A1 ICCS 312 Algorithms and Tractability

ALEXANDER OGAY 6380727

April 28, 2024

1 Problem 1

(a) $3n^3 + 75n^2 + 8log_2n \in \mathcal{O}(n^3)$

1. $3n^3 + 75n^2 + 8log_2n \leq c \times n^3$
2. if c = 7 and $n_0 = 20$
3. $54034 \leq 56000$
4. $f(n) \in \mathcal{O}(n^3)$

(b) $1 + 3 + 5 + ... + (2n - 1) \in \mathcal{O}(n^2)$

1. $1 + 3 + 5 + ... + (2n - 1) = n^2$
2. if c = 2 and $n_0 = 1$
3. $1 \leq 2$
4. $f(n) \in \mathcal{O}(n^2)$

(c) Show that $1 + 2 + 4 + 8 + ... + 2^n$ **is** $\in \mathcal{O}(2^n)$

1. $1 + 2 + 4 + 8 + ... + 2^n = 2^{n+1} - 1$
2. if c = 3 and $n_0 = 1$
3. $3 \leq 6$
4. $f(n) \in \mathcal{O}(2^n)$

(d) Show that $1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^n}$ **is** $\in \mathcal{O}(1)$

1. $1 + \frac{1}{2} + \frac{1}{4} + ... + \frac{1}{2^n} = \frac{2^{n+1}-1}{2^n}$
2. if c = 3 and $n_0 = 1$
3. $1.5 \leq 3$
4. $f(n) \in \mathcal{O}(1)$

2 Problem 2

(a) We need to show that BUBBLESORT swaps the positions of two elements in the array if one is smaller than the other. We also need to show that there are no problems looping through i and j, such as trying to loop to an unreachable index. Furthermore BUBBLESORT has to go through every single element in the list and make sure that the array is sorted in the correct order.

(b) Initialization, loop j has to start from the last element of the array and goes all the way to the 2nd element, and then also checks the first element because of $A[j] - A[j-1]$, this means that the loop checks every element of the array in the beginning, and constantly carries the lowest value element to the beginning of the array because of line 4, meaning the first loop correctly sorts the lowest value element in the right location, making this true.

Maintenance, looping each time afterwards would make the array become sorted because the lowest values would keep being pushed towards the beginning while consequently the largest values would switch places and go towards the top, as we assume the i value is not updated, this would mean the loop would go through all elements each time, but this does not negatively affect the sorting of the algorithm, meaning this is true, but this only happens if the i value is updated therefore if we assume j only loops through once then we don't have to discuss this as the loop only goes 1 time.

Termination, if the i value is not updated then we can't believe that the whole array is sorted, but from the first loop we can believe that the first element is the lowest value element and if the i value is updated and the j loop restarted then the conditions show that there are not problems terminating, making this true.

(c) Initialization, the loop j goes through the whole array, meaning that the lowest value element has to go towards the beginning of the loop even if it has to be dragged all the way from the end of the loop, meaning the first element of the array would become correctly sorted, j=A.length down to i+1, i in the beginning is 1 (the first element), so j stops at 2 (i+1) but checks A[j-1], which is A[1] the first element.

Maintenance, the next loop then goes through the whole array once again from top to bottom but the i value is changed this time, meaning the sorted first element of the array is not touched, this would mean that the next lowest value would be continually pushed towards the beginning of the array until it hits the 2nd element, and once the i

value updates again after the loop, it would go to the 3rd element, continually checking the rest of the unsorted array and placing the lowest value onto the i value.

Termination, loop termination happens once i reaches the second last index of the array and j checks the last array element with the second last, and switches them if necessary, (i = A.length-1 is the last check that is allowed before termination, i < A.length) Seeing that the algorithm continuously pushed the lowest value towards the beginning and sorted it that way, the last values to be checked are the largest in the array, concluding that the algorithm is correct.

(d) The worst case running time of bubble sort is $\mathcal{O}(n^2)$, it has the same running time as insertion sort.

3 Problem 3

(a) $S(n) = 1^c + 2^c + 3^c + ... + n^c \in \mathcal{O}(n^{c+1})$

1. $1^c + 2^c + 3^c + ... + n^c = \frac{n(n+1)}{2}$
2. if c = 1 and $n_0 = 3$
3. $6 \leq 8$
4. $f(n) \in \mathcal{O}(n^{c+1})$

(b) $S(n) = 1^c + 2^c + 3^c + ... + n^c \in \Omega(n^{c+1})$

1. $1^c + 2^c + 3^c + ... + n^c = \frac{n(n+1)}{2}$
2. if c = 0 and $n_0 = 2$
3. $2 \leq 2$
4. $f(n) \in \Omega(n^{c+1})$

4 Problem 4

(a) $\mathcal{O}(logn)$

(b) $\mathcal{O}(logn)$

(c) $\mathcal{O}(n^3)$

5 Problem 5

(a) Algorithm

1. While A.length >= 2:
2. Check if the first element or the middle element are i, if so return true
3. Divide the array. If i is smaller than the middle element, go into the subarray from the first value to the middle element. If i is larger than the middle element, go into the subarray from the middle value to A.length
4. If it reaches the end of the loop, and the array is of size 1, check if the $A[0] == i$, else return false.

The algorithm checks half the array each time keeping the log n time, it makes sure that the index is within the array unless it is larger than the largest value, which is this case the termination of the loop would result in showing that the index doesn't exist.

(b) We could use the loop in (a), but add an if statement at the beginning that checks whether the index value is smaller than 1 **and whether the index value is larger than A[A.length], if either case is true, return false.**

6 Problem 6

(a)

1. make a variable [a] that holds A[1]
2. make a counter
3. for i=1 to A.length:
4. if A[i] >= variable [a], increase counter by 1, update variable (a) to A[i]
5. else stop the loop and return counter

(b)

1. check the middle element of the array, if the middle element is x, return true
2. check the size of the elements on either side and whichever side x is closer to, make a loop that checks that half of the array