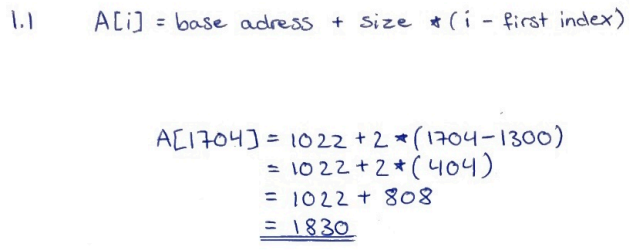
Algorithms and Data Structures Exam

PG4200

Candidate number
2038
9 August 2023

1.1 To access any random element from a 1-D array where the given base address of an array A[1300...1900] is 1022, and the size of each element is 2 bytes in memory, we can find the address of A[1704] by using the formula A[i] = base address + size * (i - first index). The first index in this case is 1300 and we end up with the result **A[1704] = 1022 + 2 * (1704-1300) = <u>1830</u>**.



1.2 In order to traverse the whole array and find the largest number among them, we need to do the following:

Step 1: Initialise a local variable for the highest value to the first element in the array[0] to store the maximum among the list. We need this to compare the elements during iteration.

Step 2: Initialise an integer i = 0 and repeat the next steps until i reaches the end of the array, starting from index 0 to the end of the array.

Step 3: Compare the current index value array[i] with the current max value.

Step 4: If the current index value is bigger than max value, set the max value to this index. If array[i] > max value = true, max value = array[I].

Step 5: Increment i once per iteration.

Step 6: Once we have traversed the whole array and the variable for highest value has been compared to each element in the array and updated, we return max value as the answer.

```java
package question1;

// 1.2
public class LargestInArray {
    public static void main(String[] args)
    {
        int[] array = {10, 7, 11, 5, 13, 8, 38, 37, 14, 92, 84, 74, 77, 20, 40, 47, 33, 65, 62, 69, 73};
        int largest = findLargest(array);

        System.out.println("Largest element in array is " + largest);
    }

    1 usage
    static int findLargest(int[] array)
    {
        int maxValue = array[0];

        for (int i : array)
            if (i > maxValue)
                maxValue = i;

        return maxValue;
    }
}
```

2.1 A deck of cards or stack of books can be used to explain the stack data structure. It is a linear data structure that follows the last in first out principle, which means that the value entered first will be removed last. It contains only one pointer top pointing to the topmost element of the stack. Whenever an element is added to the stack, it is added on top of the stack, and the element can be deleted from the stack. This data structure can be defined as a container in which insertion and deletion can be done from one end, the top.

The operation for inserting an element in a stack is called push() and is done by following these steps:

Step 1: Before inserting an element in a stack, we check whether the stack is full. The top pointer element is the last element in the stack and indicates whether the stack has reached its full capacity or not. If the position of the top pointer equals the maximum index of the stack (max capacity - 1), the stack is full and the overflow condition occurs. That means we can not proceed with the push operation until we have increased the capacity of the stack, and is handled by these under-steps:

> Step 1.1: Determine new capacity, for example twice the size as the original stack.
>
> Step 1.2: Copy the contents of the original stack to the new one. Do this by iterating through the original stack to ensure the correct position for each element.
>
> Step 1.3: Update the top pointer of the new stack to indicate the last

inserted element.

Step 1.4: Replace old stack with new stack, now the stack has twice the size as the original one.

Step 2: When we initialise a stack, we set the value of the top as -1 to check if the stack is empty.

Step 3: A new element is pushed in the stack where the top element is pointing, i.e at the top of the stack.

Step 4: The top value gets incremented, i.e top++, and the element will be placed at the new position of the top. The elements will be inserted until we reach the max size of the stack.

The operation for deleting an element is known as pop() and works by following these steps:

Step 1: Before deleting the top element from the stack, we check whether the stack is empty. This is done by comparing the top to 0 in this case, referencing the first index. If top equals this index is true, the underflow condition occurs. This means that the stack is empty and we can not perform the pop operation since there is no element to delete. This can be handled by an exception.

