of MERGE-SORT.A; 1; n/ has n □ 1, the test r.

Answer

The following is a rewrite of MERGE which avoids the use of sentinels. Much like MERGE, it begins by copying the subarrays of A to be merged into arrays L and R. At each iteration of the while loop starting on line 13 it selects the next smallest element from either L or R to place into A. It stops if either L or R runs out of elements, at which point it copies the remainder of the other subarray into the remaining spots of A.

2.3-3 State a loop invariant for the while loop of lines 12318 of the MERGE procedure. Show how to use it, along with the while loops of lines 20323 and 24327, to prove that the MERGE procedure is correct.

Answer

Since n is a power of two, we may write n = 2k . If k = 1, T(2) = 2 = 2 lg(2). Suppose it is true for k, we will show it is true for k + 1

. T(2k+1) = 2T 2 k+1 2 + 2k+1 = 2T 2 k + 2k+1

= 2(2k lg(2k)) + 2k+1 = k2 k+1 + 2k+1 = (k + 1)2k+1 = 2k+1 lg(2k+1) = n lg(n

2.3-4 Use mathematical induction to show that when n ⍰ 2 is an exact power of 2, the solution of the recurrence

T .n/ D (2 if n D 2 ; 2T .n=2/ C n if n > 2

is T .n/ D n lg n.

Answer

Let T(n) denote the running time for insertion sort called on an array of size n. We can express T(n) recursively as

T(n)= {(1) if n ≤ c T(n − 1) + I(n) otherwise

where I(n) denotes the amount of time it takes to insert A[n] into the sorted array A[1..n − 1]. Since we may have to shift as many as n − 1 elements once we find the correct place to insert A[n], we have I(n) = θ(n

2.3-5 You can also think of insertion sort as a recursive algorithm. In order to sort AŒ1 W n�, recursively sort the subarray AŒ1 W n ⍰ 1� and then insert AŒn� into the sorted subarray AŒ1 W n ⍰ 1�. Write pseudocode for this recursive version of insertion sort. Give a recurrence for its worst-case running time.

Answer

1: BinSearch(a,b,v)

2: if thena > b

3: return NIL

4: end if

5: m = b a+b 2 c

6: if thenm = v

7: return m

8: end if

9: if thenm < v

10: return BinSearch(a,m,v)

11: end if

12: return BinSearch(m+1,b,v)

2.3-6 Referring back to the searching problem (see Exercise 2.1-4), observe that if the subarray being searched is already sorted, the searching algorithm can check the midpoint of the subarray against v and eliminate half of the subarray from further Problems for Chapter 2 45 consideration. The binary search algorithm repeats this procedure, halving the size of the remaining portion of the subarray each time. Write pseudocode, either iterative or recursive, for binary search. Argue that the worst-case running time of binary search is ,.lg n/.

Answer

A binary search wouldn't improve the worst-case running time. Insertion sort has to copy each element greater than key into its neighboring spot in the array. Doing a binary search would tell us how many how many elements need to be copied over, but wouldn't rid us of the copying needed to be done.

2.3-7 The while loop of lines 537 of the INSERTION-SORT procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray AŒ1 W j 1. What if insertion sort used a binary search (see Exercise 2.3-6) instead of a linear search? Would that improve the overall worst-case running time of insertion sort to ,.n lg n/?

Answer

We can see that the while loop gets run at most O(n) times, as the quantity j−i starts at n−1 and decreases at each step. Also, since the body only consists of a constant amount of work, all of lines 2-15 takes only O(n) time. So, the runtime is dominated by the time to perform the sort, which is Θ(n lg(n)). We will prove correctness by a mutual induction. Let mi,j be the proposition A[i] + A[j] < S and Mi,j be the proposition A[i] + A[j] > S. Note that because the array is sorted, mi,j ⇒ ∀k < j, mi,k, and Mi,j ⇒ ∀k > i, Mk,j . Our program will obviously only output true in the case that there is a valid i and j. Now, suppose that our program output false, even though there were some i, j that was not considered for which A[i] + A[j] = S. If we have i > j, then swap the two, and the sum will not change, so, assume i ≤

j. we now have two cases: Case 1 ∃k,(i, k) was considered and j < k. In this case, we take the smalles.

1: Use Merge Sort to sort the array A in time Θ(n lg(n))

2: i = 1

3: j = n

4: while i < j do

5: if A[j] + A[j] = S then

6: return true

7: end if

8: if A[i] + A[j] < S then

9: i = i + 1

10: end if

11: if A[i] + A[j] > S then

12: j = j − 1

13: end if

14: end while

15: return false

2.3-8 Describe an algorithm that, given a set S of n integers and another integer x, determines whether S contains two elements that sum to exactly x. Your algorithm should take ,.n lg n/ time in the worst case.

Answer

To solve this problem efficiently, we can use a sorting and two-pointer approach. The algorithm leverages the idea that if the elements are sorted, we can check for pairs that sum up to xxx more efficiently by using a two-pointer technique. This approach gives us a time complexity of O(nlog⁡n)O(n \log n)O(nlogn) in the worst case.

```
def has_pair_with_sum(S, x):

# Step 1: Sort the set S

 S.sort()

# Step 2: Initialize two pointers

 left = 0
```

```
right = len(S) – 1
 # Step 3: Use two-pointer technique
while left < right:
sum = S[left] + S[right]
 if sum == x:
 return True # Found a pair that sums to x
elif sum < x
: left += 1 # Move left pointer to the right to increase the sum
 else:
right -= 1
# Move right pointer to the left to decrease the sum
 # No such pair found
return False
```

Problems

2-1

Answer

a. The time for insertion sort to sort a single list of length k is Θ(k 2), so, n/k of them will take time Θ(n k k 2) = Θ(nk). b. Suppose we have coarseness k. This meas we can just start using the usual merging procedure, except starting it at the level in which each array has size at most k. This means that the depth of the merge tree is lg(n) − lg(k) = lg(n/k). Each level of merging is still time cn, so putting it together, the merging takes time Θ(n lg(n/k)). c. Viewing k as a function of n, as long as k(n) ∈ O(lg(n)), it has the same asymptotics. In particular, for any constant choice of k, the asymptotics are the same. d. If we optimize the previous expression using our calculus 1 skills to get k, we have that c1n− nc2 k = 0 where c1 and c2 are the coeffients of nk and n lg(n/k) hidden by the asymptotics notation. In particular, a constant choice of k is optimal. In practice we could find the best choice of this k by just trying and timing for various values for sufficiently large n.

