Algorithms and Analysis Assignment 2

### Algorithm Design and Complexity Analysis

Name: Jared Song Student Number: s3857657

1. is true if there exists some positive constants and and some nonnegative integer such that for all .

Proving the right inequality (upper bound):

for all as

Proving the left inequality (lower bound):

for all as

Hence, we can select , and therefore the statement is true.

1. is true if

Apply L'Hôpital's rule:

. Therefore, the statement is true.

1. is true if

Apply L'Hôpital's rule:

. Therefore, the statement is false.

1. or
2. Description: A modified reverse-bubblesort that runs for iterations. This bubblesort will bring the smallest elements to the front of the array rather than the typical bubblesort which brings the largest element to the end of the array after every iteration. Because a bubblesort is a stable sorting algorithm, it will work even if there are repeated elements. Denote *A* as the array of real numbers of size *n*, and *m* number of smallest elements to be found.

|  |
| --- |
| **Algorithm** ModifiedBubbleSort(*A[0…n-1], m, n*) |
| **for** to **do**  **for** *j*down to *j* **do** // Start from the back of the array  **if then**  swap; //Move the smaller element towards the front of the array  **end if**  **end for**  **end for**  **for**  to **do** // Print the first m elements of the array  print ;  **end for** |

Complexity analysis: The first loop is a nested loop that runs very similarly to a conventional bubblesort except that the total number of iterations is *m*, not the size of the entire array. For each iteration, there are *n*-1 comparisons made. The last loop simply runs *m* times to display the first *m* indexed elements in the array. Thus

as .

1. Description: Now that we can use extra data structures, we can use a max heap to implement a priority queue. We can create a heap of *m* elements, then compare the root – the maximum value of the heap – with the rest of the elements in the array and dequeue the maximum value if a value in the array is smaller. In doing so, the size of the maxHeap is always *m* and effectively represents the smallest *m* elements in the list of iterated elements as the loop continues.

|  |
| --- |
| **Algorithm** MaxHeapSort(*A[0…n-1], m, n*) |
| **for**  to **do**  insert into maxHeap; // Insert the first *m* elements from the array into the minHeap  **end for**  **for** *j*to *j* **do** // (n-m-1) iterations  **if** maxHeap[0] > // The element in the array is smaller and should be in the heap  dequeue maxHeap; // Remove the maximum value in the heap  insert into maxHeap; // Insert a value smaller than the old maximum into the heap  **end if**  **end for**  print maxHeap; // Print the smallest *m* values from largest to smallest |

Complexity analysis: The first loop runs *m* times to get the first *m* elements from the array. Comparisons occur in the second loop, which runs *n-m-*1 times. The last operation of prints *m* elements from the heap. In the worst case, where the heap must be dequeued and a new element must be inserted every time, complexity is . Since log, the complexity is .

1. Description: An iterative algorithm that traverses the right-most path from the root of a binary tree to determine the largest key in the tree. The algorithm makes use of the BST’s ordering that all values in the right subtree are larger than the parent’s value and continues accessing the right subtree until a subtree does not exist. Denote *T* as the root node of a given binary search tree.

|  |
| --- |
| **Algorithm** BSTMax(*T*) |
| Node *current* = *T*; // Initialise the maximum value as the root node  **while** **do** // If there is a right subtree then continue  // Set the current node as the root of the right subtree  **end while**  **return** *current*; |

1. N/A
2. Description: Use an adjacency list to represent the list of *n* courses as vertexes and its dependent courses as edges, then run a Depth-First Search on the list. A dependent course refers to a course that requires another course as a pre-requisite. From *m* pairs of subjects (*i*,*j*) where *i* is a PRQ of *j*, *j* is the dependent course. Calculate the nonequivalent total credits for each course by traversing the entire adjacency list and adding each node’s credit points *ci* to the total accumulated credits of each edge. Denote *n* as the total number of courses with *m* pairs of subjects that have pre-requisites, and for each subject *i* its attributes: credit points *ci* and list of PRQ’s *Pi*. Additionally, denote *L* as the list of courses and *T* as the table of total credits.