Step 2: Given that the stack is not empty, we access the top element indicated by the top pointer.

Step 3: Having accessed the top element, we decrease the top by 1 to point to the index of the second last element in the stack. We do this because the top can not be pointed at an element that soon will cease to exist.

Step 4: Change the position og this element to null, -1 or 0, depending on the implementation and return the element. The element is deleted and the top pointer has changed to the next available position in the stack.

2.2 When describing algorithms I prefer to write the code to reference as it personally helps me understand on a deeper level. Because of this, I had already created a stack data structure with push() and pop() operations as can be seen in the Stack class under the package *question2* in the Java project folder. To answer this question I extended my code with helper methods called pushHelper(), popHelper() and printStack() as well as the getMin() method.

I implemented a hard coded stack data structure using my own Node class found under the package *nodes*. Given N elements, I retrieved a minimum element in O(1) time with this input:

- push(5)
- push(7)
- push(2)
- getMin()
- pop()
- push(25)
- getMin()
- printStack()
- push(1)
- printStack()
- push(10)
- getMin()
- printStack()
- pop()
- pop()
- getMin()
- printStack()

I made sure to test the algorithm properly to ensure that it was done correct. I had one Main class where I initialised the Stack class and performed all helper operations. In the Stack class I declared top of type Node and *minElement* of type integer. The class has one constructor where the top is set to null before elements are inserted.

The push method takes an integer parameter x, referencing the value of the element to be inserted. A new node is initialised and given this value. It checks whether the stack is empty or not and makes sure to change the top pointer accordingly. My implementation of the stack data structure is based on a linked list representation and doesn't have constraints, meaning we can not check if the stack is full. However this would be a good practice and is necessary in for example an array list. In this case I could make a helper method that defines constraints based on memory etc.

Since the helper methods are called from the Main class, the push- and pop-helper methods takes a stack as parameter. The pushHelper also takes integer x to pass onto the push method. It first checks if the stack is empty with the condition top == null, if true it will push x and set *minElement* to x before returning. If false, meaning there are already items in the stack, it

checks whether or not x is smaller than *minElement*. If true it will push 2 * x - *minElement*, and set *minElement* to x. If false it will push x. This is to ensure that the *minElement* is stored correctly.

The pop method checks if the stack is empty or not, to ensure that there is something to delete. It declares the popped value as top.data, since stack deletes from the top down. Then the top is changed to the second top element and the popped value is returned.

The popHelper takes in a stack as described above. It declares an integer variable by calling on the pop method. Then it is necessary to keep track on the *minElement*. If the popped value is smaller than *minElement* then *minElement* is set to 2 * *minElement* - poppedValue to make sure the minimum element is stored correctly.

The getMin method checks if the stack is empty, as long as it is not empty it will print the *minElement*. The printStack method initialises a variable *current* that is set to top. As long as current is not null it will iterate and print all elements of the stack from top to bottom, setting *current* to *current*'s next for each iteration. It will also check if the min element stores the value of the top element.

This code gives the following output in O(1) time:

- Pushed 5
- Pushed 7
- Pushed 2
- Min element is 2
- Popped 2
- Pushed 25
- Min element is 5
- Stack content from top to bottom: 25 7 5
- Pushed 1
- Stack content from top to bottom: 1 25 7 5
- Pushed 10
- Min element is 1
- Stack content from top to bottom: 10 1 25 7 5
- Popped 10
- Popped 1
- Min element is 5
- Stack content from top to bottom: 25 7 5

3.1 When writing an algorithm for binary search of an array with recursive approach we need to divide, conquer and obtain the solution. This can be

done with the following steps:

Step 1: Initialise an array = {10, 12, 13, 14, 18, 20, 25, 27, 30, 35, 40, 45, 47} and suppose the target x = 18.

Step 2: With this array we know that it's size is it's length - 1, i.e 12. The size is the highest value, and the lowest is 0 (starting index). We declare these values as left (lowest) and right (highest). Take note that the array is sorted before proceeding.