2-2

Answer

1. We need to prove that A0 contains the same elements as A, which is easily seen to be true because the only modification we make to A is swapping its elements, so the resulting array must contain a rearrangement of the elements in the original array.

2. The for loop in lines 2 through 4 maintains the following loop invariant: At the start of each iteration, the position of the smallest element of A[i..n] is at most j. This is clearly true prior to the first iteration because the position of any element is at most A.length. To see that each iteration maintains the loop invariant, suppose that j = k and the position of the smallest element of A[i..n] is at most k. Then we compare A[k] to A[k − 1]. If A[k] < A[k − 1] then A[k − 1] is not the smallest element of A[i..n], so when we swap A[k] and A[k − 1] we know that the smallest element of A[i..n] must occur in the first k − 1 positions of the subarray, the maintaining the invariant. On the other hand, if A[k] ≥ A[k − 1] then the smallest element can't be A[k]. Since we do nothing, we conclude that the smallest element has position at most k − 1. Upon termination, the smallest element of A[i..n] is in position i.

3. The for loop in lines 1 through 4 maintain the following loop invariant: At the start of each iteration the subarray A[1..i − 1] contains the i − 1 smallest elements of A in sorted order. Prior to the first iteration i = 1, and the first 0 elements of A are trivially sorted. To see that each iteration maintains the loop invariant, fix i and suppose that A[1..i − 1] contains the i − 1 smallest elements of A in sorted order. Then we run the loop in lines 2 through 4. We showed in part b that when this loop terminates, the smallest element of A[i..n] is in position i. Since the i − 1 smallest elements of A are already in A[1..i − 1], A[i] must be the i th smallest element of A. Therefore A[1..i] contains the i smallest elements of A in sorted order, maintaining the loop invariant. Upon termination, A[1..n] contains the n elements of A in sorted order as desired.

4. The i th iteration of the for loop of lines 1 through 4 will cause n − i iterations of the for loop of lines 2 through 4, each with constant time execution, so the worst-case running time is Θ(n 2). This is the same as that of insertion sort; however, bubble sort also has best-case running time Θ(n 2) whereas insertion sort has best-case running time Θ(n).

2-3

Answer

a. If we assume that the arithmetic can all be done in constant time, then since the loop is being executed n times, it has runtime Θ(n).

1: y = 0

2: for i=0 to n do

3: yi = x

4: for j=1 to n do

5: yi = yix

6: end for

7: y = y + aiyi

8: end for

2-4

Answer

a. The five inversions are (2, 1), (3, 1), (8, 6), (8, 1), and (6, 1).

b. The n-element array with the most inversions is hn, n − 1, . . . , 2, 1i. It has n − 1 + n − 2 + . . . + 2 + 1 = n(n−1) 2 inversions.

c. The running time of insertion sort is a constant times the number of inversions. Let P I(i) denote the number of j < i such that A[j] > A[i]. Then n i=1 I(i) equals the number of inversions in A. Now consider the while loop on lines 5-7 of the insertion sort algorithm. The loop will execute once for each element of A which has index less than j is larger than A[j]. Thus, it will execute I(j) times. We reach this while loop once for each iteration of the P for loop, so the number of constant time steps of insertion sort is n j=1 I(j) which is exactly the inversion number of A.

d. We'll call our algorithm M.Merge-Sort for Modified Merge Sort. In addition to sorting A, it will also keep track of the number of inversions. The algorithm works as follows. When we call M.Merge-Sort(A,p,q) it sorts A[p..q] and returns the number of inversions in the elements of A[p..q], so lef t and right track the number of inversions of the form (i, j) where i and j are both in the same half of A. When M.Merge(A,p,q,r) is called, it returns the number of inversions of the form (i, j) where i is in the first half of the array and j is in the second half. Summing these up gives the total number of inversions in A. The runtime is the same as that of Merge-Sort because we only add an additional constant-time operation to some of the iterations of some of the loops. Since Merge is Θ(n log n), so is this algorithm.

Algorithm 6 M.Merge-Sort(A, p, r)

if p < r then

q = b(p + r)/2c

lef t = M.Merge − Sort(A, p, q)

right = M.Merge − Sort(A, q + 1, r)

inv = M.Merge(A, p, q, r) + lef t + right

return inv

end if

return 0

Chapter No# 3 Characterizing Running Times

Exercises

3.1-1 Modify the lower-bound argument for insertion sort to handle input sizes that are not necessarily a multiple of 3.

Answer

Since we are requiring both f and g to be aymptotically non-negative, suppose that we are past some n1 where both are non-negative (take the max of the two bounds on the n corresponding to both f and g).

Let c1 = .5 and c2 = 1

. 0 ≤ .5(f(n) + g(n)) ≤ .5(max(f(n), g(n)) + max(f(n), g(n)))

= max(f(n), g(n)) ≤ max(f(n), g(n)) + min(f(n), g(n)) = (f(n) + g(n))

3.1-2 Using reasoning similar to what we used for insertion sort, analyze the running time of the selection sort algorithm from Exercise 2.2-2.

Answer

Let c = 2b and n0 ≥ 2a. Then for all n ≥ n0 we have (n + a) b ≤ (2n) b = cnb so (n+a) b = O(n b). Now let n0 ≥ −a 1−1/2 1/b and c = 1 2 . Then n ≥ n0 ≥ −a 1−1/2 1/b if and only if n − n 2 1/b ≥ −a if and only if n + a ≥ (1/2)a/bn if and only if (n + a) b ≥ cnb . Therefore (n + a) b = Ω(n b). By Theorem 3.1, (n + a) b = Θ(n b)

3.1-3 Suppose that ¸ is a fraction in the range 0 < ¸ < 1. Show how to generalize the lower-bound argument for insertion sort to consider an input in which the ¸n largest values start in the ûrst ¸n positions. What additional restriction do you need to put on ¸ ? What value of ¸ maximizes the number of times that the ¸n largest values must pass through each of the middle .1 ? 2¸/n array positions?

Answer

There are a ton of different funtions that have growth rate less than or equal to n 2 . In particular, functions that are constant or shrink to zero arbitrarily fast. Saying that you grow more quickly than a function that shrinks to zero quickly means nothing

3.2-1 Let f .n/ and g.n/ be asymptotically nonnegative functions. Using the basic deûnition of ,-notation, prove that max ff .n/; g.n/g D ,.f .n/ C g.n//.