|  |
| --- |
| **Algorithm** NonEquivalentPRQ(*L[0…n-1]*, *m, n, ci, Pi*, *T[0…n-1]*) |
| AdjacencyList *A* = new AdjacencyList(*n*); // Initialise a new adjacency list of size *n*  **for**  to **do**    *A*.addVertex(*)*;  **for** (course )**do** // For each course in the list of pre-requisites  *A*.addEdge; // Add an edge from *j* or the PRQ to *i* as *i* is the dependent course  **end for**  **end for**  **for**  to **do** // For every subject in the adjacency list  **for** (course )**do** // For each edgeor dependent course  ; // Add the PRQ subject’s credits to the dependent course’s total  **end for**  **end for** |

Complexity analysis: to create the adjacency list, we have the complexity as there are *n* subjects and *m* pairs of subjects to insert. The DFS of the adjacency list is completed *n* times to calculate the total credits for all courses and iterates over all the vertices and their corresponding edges, which requires complexity. Hence, the algorithm’s complexity is and can be simplified to .

Output table:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Course *i* | Adv. Prog. Tech. | Algo. Anal. | AI | Cloud Comp. | Comp. Theory | Discrete Struct. | Further Prog. | OS Principles | Prog. Fund. | Prog. Tech. | Soft. Eng. Fund. | Soft. Eng. P&T |
| Required accumulated credits T*i* | 6 | 24 | 36 | 24 | 48 | 0 | 6 | 18 | 0 | 0 | 18 | 36 |

1. Description: A top-down dynamic programming algorithm with a recurrence formula and backtrace that gives the sequence of courses the student must take as pre-requisites for a given course, and the minimum number of credits required to enroll in the course if the PRQ’s are equivalent. This algorithm requires the global arrays *i*[*0…n-1*] for course indexes, *ci*[*0…n-1*] for course credits for each course *i* and table F[*0…n-1, 0…n-1*] initialised with -1’s at all F[*i,i*].

|  |
| --- |
| **Algorithm** EquivalentPRQ(*i,Pi*) |
| **if**  **then** // The value has not been calculated  **if**  **then** // If the course has no pre-requisites  ; // Then  **else**  **for** (course )**do** // For each pre-requisite course  **if**  **then** // If the pre-requisite course has no pre-requisites  ; // Compare *x* with the credits of course *j*  e**lse**  ; // Complete the recurrence relation to find *x*  **end if**  **end for**  **end if**  ; // Update the dynamic programming table  **end if**  ; // Update the output table  **return** ; |

Complexity analysis: The first if-else condition checks if a course has a PRQ and if that number is zero, the algorithm runs for all subjects in linear time or . If a course has re-requisite, another if-else condition is entered – checking if the PRQ has its own PRQ’s. In the worst case, to complete a backtrace and find the optimal or minimum number of total credits for a given course, the complexity is . Since the graph is acyclic and has no loops, we can write the equation . Thus, the algorithm’s complexity is .

Recurrence formula: The algorithm is recursive when any courses pre-requisite has additional pre-requisites. This is to distinguish between cases where a course without PRQ’s is searched initially, or if it is called in the recurrence relation. If a PRQ has its own PRQ’s then the recurrence relation is called for course *j* and its PRQ’s, to find the minimum among back-traced values. Consequently, the algorithm will continue to function recursively for as many PRQ’s a course has in total, including the PRQ’s own list of PRQ’s and so on.

Backtrace: Following a top-down procedure rather than bottom-up which would compute all sub-problems, this algorithm computes the subset of *m* sub-problems for each pair of courses that have the input course *i* as a dependent course. For example, the course with the most sub-problems is Computing Theory. This course has four sets of sub-problems to solve: the minimum credits for Algorithms and Analysis, Further Programming, Programming Techniques and Discrete Structures. From , we compute and , then then lastly to get a minimum value of 6.

Output table:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Course *i* | Adv. Prog. Tech. | Algo. Anal. | AI | Cloud Comp. | Comp. Theory | Discrete Struct. | Further Prog. | OS Principles | Prog. Fund. | Prog. Tech. | Soft. Eng. Fund. | Soft. Eng. P&T |
| Required accumulated credits T*i* | 6 | 6 | 6 | 6 | 6 | 0 | 6 | 6 | 0 | 0 | 6 | 6 |

1. GitHub link to code: <https://github.com/Jared-Song/AAA/>

Diagram of tree:

Graphical user interface

Description automatically generated with medium confidence