Step 3: Check whether the left index is greater than the right index. If so return -1. This means the target element is not found.

Step 4: As long as the left index is less than the right index, we calculate the middle index to be left + (right - left) / 2. In the first iteration we will find the middle index on position 6 with the value 25.

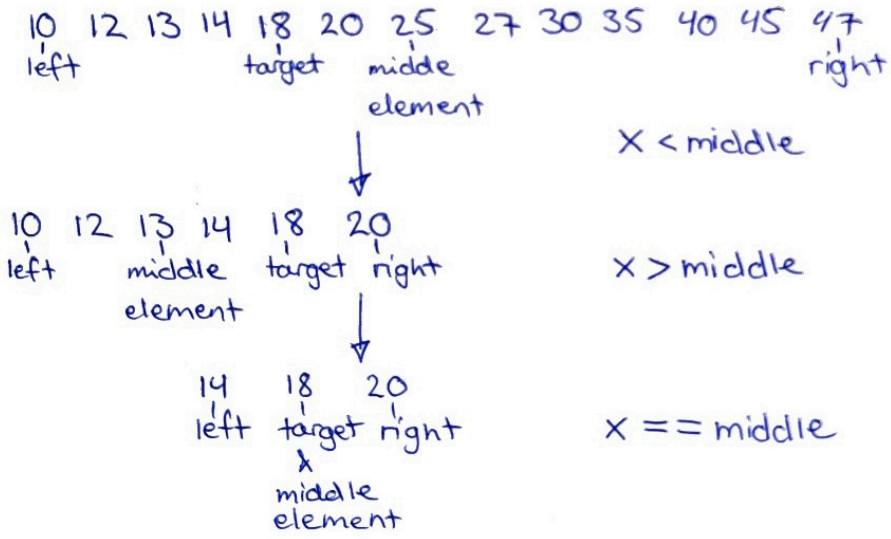
Step 5: If the middle element is equal to the target element, return the middle index and quit. Since the target is not the middle element in this case we need to search by dividing the array and find in which subarray the target is located.

> Case 5.1 If the target element is greater than the middle element, we recursively call the method with the middle index +1 and the right index. In this case the target 18 is not greater than the middle element 25, so we need to check the left subarray.
>
> Case 5.2: If the target element is less than the middle element, we recursively call the method with the left index and middle index -1. In this first iteration 18 is less than 25 so we proceed.

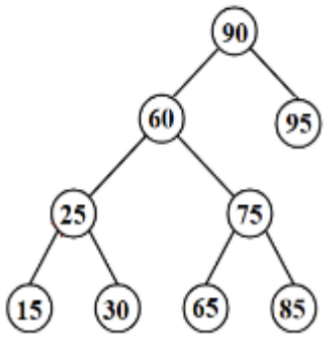
Step 6: Having selected the left subarray = {10, 12, 13, 14, 18, 20} we repeat step 3-5 until we find the target.

To further explain, the new middle index is 2 with the value 13. Left is still 0, but right has changed to index 5 with value 20. The target 18 is greater than the middle 13 so we will proceed to case 5.1. For this iteration the middle index is 4 with the value 18, which matches the target element. We have found the target element.

The algorithm for this question can be found in the BinarySearch class located in the *question3* package in the Java project folder. Note the graphical representation below.



3.2 In order to delete a node with value of 60 in given binary search tree, the following steps needs to be executed:



Dr. Gupta, R (2023). Illustration

Step 1: First check if the node to be deleted is null. If the node is null, the binary search tree is already traversed and the node to be deleted does not exist. Return null. Our target node is not null so we can proceed.

Step 2: The node needs to be replaced. An important observation is that this node has two children (25 and 75). This gives us two options, replace from the left or right subtree:

Option 2.1: Choosing the left subtree, we take the node with the largest value, also called in-order predecessor node and replace it with the target node. In this case the values 60 and 30 will switch places.

Option 2.2: Choosing the right subtree, we take the node with the smallest value, also called in-order successor node and replace it with the target node. In this case the values 60 and 65 will switch places.