Answer

Let n1 < n2 be arbitrary. From f and g being monatonic increasing, we know f(n1) < f(n2) and g(n1) < g(n2). So

f(n1) + g(n1) < f(n2) + g(n1) < f(n2) + g(n2)

Since

g(n1) < g(n2), we have f(g(n1)) < f(g(n2)). Lastly, if both are nonegative, then,

f(n1)g(n1) = f(n2)g(n1) + (f(n2) − f(n1))g(n1)

= f(n2)g(n2) + f(n2)(g(n2) − g(n1)) + (f(n2) − f(n1))g(n1) Since f(n1) ≥ 0, f(n2) > 0,

so, the second term in this expression is greater than zero. The third term is nonnegative, so, the whole thing is< f(n2)g(n2)

3.2-2 Explain why the statement, The running time of algorithm A is at least O(n2) is meaningless.

Answer

Since O-notation provides only an upper bound, and not a tight bound, the statement is saying that the running of time of algorithm A is at least a function whose rate of growth is at most n 2.

3.2-3 Is 2 nC1 D O.2n /? Is 2 2n D O.2n /?

Answer

2 nC1 D O.2n /, but 2 2n ¤ O.2n /.

To show that 2 nC1 D O.2n /, we must find constants c; n0 > 0 such that 0 2 nC1 c 2 n for all n n0 :

Since 2 nC1 D 2 2 n for all n, we can satisfy the definition with c D 2 and n0 D 1. To show that 2 2n 6D O.2n /, assume there exist constants c; n0 > 0 such that 0 2 2n c 2 n for all n n0 :

Then 2 2n D 2 n 2 n c 2 n) 2 n c. But no constant is greater than all 2 n , and so the assumption leads to a contradiction.

3.2-4 Prove Theorem 3.1.

Answer

dlg neŠ is not polynomially bounded, but dlg lg neŠ is. Proving that a function f .n/ is polynomially bounded is equivalent to proving that lg f .n/ D O.lg n/ for the following reasons.

If f .n/ is polynomially bounded, then there exist positive constants c, k, and n0 such that 0 f .n/ cnk for all n n0. Without loss of generality, assume that c 1, since if c < 1, then f .n/ cnk implies that f .n/ n k . Assume also that n0 2, so that n n0 implies that lg c .lg c/.lg n/. Then, we have lg f .n/ lg c C k lg n .lg c C k/lg n ; which, since c and k are constants, means that lg f .n/ D O.lg n/.

Now suppose that lg f .n/ D O.lg n/. Then there exist positive constants c and n0 such that 0 lg f .n/ c lg n for all n n0. Then, we have 0 f .n/ D 2 lg f .n/ 2 c lg n D .2lg n / c D n c for all n n0, so that f .n/ is polynomially bounded. In the following proofs, we will make use of the following two facts:

1. lg.nŠ/ D ,.n lg n/ (by equation (3.28)).

2. dlg ne D ,.lg n/, because dlg ne

lg n, and dlg ne < lg n C 1 2 lg n for all n 2. We have lg.dlg neŠ/ D ,.dlg ne lg dlg ne/ D ,..lg n/.lg lg n// D !.lg n/ : Therefore, lg.dlg neŠ/ is not O.lg n/, and so dlg neŠ is not polynomially bounded. We also have lg.dlg lg neŠ/ D ,.dlg lg ne lg dlg lg ne/ D ,..lg lg n/.lg lg lg n// D o..lg lg n/2 / D o.lg2 .lg n// D o.lg n/ : The last step above follows from the property that any polylogarithmic function grows more slowly than any positive polynomial function, i.e., that for constants a; b > 0, we have lgb n D o.na /. Substitute lg n for n, 2 for b, and 1 for a, giving lg2 .lg n/ D o.lg n/. Therefore, lg.dlg lg neŠ/ D O.lg n/, and so dlg lg neŠ is polynomially bounded.

3.2-5 Prove that the running time of an algorithm is ,.g.n// if and only if its worst-case running time is O.g.n// and its best-case running time is ⍰.g.n//.

Answer

Note that lg∗ (2n) = 1 + lg∗ (n),

so,

limn→∞ lg(lg∗ (n)) lg∗ (lg(n))

= limn→∞ lg(lg∗ (2n)) lg∗ (lg(2n))

= limn→∞ lg(1 + lg∗ (n)) lg∗ (n)

= limn→∞ lg(1 + n) n

= limn→∞ 1 1 + n = 0 So,

we have that lg∗ (lg(n)) grows more quickly

3.2-6 Prove that o.g.n// \ !.g.n// is the empty set.

Answer

ϕ 2 = (1 + √ 5) 2 !2 = 6 + 2√ 5 4 = 1 + 1 + √ 5 2 = 1 + ϕ

ϕˆ2 = (1 − √ 5) 2 !2 = 6 − 2 √ 5 4 = 1 + 1 − √ 5 2 = 1 + ϕˆ

3.2-7 We can extend our notation to the case of two parameters n and m that can go to 1 independently at different rates. For a given function g.n; m/, we denote by O.g.n; m// the set of functions....?

Answer

First, we show that

1 + ϕ = 6+2√ 5 4 = ϕ 2 . So, for every i, ϕ i−1 + ϕ i−2 = ϕ i−2 (ϕ + 1) = ϕ i . Similarly for ϕˆ

. For i = 0, ϕ 0−ϕˆ0 √ 5 = 0. For i = 1, 1+√ 5 2 − 1− √ 5 √ 2 5 = √ √ 5 5 = 1. Then, by induction, Fi = Fi−1 + Fi−2 = ϕ i−1+ϕ i−2−(ϕˆi−1+ϕˆi−2) √ 5 = ϕ i−ϕˆi √ 5

3.3-1 Show that if f .n/ and g.n/ are monotonically increasing functions, then so are the functions f .n/ C g.n/ and f .g.n//, and if f .n/ and g.n/ are in addition nonnegative, then f .n/ g.n/ is monotonically increasing.

Answer

To prove that if f(n)f(n)f(n) and g(n)g(n)g(n) are monotonically increasing functions, then the functions f(n)+g(n)f(n) + g(n)f(n)+g(n) and f(g(n))f(g(n))f(g(n)) are also monotonically increasing, and if f(n)f(n)f(n) and g(n)g(n)g(n) are nonnegative, then f(n)·g(n)f(n) \cdot g(n)f(n)·g(n) is monotonically increasing, we begin by recalling the definition of a monotonically increasing function. A function f(n)f(n)f(n) is monotonically increasing if for any n1≤n2n_1 \leq n_2n1≤n2, it holds that f(n1)≤f(n2)f(n_1) \leq f(n_2)f(n1)≤f(n2).