Step 3: Given that the right subtree is chosen, we'll iterate from the root with value 90 to the target node recursively.

Case 3.1: If the target is less than the current node, we traverse the left subtree.

Case 3.2: If the target is greater than the current node, we traverse the right subtree.

Case 3.3: If the target is equal to the current node, we check whether or not it is a leaf node.

— If so return null.
— If it is not a leaf node, we need to check whether or not the current node has one or more children.

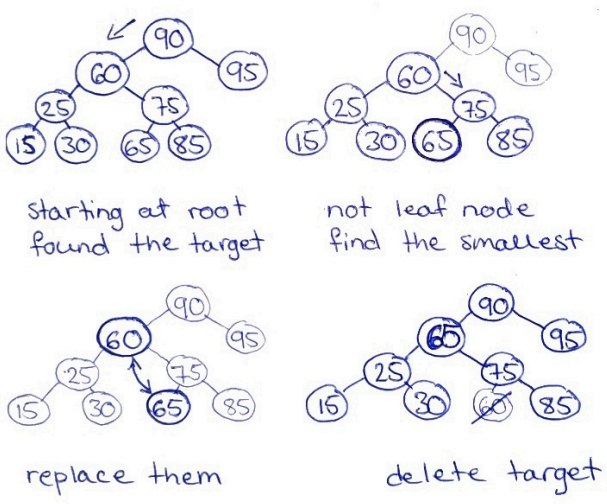
- In the case of one child, return the child node.
- In the case of more children, find the smallest value in the right subtree, replace the value of the target node with the smallest value in the right subtree and set the targeted leaf node to null.

Step 4: For the first iteration, since our target node 60 is less than the root node of 90, we will first iterate the left subtree (case 3.1).

Step 5: For the second iteration the current node equals the target node (case 3.3). From this we know the node has children. Therefore the smallest value from the right subtree is replaced with the target (smallest 65 <=> 60 target).

Step 6: The leaf node with value of 60 will then be set to null, and we have successfully deleted the target from the binary search tree.

This algorithm can be found in the BinarySearchTree class in the *question3* package in the Java project folder. See graphical representation below.



The in-order traversal sequence for this binary search tree will be 15 25 30 60 65 75 85 90 95 before deleting and 15 25 30 65 75 85 90 95 after.

4.1 Merge sort is a divide and conquer algorithm. It works by first dividing the unsorted array in half, then creating subarrays until each subarray consists of one single element. Followed by sorting the subarrays by merging pairs in order until the whole array has been sorted. Some sorting algorithms such as bubble or insertion sort, would be affected by the initial order of elements in the array. Merge sort differs on this and is unaffected by the initial order of elements. The number of divisions and merges is determined by the total number of elements.

Given we have initialised an array = {70, 50, 30, 10, 20, 40, 60}, these are the steps to use merge sort:

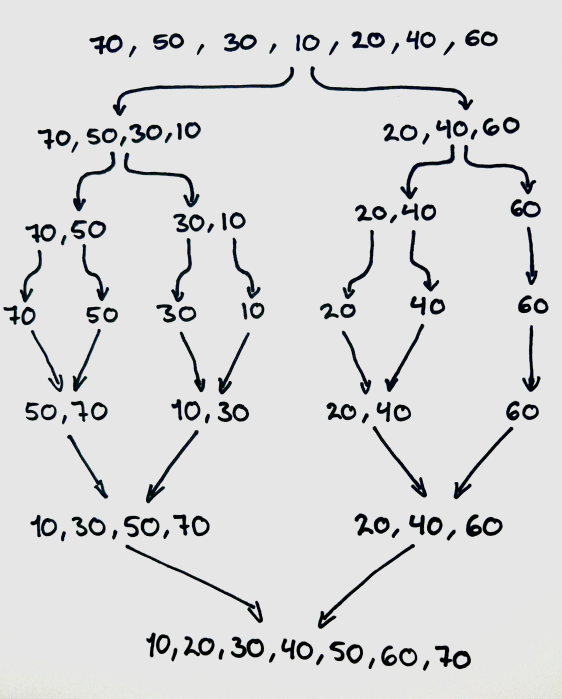
Step 1: Find the middle element of the array and split the array in two, evenly if possible. Middle index = (left + right) / 2. To start off left = starting index and right = array.length -1. In this case it means that (0 + 6) / 2 = 3 and at index 3 we find the value 10. This step is repeated recursively until the base case is fulfilled, meaning every element is alone in a subarray.

The first subarrays would be {70, 50, 10, 10} and {20, 40, 60}. Then it would be {70, 50}, {30, 10}, {20, 40}, {60}. Finally the subarrays would be {70}, {50}, {30}, {10}, {20}, {40}, {60} and the base case is fulfilled.

Step 2: Once the base case is fulfilled we need to merge the subarrays by comparing the left and right part iteratively until we have the full array sorted.

Step 3: We initialise empty subarrays and after comparing the elements we identify the smallest element to copy over. After copying all elements over in order, we set the original unsorted array to this new sorted array and we have successfully performed the merge sort algorithm.

The algorithm for merge sort can be found in the MergeSort class under package *question4* in the Java project folder. See graphical solution.



4.2 Just like merge sort, quick sort is also an algorithm that follows the divide and conquer principle. Using an array, it is divided recursively into two parts and sorted. It uses an element as pivot and partitions the array around said pivot. This item is placed on the correct index compared to the data surrounding it. Elements smaller than this item are placed to the left, and bigger elements are placed to the right. Where quick sort differs from merge sort, is that it's algorithm is heavily affected by the pivot. In some instances this algorithm can have a worst case scenario of $O(n^2)$. This can happen if e.g. the array being used already has been sorted and the smallest or greatest is selected as pivot. Because of this, randomising the data before sorting is recommended. Given we have an array = {15, 22, 13, 27, 12, 10, 20, 25} we can implement the quick sort algorithm by doing the following steps:

Step 1: Choose a pivot. It can be any element but in this case the last element is chosen, pivot = array[array.length -1], found at index 7. Note that I am randomising the data before starting. The pivot divides the array in two, creating two subarrays.

Step 2: Initialise two pointers to compare the elements. All elements smaller than the pivot will be placed into the left subarray, while items larger than or equal to the pivot will be placed into the right subarray. These pointers can be called left and right pointers. The left pointer is set to left -1 while the right pointer is set to left.

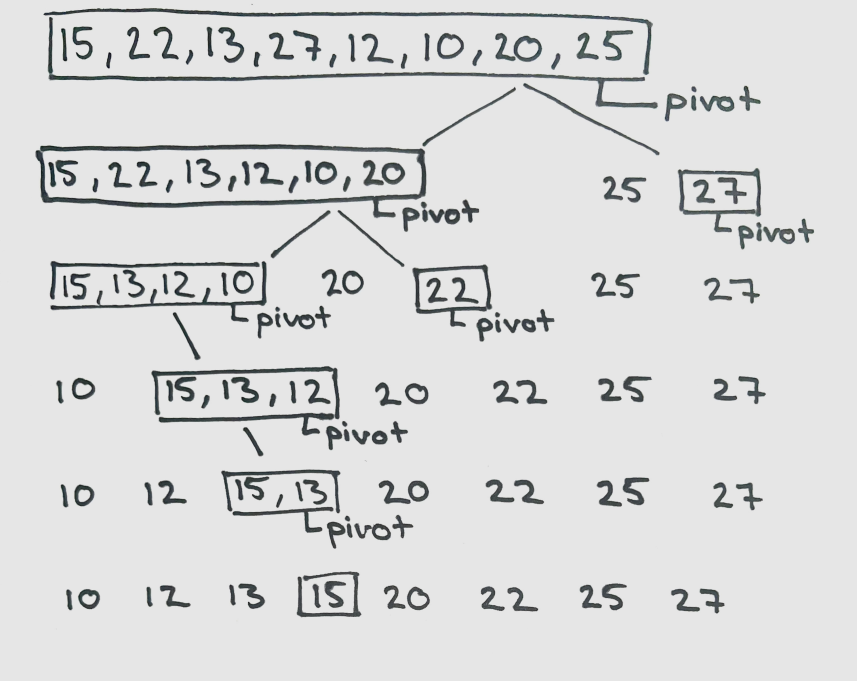
Step 3: Traverse the array and compare elements using the pointers mentioned in step 2. The right pointer moves through the array from left to right -1, comparing each element with the pivot.