Given that f(n)f(n)f(n) and g(n)g(n)g(n) are monotonically increasing, for any n1≤n2n_1 \leq n_2n1 ≤n2, we have f(n1)≤f(n2)f(n_1) \leq f(n_2)f(n1)≤f(n2) and g(n1)≤g(n2)g(n_1) \leq g(n_2)g(n1)≤g(n2). Now consider the sum f(n)+g(n)f(n) + g(n)f(n)+g(n). For n1≤n2n_1 \leq n_2n1≤n2, we can write:

f(n1)+g(n1)≤f(n2)+g(n2)f(n_1) + g(n_1) \leq f(n_2) + g(n_2)f(n1)+g(n1)≤f(n2)+g(n2)

This demonstrates that f(n)+g(n)f(n) + g(n)f(n)+g(n) is also monotonically increasing.

Next, we examine the composition f(g(n))f(g(n))f(g(n)). Since g(n)g(n)g(n) is monotonically increasing, we know that for n1≤n2n_1 \leq n_2n1≤n2:

g(n1)≤g(n2)g(n_1) \leq g(n_2)g(n1)≤g(n2)

Applying the monotonically increasing property of f(n)f(n)f(n) to the outputs of g(n)g(n)g(n) gives us:

f(g(n1))≤f(g(n2))f(g(n_1)) \leq f(g(n_2))f(g(n1))≤f(g(n2))

Thus, f(g(n))f(g(n))f(g(n)) is also monotonically increasing.

Now we consider the product f(n)·g(n)f(n) \cdot g(n)f(n)·g(n) under the condition that both functions are nonnegative. Since f(n)f(n)f(n) and g(n)g(n)g(n) are nonnegative and monotonically increasing, for n1≤n2n_1 \leq n_2n1≤n2, we have f(n1)≤f(n2)f(n_1) \leq f(n_2)f(n1)≤f(n2) and g(n1)≤g(n2)g(n_1) \leq g(n_2)g(n1)≤g(n2). We want to show that:

f(n1)·g(n1)≤f(n2)·g(n2)f(n_1) \cdot g(n_1) \leq f(n_2) \cdot g(n_2)f(n1)·g(n1)≤f(n2)·g(n2)

We can use the property of nonnegative functions. Since g(n2)≥g(n1)g(n_2) \geq g(n_1)g(n2)≥g(n1), we have:

f(n1)·g(n2)≥f(n1)·g(n1)f(n_1) \cdot g(n_2) \geq f(n_1) \cdot g(n_1)f(n1)·g(n2)≥f(n1)·g(n1)

Also, since f(n2)≥f(n1)f(n_2) \geq f(n_1)f(n2)≥f(n1), we can write:

f(n2)·g(n1)≥f(n1)·g(n1)f(n_2) \cdot g(n_1) \geq f(n_1) \cdot g(n_1)f(n2)·g(n1)≥f(n1)·g(n1)

Therefore, combining these two results, we observe that:

f(n2)·g(n2)≥f(n2)·g(n1)andf(n2)·g(n2)≥f(n1)·g(n2)f(n_2) \cdot g(n_2) \geq f(n_2) \cdot g(n_1) \quad \text{and} \quad f(n_2) \cdot g(n_2) \geq f(n_1) \cdot g(n_2)f(n2)·g(n2)≥f(n2)·g(n1)andf(n2)·g(n2)≥f(n1)·g(n2)

Thus, f(n1)·g(n1)≤f(n2)·g(n2)f(n_1) \cdot g(n_1) \leq f(n_2) \cdot g(n_2)f(n1)·g(n1)≤f(n2)·g(n2) holds true, proving that f(n)·g(n)f(n) \cdot g(n)f(n)·g(n) is monotonically increasing.

In summary, we have shown that if f(n)f(n)f(n) and g(n)g(n)g(n) are monotonically increasing, then both f(n)+g(n)f(n) + g(n)f(n)+g(n) and f(g(n))f(g(n))f(g(n)) are monotonically increasing. Moreover, if f(n)f(n)f(n) and g(n)g(n)g(n) are also nonnegative, then f(n)·g(n)f(n) \cdot g(n)f(n)·g(n) is monotonically increasing.

3.3-2 Prove that b ¸ncCd.1 ¸/ne D n for any integer n and real number ¸ in the range 0 ≤ ¸ ≤ 1.

Answer

To prove that

⌊αn⌋+⌈(1−α)n⌉=n\lfloor \alpha n \rfloor + \lceil (1 - \alpha) n \rceil = n⌊αn⌋+⌈(1−α)n⌉=n

for any integer nnn and real number α\alphaα in the range 0≤α≤10 \leq \alpha \leq 10≤α≤1, we start by defining ⌊αn⌋\lfloor \alpha n \rfloor⌊αn⌋ and ⌈(1−α)n⌉\lceil (1 - \alpha) n \rceil⌈(1−α)n⌉.

Let αn=k+f\alpha n = k + fαn=k+f, where k=⌊αn⌋k = \lfloor \alpha n \rfloork=⌊αn⌋ is the integer part and fff is the fractional part such that 0≤f<10 \leq f < 10≤f<1. Thus, we can express:

⌊αn⌋=k\lfloor \alpha n \rfloor = k⌊αn⌋=k

Now, consider the term (1−α)n(1 - \alpha) n(1−α)n:

(1−α)n=n−αn=n−(k+f)=(n−k)−f(1 - \alpha) n = n - \alpha n = n - (k + f) = (n - k) - f(1−α)n=n−αn=n−(k+f)=(n−k)−f

Next, we analyze ⌈(1−α)n⌉\lceil (1 - \alpha) n \rceil⌈(1−α)n⌉:

⌈(1−α)n⌉=⌈(n−k−f)⌉\lceil (1 - \alpha) n \rceil = \lceil (n - k - f) \rceil⌈(1−α)n⌉=⌈(n−k−f)⌉

The value of (n−k−f)(n - k - f)(n−k−f) is between n−k−1n - k - 1n−k−1 and n−kn - kn−k. Thus, we have:

- If f=0f = 0f=0, then (1−α)n(1 - \alpha) n(1−α)n is an integer, and we find:

⌈(1−α)n⌉=n−k\lceil (1 - \alpha) n \rceil = n - k⌈(1−α)n⌉=n−k

- If f>0f > 0f>0, then (1−α)n(1 - \alpha) n(1−α)n is not an integer, and we find:

⌈(1−α)n⌉=n−k\lceil (1 - \alpha) n \rceil = n - k⌈(1−α)n⌉=n−k

In both cases, we can see that ⌈(1−α)n⌉=n−k\lceil (1 - \alpha) n \rceil = n - k⌈(1−α)n⌉=n−k.