Step 4: Swap items. If the element of the right pointer is greater than or equal to the pivot, the left pointer is incremented and the elements at the positions of the pointers are swapped. We do this to ensure that items less than or equal to the pivot are positioned to the left side of the left pointer, and items greater than the pivot are positioned to the right side of the left pointer.

Step 5: When the pointers reaches one another in the array, all elements has been compared with the pivot and placed accordingly. Now a new pivot must be placed for each subarray, and we recursively repeat the partitioning until the whole array is sorted.

The algorithm for quick sort can be found in the QuickSort class in the *question4* package in the Java project folder. See Graphical representation

below, where the elements in boxes are in subarrays, and out of boxes has been placed in the correct position.



5.1 When solving fib(n) for 1<= n <= 5 with both iteration and recursion, I stumbled upon an issue with the recursive method. The problem with recursion for Fibonacci term is that for every recursive call, we must recalculate all fibonacci numbers until we reach the base case. This is redundant as the fibonacci numbers has already been solved. Given that n = 5, this is not an issue. However if n increases, the number of calculations increases and so does the complexity, resulting in Big O(2^n). The higher n is the higher is the chance of encountering performance issues and stack overflow in the domain, as it is most likely confined by constraints. A solution to this problem is storing the fibonacci numbers in an array. This ensures that the algorithm does not have to perform redundant calculations. I chose this implementation for the recursive algorithm, also known by the name memoization. Recursion with memoization greatly improves the time complexity to O(n) and space complexity to O(N). When comparing this algorithm to the iterative approach, it performed 1000-2000 nanoseconds better in execution time when n = 5. However when n = 100, the algorithm using iteration performed 14000 nanoseconds better than the one using recursion with memoization. The reason for this is that storing the fibonacci numbers as local variables significantly reduces the use of memory space and gives a space complexity of O(1) and time complexity of O(n). In this case, only the two last values are stored. The difference increases exponentially as n increases. Using memoization is a good approach to

improve the worst case scenario for recursion, however I would argue that an iterative approach is more efficient for solving fibonacci term, because of the execution time and reduced memory usage.

The code for this solution can be found in the Fibonacci class under the *question5* package in the Java project folder.

5.2 When traversing a directed graph with Depth First Search (DFS), using recursion or an explicit stack are familiar implementations. The key difference between them is how thy manage the vertices. I chose to implement DFS with recursion in my code, where the call stack is used to monitor the vertices. For every call to the DFSUtil method, a new layer is added to the stack. When using recursive DFS there is a risk of stack overflow, so this is important to keep in mind.

The DFS algorithm categorises each node of the graph as either visited or not visited. The goal is to do this without rotation. Follow the steps below to implement this algorithm.

Step 1: Initialise an array to keep track of which nodes that has been visited. In this implementation we are using an array of type boolean with the fixed side of number of vertices.

Step 2: Take the top vertex and add it to the array we declared in step 1. Since the stack data structure follows the first in last out principle, we can only take out the top element. Given this implementation uses an already declared graph, the first top element is 5.

Step 3: Declare a list of the current node's adjacent nodes, and add those that are yet to be visited to the top of the stack. In this first round the stack will be 7, 3 from bottom to top.

Step 4: Traverse the graph. The current vertex is marked as visited and the neighbours are iterated. If a neighbour has not been visited yet, the method calls itself recursively with the neighbour as the new current node.