Now, we can combine both results:

⌊αn⌋+⌈(1−α)n⌉=k+(n−k)=n\lfloor \alpha n \rfloor + \lceil (1 - \alpha) n \rceil = k + (n - k) = n⌊αn⌋+⌈(1−α)n⌉=k+(n−k)=n

This shows that

⌊αn⌋+⌈(1−α)n⌉=n\lfloor \alpha n \rfloor + \lceil (1 - \alpha) n \rceil = n⌊αn⌋+⌈(1−α)n⌉=n

is valid for any integer nnn and real number α\alphaα in the range 0≤α≤10 \leq \alpha \leq 10≤α≤1. Thus, the proof is complete.

3.3-3 Use equation (3.14) or other means to show that .n C o.n//k D ,.nk / for any real constant k. Conclude that dne k D ,.nk / and bnc k D ,.nk /.

Answer

To show that (n+o(n))k=Θ(nk)(n + o(n))^k = \Theta(n^k)(n+o(n))k=Θ(nk) for any real constant kkk, we will use the binomial theorem and properties of o(n)o(n)o(n).

We start by expanding (n+o(n))k(n + o(n))^k(n+o(n))k using the binomial theorem:

(n+o(n))k=∑i=0k(ki)nk−io(n)i(n + o(n))^k = \sum_{i=0}^{k} \binom{k}{i} n^{k-i} o(n)^i(n+o(n))k=i=0∑k(ik)nk−io(n)i

Here, (ki)\binom{k}{i}(ik) is the binomial coefficient. We analyze the terms in the sum:

1. When $i=0$i = 0i=0:

$(k0)nko(n)0=nk\binom{k}{0} n^k o(n)^0 = n^k(0k)nko(n)0=nk$

2. When $i=1$i = 1i=1:

$(k1)nk−1o(n)=knk−1o(n)\binom{k}{1} n^{k-1} o(n) = k n^{k-1} o(n)(1k)nk−1o(n)=knk−1o(n)$

3. For $i≥2$i \geq 2i≥2:

$(ki)nk−io(n)i\binom{k}{i} n^{k-i} o(n)^i(ik)nk−io(n)i$

Each of these terms will have $nk−in^{k-i}nk−i$ multiplied by $o(n)io(n)^io(n)i$, which will be smaller than $nk−in^{k-i}nk−i$.

Next, we know that $o(n)o(n)o(n)$ denotes a function that grows slower than nnn. This means that:

$o(n)=n·ϵ(n)whereϵ(n)→0 as n→∞o(n) = n \cdot \epsilon(n) \quad \text{where} \quad \epsilon(n) \to 0 \text{ as } n \to \inftyo(n)=n·ϵ(n)whereϵ(n)→0 as n→∞$

From this, we see that for large nnn, $o(n)o(n)o(n)$ will not affect the leading term of the polynomial when raised to any fixed power kkk.

As $n→∞n \to \inftyn→∞$, the dominant term in the expansion is nkn^knk. Therefore, we can say:

$(n+o(n))k∼nk(n + o(n))^k \sim n^k(n+o(n))k∼nk$

Thus, we have:

$(n+o(n))k=Θ(nk)(n + o(n))^k = \Theta(n^k)(n+o(n))k=Θ(nk)$

This shows that:

$(n+o(n))k=Θ(nk)(n + o(n))^k = \Theta(n^k)(n+o(n))k=Θ(nk)$

From this, we can also conclude:

1. For $d(n)=n+o(n)d(n) = n + o(n)d(n)=n+o(n)$:

$d(n)k=Θ(nk)d(n)^k = \Theta(n^k)d(n)k=Θ(nk)$

2. For $⌊n⌋k\lfloor n \rfloor^k⌊n⌋k$:

$⌊n⌋k=Θ(nk)\lfloor n \rfloor^k = \Theta(n^k)⌊n⌋k=Θ(nk)$

Thus, both $d(n)kd(n)^kd(n)k$ and $⌊n⌋k\lfloor n \rfloor^k⌊n⌋k$ can be expressed as $Θ(nk)\Theta(n^k)Θ(nk)$.

Overall, we conclude that:

$(n+o(n))k=Θ(nk),d(n)k=Θ(nk),and⌊n⌋k=Θ(nk).(n + o(n))^k = \Theta(n^k), \quad d(n)^k = \Theta(n^k), \quad \text{and} \quad \lfloor n \rfloor^k = \Theta(n^k).(n+o(n))k=Θ(nk),d(n)k=Θ(nk),and⌊n⌋k=Θ(nk).$

3.3-4 Prove the following:

a. Equation (3.21).

b. Equations (3.26)3(3.28).

c. lg.,.n// D ,.lg n/.

Answer

Equation (3.21) states that

Θ(nk) for k∈R means ∃c1,c2>0,n0 such that c1nk≤f(n)≤c2nk for all n≥n0.\Theta(n^k) \text{ for } k \in \mathbb{R} \text{ means } \exists c_1, c_2 > 0, n_0 \text{ such that } c_1 n^k \leq f(n) \leq c_2 n^k \text{ for all } n \geq n_0.Θ(nk) for k∈R means ∃c1,c2>0,n0 such that c1nk≤f(n)≤c2nk for all n≥n0.

To prove this, we start by recalling the definition of Θ\ThetaΘ:

1. If f(n)=Θ(nk)f(n) = \Theta(n^k)f(n)=Θ(nk), by definition, there exist positive constants c1c_1c1, c2c_2c2, and n0n_0n0 such that:

c1nk≤f(n)≤c2nkfor all n≥n0.c_1 n^k \leq f(n) \leq c_2 n^k \quad \text{for all } n \geq n_0.c1 nk≤f(n)≤c2nkfor all n≥n0.

2. This definition captures that f(n)f(n)f(n) grows asymptotically as nkn^knk, bounded above and below by constant multiples of nkn^knk.

3. Thus, the assertion in Equation (3.21) is proven, showing the relationship of f(n)f(n)f(n) with nkn^knk.

b. Prove Equations (3.26) to (3.28)

Equation (3.26) states that:

logab=logcblogca\log_a b = \frac{\log_c b}{\log_c a}logab=logcalogcb

To prove this, we start by using the definition of logarithms.

1. Let x=logabx = \log_a bx=logab. This means:

ax=ba^x = bax=b

2. Taking logarithm base ccc on both sides:

logc(ax)=logcb\log_c (a^x) = \log_c blogc(ax)=logcb

3. By the power rule of logarithms, we have:

x⋅logca=logcbx \cdot \log_c a = \log_c bx⋅logca=logcb

4. Rearranging gives:

x=logcblogcax = \frac{\log_c b}{\log_c a}x=logcalogcb

5. Hence, we have shown that logab=logcblogca\log_a b = \frac{\log_c b}{\log_c a}logab=logcalogcb, proving Equation (3.26).