Repeat step 2-4 until all elements has been visited. To continue the next current node would be 3. It is marked as visited and removed from the stack. It's neighbours are added to the stack. The array keeping track of visited vertices will now contain 5, 3 and the stack will consist of 7, 4, 2. As shown on the illustration below, the DFS algorithm explores as far as it can reach on every branch before backtracking. **The final output will be 5, 3, 2, 4, 8, 7.**

Breadth First Search (BFS) differs from DFS. Where DFS reaches deep into a graph, exploring each branch as far as possible before backtracking, BFS is different because this algorithm will investigate neighbours before proceeding to the next level. DFS it is well suited when the search space is large and solutions are dense but located deep in the graph, while BFS is well suited for searching at a shallow level.

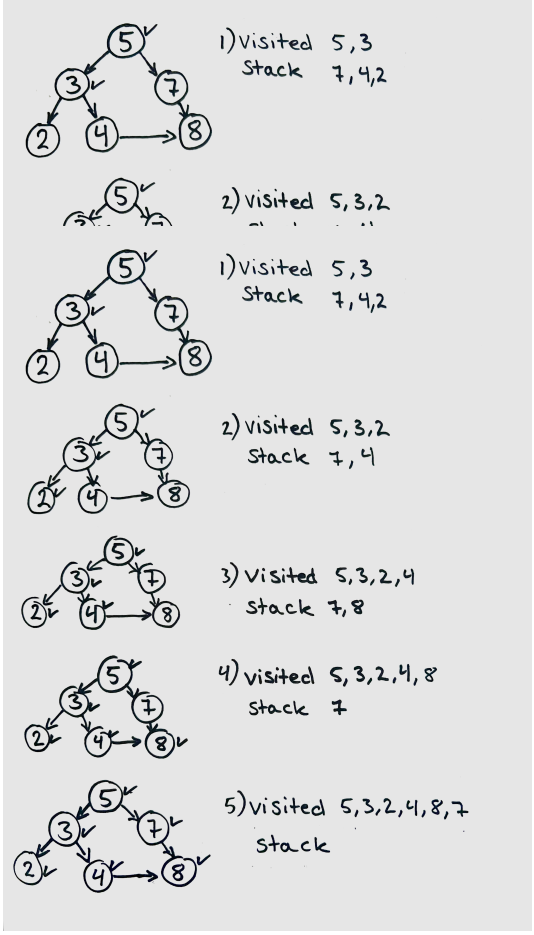


Illustration for DFS.

Much like the DFS implementation, the BFS algorithm will categorise each node of the graph as either visited or not visited. The goal is to do this without rotation. Follow the steps below to implement this algorithm.

Step 1: Initialise an array to keep track of which nodes that has been visited. In this implementation we are using an array of type boolean with the fixed side of number of vertices.

Step 2: Declare a queue to maintain the order of vertices to be visited and enqueue the starting node. Any element can be the starting node, but for this case we will use 5.

Step 3: Since queue follows the first in first out principle, the node at the head will be dequeued and added to the array of visited items.

Step 4: All unvisited neighbours of the current element is then added to the back of the queue and marked as visited.

Step 5: Repeat step 3 and 4 until every vertex that can be reached from the starting node has been visited, i.e. the graph has successfully been traversed. Then dequeue the last element.

Following the graph the adjacent nodes of 5 are 3 and 7. They are marked as visited and 5 is removed from the queue, making 3 the head of the queue and 7 the tail. After iterating their adjacent vertices they are also removed from the queue. We end up with **the final output of 5, 3, 7, 2, 4, 8.** In terms of time complexity, BFS has an upper bound of O(V+E), where V (vertices) is the amount of nodes, and E (edges) is the amount of neighbours.