Now, for Equations (3.27) and (3.28):

- Equation (3.27) states:

log⁡a(b)+log⁡b(c)=log⁡a(c)\log_a(b) + \log_b(c) = \log_a(c)loga(b)+logb(c)=loga(c)

1. Using the definition of logarithms, we let:

x=log⁡a(b)andy=log⁡b(c).x = \log_a(b) \quad \text{and} \quad y = \log_b(c).x=loga(b)andy=logb(c).

2. From the definitions, we can express bbb and ccc in terms of aaa:

ax=bandby=c.a^x = b \quad \text{and} \quad b^y = c.ax=bandby=c.

3. Substituting bbb from the first equation into the second gives:

(ax)y=c ⟹ axy=c.(a^x)^y = c \implies a^{xy} = c.(ax)y=c⟹axy=c.

4. Therefore, by the definition of logarithms:

log⁡a(c)=xy=log⁡a(b)⋅log⁡b(c).\log_a(c) = xy = \log_a(b) \cdot \log_b(c).loga(c)=xy=loga(b)⋅logb(c).

5. Hence, we have shown log⁡a(b)+log⁡b(c)=log⁡a(c)\log_a(b) + \log_b(c) = \log_a(c)loga(b)+logb(c)=loga(c).

- Equation (3.28) states:

log⁡a(a)=1.\log_a(a) = 1.loga(a)=1.

1. By definition, if x=log⁡a(a)x = \log_a(a)x=loga(a), then:

ax=a.a^x = a.ax=a.

2. The only solution is x=1x = 1x=1. Thus, log⁡a(a)=1\log_a(a) = 1loga(a)=1.

This concludes the proof of Equations (3.26) through (3.28).

c. Prove log⁡(Θ(n))=Θ(log⁡n)\log(\Theta(n)) = \Theta(\log n)log(Θ(n))=Θ(logn)

To show this, we start with the definition of Θ(n)\Theta(n)Θ(n).

1. If f(n)=Θ(n)f(n) = \Theta(n)f(n)=Θ(n), then there exist constants c1,c2>0c_1, c_2 > 0c1,c2>0 and n0n_0n0 such that:

c1n≤f(n)≤c2nfor all n≥n0.c_1 n \leq f(n) \leq c_2 n \quad \text{for all } n \geq n_0.c1n≤f(n)≤c2 nfor all n≥n0.

2. Taking logarithms of the inequalities gives:

log⁡(c1n)≤log⁡(f(n))≤log⁡(c2n).\log(c_1 n) \leq \log(f(n)) \leq \log(c_2 n).log(c1n)≤log(f(n))≤log(c2 n).

3. Using properties of logarithms, we can express this as:

log⁡(c1)+log⁡(n)≤log⁡(f(n))≤log⁡(c2)+log⁡(n).\log(c_1) + \log(n) \leq \log(f(n)) \leq \log(c_2) + \log(n).log(c1)+log(n)≤log(f(n))≤log(c2)+log(n).

4. This implies:

log⁡(n)+log⁡(c1)≤log⁡(f(n))≤log⁡(n)+log⁡(c2).\log(n) + \log(c_1) \leq \log(f(n)) \leq \log(n) + \log(c_2).log(n)+log(c1)≤log(f(n))≤log(n)+log(c2).

5. Thus, we have established that:

log⁡(f(n))=Θ(log⁡n)\log(f(n)) = \Theta(\log n)log(f(n))=Θ(logn)

This proves that log⁡(Θ(n))=Θ(log⁡n)\log(\Theta(n)) = \Theta(\log n)log(Θ(n))=Θ(logn).

In summary, we have proven:

- Equation (3.21) about the definition of Θ\ThetaΘ.
- Equations (3.26) to (3.28) regarding logarithmic identities.
- The relationship log⁡(Θ(n))=Θ(log⁡n)\log(\Theta(n)) = \Theta(\log n)log(Θ(n))=Θ(logn).

3.3-5 Is the function dlg neŠ polynomially bounded? Is the function dlg lg neŠ polynomially bounded?

Answer

To determine if the functions dlog⁡nd \log ndlogn and dlog⁡log⁡nd \log \log ndloglogn are polynomially bounded, we need to consider the definition of polynomially bounded functions. A function f(n)f(n)f(n) is polynomially bounded if there exists a polynomial p(n)p(n)p(n) such that f(n)f(n)f(n) grows at a rate no faster than p(n)p(n)p(n) as n→∞n \to \inftyn→∞. This means that for large nnn, f(n)f(n)f(n) must be less than or equal to p(n)p(n)p(n).

Analyzing dlog⁡nd \log ndlogn

1. **Function Definition**: The function f(n)=dlog⁡nf(n) = d \log nf(n)=dlogn grows logarithmically with respect to nnn.
2. **Comparison with Polynomials**: To check if dlog⁡nd \log ndlogn is polynomially bounded, we compare it with polynomials of the form nkn^knk for some constant kkk.

As n→∞,log⁡n→∞, but much slower than any polynomial nk.\text{As } n \to \infty, \log n \to \infty, \text{ but much slower than any polynomial } n^k.As n→∞,logn→∞, but much slower than any polynomial nk.

3. **Conclusion**: Since dlog⁡nd \log ndlogn grows slower than any polynomial (because log⁡n\log nlogn grows slower than nkn^knk for any k>0k > 0k>0), we conclude that:

dlog⁡n is polynomially bounded.d \log n \text{ is polynomially bounded.}dlogn is polynomially bounded.

Analyzing dlog□log□nd \log \log ndloglogn

1. **Function Definition**: The function g(n)=dlog□log□ng(n) = d \log \log ng(n)=dloglogn grows even slower than dlog□nd \log ndlogn.
2. **Comparison with Polynomials**: To check if dlog□log□nd \log \log ndloglogn is polynomially bounded, we again compare it with polynomials.

As n→∞,log□log□n also grows very slowly compared to any polynomial nk.\text{As } n \to \infty, \log \log n \text{ also grows very slowly compared to any polynomial } n^k.As n→∞,loglogn also grows very slowly compared to any polynomial nk.

? 3.3-6 Which is asymptotically larger: lg.lg □ n/ or lg □ .lg n/?\

Answer

The two functions log□(log□n)\log(\log n)log(logn) and log□(log□n)\log(\log n)log(logn) are actually the same, so there is no difference between them in terms of growth rate. Since both functions are identical, they grow at the same rate for large values of nnn. Therefore, neither function is asymptotically larger than the other; they are equal in their asymptotic behavior.

3.3-7 Show that the golden ratio **□** and its conjugate **□**y both satisfy the equation x 2 D x C 1.

Answer

To show that the golden ratio ϕ\phiϕ and its conjugate ψ\psiψ satisfy the equation x2=x+1x^2 = x + 1x2=x+1, we first need to define the golden ratio and its conjugate.

The golden ratio ϕ\phiϕ is defined as:

ϕ=1+52\phi = \frac{1 + \sqrt{5}}{2}ϕ=21+5

And its conjugate ψ\psiψ is defined as:

ψ=1−52\psi = \frac{1 - \sqrt{5}}{2}ψ=21−5

Step 1: Show that ϕ\phiϕ satisfies x2=x+1x^2 = x + 1x2=x+1

1. Substitute ϕ\phiϕ into the left side of the equation:

ϕ2=(1+52)2\phi^2 = \left(\frac{1 + \sqrt{5}}{2}\right)^2ϕ2=(21+5)2

2. Calculate ϕ2\phi^2ϕ2:

ϕ2=(1+5)24=1+25+54=6+254=3+52\phi^2 = \frac{(1 + \sqrt{5})^2}{4} = \frac{1 + 2\sqrt{5} + 5}{4} = \frac{6 + 2\sqrt{5}}{4} = \frac{3 + \sqrt{5}}{2}ϕ2=4(1+5)2=41+25+5=46+25=23+5

3. Now calculate ϕ+1\phi + 1ϕ+1:

$\phi+1=1+\frac{1+\sqrt{5}}{2}+1=\frac{1+\sqrt{5}}{2}+\frac{2}{2}=\frac{3+\sqrt{5}}{2}\phi+1=21+5+1=21+5+22=3+5$

4. We see that:

$\phi^2=\phi+1\phi^2=\phi+1$

Thus, ϕ satisfies the equation $x^2=x+1x^2=x+1x2=x+1$.

Step 2: Show that ψ satisfies $x^2=x+1x^2=x+1x2=x+1$

1. Substitute ψ into the left side of the equation:

$\psi^2=\left(\frac{1-\sqrt{5}}{2}\right)^2\psi2=(21-5)2$

2. Calculate $\psi^2\psi2$:

$\psi^2=\frac{(1-\sqrt{5})^2}{4}=\frac{1-2\sqrt{5}+5}{4}=\frac{6-2\sqrt{5}}{4}=\frac{3-\sqrt{5}}{2}\psi2=4(1-5)2=41-25+5=46-25=23-5$

3. Now calculate $\psi+1$:

$\psi+1=1-\frac{1-\sqrt{5}}{2}+1=\frac{1-\sqrt{5}}{2}+\frac{2}{2}=\frac{3-\sqrt{5}}{2}\psi+1=21-5+1=21-5+22=23-5$

4. We see that:

$\psi^2=\psi+1\psi^2=\psi+1$

Thus, ψ also satisfies the equation $x^2=x+1x^2=x+1x2=x+1$.

3.3-8 Prove by induction that the ith Fibonacci number satisûes the equation

Answer

To prove by induction that the iii-th Fibonacci number $F(i)F(i)F(i)$ satisfies the equation $F(i)=F(i-1)+F(i-2)F(i)=F(i-1)+F(i-2)F(i)=F(i-1)+F(i-2)$ for $i\geq 2i \geq 2i\geq 2$, we start with the base cases. For $i=0i=0i=0$, $F(0)=0F(0)=0F(0)=0$ and for $i=1i=1i=1$, $F(1)=1F(1)=1F(1)=1$, and both satisfy the Fibonacci definition. Now, assume the equation holds for $i=ki=ki=k$ and $i=k-1i=k-1i=k-1$, meaning $F(k)=F(k-1)+F(k-2)F(k)=F(k-1)+F(k-2)F(k)=F(k-1)+F(k-2)$ and $F(k-1)=F(k-2)+F(k-3)F(k-1)=F(k-2)+F(k-3)F(k-1)=F(k-2)+F(k-3)$. For $i=k+1i=k+1i=k+1$, we can express it as:

$F(k+1)=F(k)+F(k-1)F(k+1)=F(k)+F(k-1)F(k+1)=F(k)+F(k-1)$

Using the induction hypothesis, we substitute $F(k)=F(k-1)+F(k-2)F(k)=F(k-1)+F(k-2)F(k)=F(k-1)+F(k-2)$:

$F(k+1)=(F(k-1)+F(k-2))+F(k-1)=2F(k-1)+F(k-2)F(k+1)=(F(k-1)+F(k-2))+F(k-1)=2F(k-1)+F(k-2)F(k+1)=(F(k-1)+F(k-2))+F(k-1)=2F(k-1)+F(k-2)$

This shows that F(k+1)F(k + 1)F(k+1) can be derived from the previous Fibonacci numbers, proving that the relation holds for i=k+1i = k + 1i=k+1. Therefore, by induction, the iii-th Fibonacci number satisfies the equation for all i≥2i \geq 2i≥2.

Problems

3.-1

Answer

a. If we pick any c > 0, then, the end behavior of cnk −p(n) is going to infinity, in particular, there is an n0 so that for every n ≥ n0, it is positive, so, we can add p(n) to both sides to get p(n) < cnk .

b. If we pick any c > 0, then, the end behavior of p(n)−cnk is going to infinity, in particular, there is an n0 so that for every n ≥ n0, it is positive, so, we can add cnk to both sides to get p(n) > cnk .

c. We have by the previous parts that p(n) = O(n k) and p(n) = Ω(n k). So, by Theorem 3.1, we have that p(n) = Θ(n k).

d. limn→∞ p(n) nk = limn→∞ n d (ad + o(1)) nk < limn→∞ 2adn d nk = 2ad limn→∞ n d−k =

e. limn→∞ n k p(n) = limn→∞ n k ndO(1) < limn→∞ n k nd = limn→∞ n k−d =

3.-2

Answer

Let αn=k+f\alpha n = k + fαn=k+f and (1−α)n=(n−k)−f(1 - \alpha) n = (n - k) - f(1−α)n=(n−k)−f, where k=⌊αn⌋k = \lfloor \alpha n \rfloork=⌊αn⌋ is the integer part of αn\alpha nαn, and fff is the fractional part, such that 0≤f<10 \leq f < 10≤f<1.

Since k=⌊αn⌋k = \lfloor \alpha n \rfloork=⌊αn⌋, by definition we have ⌊αn⌋=k\lfloor \alpha n \rfloor = k⌊αn⌋=k. Now, consider ⌈(1−α)n⌉\lceil (1 - \alpha) n \rceil⌈(1−α)n⌉: since (1−α)n=(n−k)−f(1 - \alpha) n = (n - k) - f(1−α)n=(n−k)−f, the smallest integer greater than or equal to (1−α)n(1 - \alpha) n(1−α)n is n−kn - kn−k. Therefore, ⌈(1−α)n⌉=n−k\lceil (1 - \alpha) n \rceil = n - k⌈(1−α)n⌉=n−k.

Adding these two expressions gives:

⌊αn⌋+⌈(1−α)n⌉=k+(n−k)=n

Thus, we have shown that ⌊αn⌋+⌈(1−α)n⌉=n\lfloor \alpha n \rfloor + \lceil (1 - \alpha) n \rceil = n⌊αn⌋+⌈(1−α)n⌉=n for any integer nnn and any real number α\alphaα in the range 0≤α≤10 \leq \alpha \leq 10≤α≤1. This completes the proof.

3.-3

Answer

To prove that (n+o(n))k=Θ(nk)(n + o(n))^k = \Theta(n^k)(n+o(n))k=Θ(nk) for any real constant kkk, we start by analyzing the term (n+o(n))k(n + o(n))^k(n+o(n))k

Since o(n)o(n)o(n) is defined as a function that grows slower than nnn, we can write o(n)=f(n)o(n) = f(n)o(n)=f(n) where f(n)→0f(n) \to 0f(n)→0 as n→∞n \to \infty n→∞. This means that for large nnn, o(n)o(n)o(n) becomes negligible compared to nnn. Now, we can use the binomial expansion to expand (n+o(n))k(n + o(n))^k(n+o(n))k:

(n+o(n))k=nk+(k1)nk−1⋅o(n)+(k2)nk−2⋅o(n)2+⋯+o(n)k(n + o(n))^k = n^k + \binom{k}{1} n^{k-1} \cdot o(n) + \binom{k}{2} n^{k-2} \cdot o(n)^2 + \dots + o(n)^k(n+o(n))k=nk+(1k)nk−1⋅o(n)+(2k)nk−2⋅o(n)2+⋯+o(n)k

3.-4

Answer

A False. Counterexample: n = O(n 2) but n 2 6= O(n).

b. False. Counterexample: n + n 2 6= Θ(n).

c. True. Since f(n) = O(g(n)) there exist c and n0 such that n ≥ n0 implies f(n) ≤ cg(n) and f(n) ≥ 1. This means that log(f(n)) ≤ log(cg(n)) = log(c) + log(g(n)). Note that the inequality is preserved after taking logs because f(n) ≥ 1. Now we need to find d such that log(f(n)) ≤ d log(g(n)). It will suffice to make log(c) + log(g(n)) ≤ d log(g(n)), which is achieved by taking d = log(c) + 1, since log(g(n)) ≥ 1.

d. False. Counterexample: 2n = O(n) but 22n 6= 2n as shown in exercise 3.1-4.

e. False. Counterexample: Let f(n) = 1 n . Suppose that c is such that 1 n ≤ c 1 n2 for n ≥ n0. Choose k such that kc ≥ n0 and k > 1. Then this implies 1 kc ≤ c k2c 2 = 1 k2c , a contradiction.

f. True. Since f(n) = O(g(n)) there exist c and n0 such that n ≥ n0 implies f(n) ≤ cg(n). Thus g(n) ≥ 1 c f(n), so g(n) = Ω(f(n)).

g. False. Counterexample: Let f(n) = 22n. By exercise 3.1-4, 22n 6= O(2n).

h. True. Let g be any function such that g(n) = o(f(n)). Since g is asymptotically positive let n0 be such that n ≥ n0 implies g(n) ≥ 0. Then f(n) + g(n) ≥ f(n) so f(n) + o(f(n)) = Ω(f(n)). Next, choose n1 such that n ≥ n1 implies g(n) ≤ f(n). Then f(n) + g(n) ≤ f(n) + f(n) = 2f(n) so f(n) +o(f(n)) = O(f(n)). By Theorem 3.1, this implies f(n) +o(f(n)) = Θ(f(n)). 7

3.5

Answer

a. Suppose that we do not have that f = O(g(n)). This means that ∀c > 0, n0, ∃n ≥ n0, f(n) > cg(n). Since this holds for every c, we can let it be arbitrary, say 1. Initially, we set n0 = 1, then, the resulting n we will call a1. Then, in general, let n0 = ai + 1 and let ai+1 be the resulting value of n. Then, on the infinite set {a1, a2, . . .}, we have f(n) > g(n), and so, f = ∞ Ω(g(n)) This is not the case for the usual definition of Ω. Suppose we had f(n) = n 2 (n mod 2) and g(n) = n. On all the even values, g(n) is larger, but on all the odd values, f(n) grows more quickly.

b. The advantage is that you get the result of part a which is a nice property. A disadvantage is that the infinite set of points on which you are making claims of the behavior could be very sparse. Also, there is nothing said about the behavior when outside of this infinite set, it can do whatever it wants.

c. A function f can only be in Θ(g(n)) if f(n) has an infinite tail that is non– negative. In this case, the definition of O(g(n)) agrees with O0 (g(n)). Similarly, for a function to be in Ω(g(n)), we need that f(n) is non-negative for some infinite tail, on which O(g(n)) is identical to O0 (g(n)). So, we have that in both directions, changing O to O0 does not change anything.

d. Suppose f(n) ∈ ∼ Θ(g(n)), then ∃c1, c2, k1, k2, n0, ∀n ≥ n0, 0 ≤ c1g(n) lgk1 (n) ≤ f(n) ≤ c2g(n) lgk2 (n), if we just look at these inequalities separately, we have c1g(n) lgk1 (n) ≤ f(n) (f(n) ∈ ∼ Ω(g(n))) and f(n) ≤ c2g(n) lgk2 (n) (f(n) ∈ ∼ O(g(n))). Now for the other direction. Suppose that we had ∃n1, c1, k1∀n ≥ n1, c1g(n) lgk1 (n) ≤ f(n) and ∃n2, c2, k2, ∀n ≥ n2, f(n) ≤ c2g(n) lgk2 (n). Putting these together, and letting n0 = max(n1, n2), we have ∀n ≥ n0, c1g(n) lgk1 (n) ≤ f(n) ≤ c2g(n) lgk2 (n).

3.6

Answer

F(n)	c	f ∗ c (n)
n - 1	0	[n]
Log n	1	log∗ n
n/2	1	[log(n)]
n/2	2	[log(n)]-1
√ n	2	lognlogn
√ n	1	undefined
n/ log n	2	Ω log n log(log n)