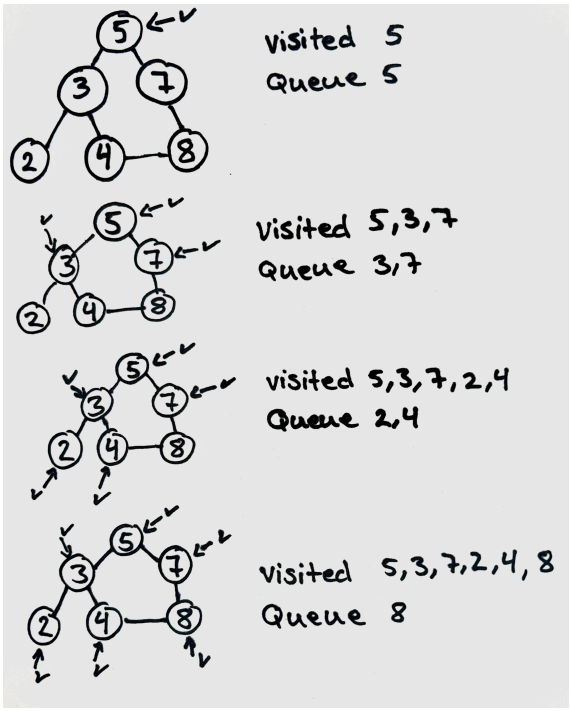


Illustration for DFS.

One thing worth noting is that DFS and BFS has the same goal, mark all nodes as visited without rotation. Going through the BFS algorithm for this graph, the goal has been achieved.

The code for both DFS and BFS is located in the TraversingGraphs class under the *question5* package in the Java project folder. Please note the illustrations for both algorithms to the right.

6.1 Some problems are yet to be solved, these problems are divided into classes called complexity classes. Such a class is a collection of problems with similar complexity. They are defined to allow scientist to label problems based on the resources required to be solved and verified. The common resources are time and space, meaning how long the algorithm takes to solve problems and verify the solutions. The time complexity of an algorithm is used to describe the number of steps required to solve a problem, and how long it takes to verify the answer. In data structures, the operations might have contrasting best and worst case time complexity. The space complexity of an algorithm describes how much memory is required to store temporary data or final result while the algorithm is operating. Having a solid comprehension of Big-O notation is vital to understand complexity classes.

These big problems are usually defined as a specific complexity class by running the problem on an abstract computational model, like a Turing machine. Since a problem can be solved in many ways, it is essential to use time and space complexity to compare suggested solutions when determining the most efficient one.

6.2 Data structures allow us to organise and store data, where as algorithms allow us to process that data efficiently and meaningfully. Big-O notation is a mathematical notation that represents the upper limit of an algorithm's cost and is associated with a big O parentheses in a formula that uses n as the size of the input. Big-O is a member of a group of notations invented by P. Bachmann and E. Landau, also known as Bachmann-Landau notation. By using Big-O notation we are categorising how an algorithm behaves as the size of the input increases. It is meaningful to to estimate how an algorithm scales, another term for this is asymptotic analysis.

The asymptote of a curve is a line where the distance between the curve and the line approach zero as they tend towards infinity, i.e. asymptotic analysis is the measurement of how the input of an algorithm affect the behaviour as the input approaches a limit. We make note of the values as they approach limits, which is to say as the values get bigger it gets more interesting.

Algorithms does not scale in the same way, so when talking about complexity analysis, it is really asymptotic analysis, the performance of an algorithm as the input approaches the upper limit. In order to know the upper limit it is essential to be familiar with the domain, as it is probably constrained. The most sensible approach is to ignore values that does not change the overall shape of the curve. One example of this is that O(n+1) would still be expressed as n because only the variable component is influencing the change in behaviour. With O(n) algorithms, the cost of the algorithm scales linearly with the size of the input. A single input has a cost of 1. One million inputs would have a cost of one million.

Logarithmic algorithms has less than a linear runtime complexity. Which means as the input grows, the cost of the algorithm does not increase at the same rate. O(log n) algorithms work by dividing a large problem into smaller chunks. By for example looking for a string “Ola” in an array of names in Norway, the algorithm will attempt to divide the problem set in half repeatedly until the target is found. The less than linear growth means that as the size of the input grows, the cost of the algorithm does not increase at the same rate. The overall cost of performing the operation on 1 million items is only twice the cost of performing the operation on 1000 items.

$O(n^2)$ algorithms are the ones where resource usage is the square of the input. One example of an $O(n^2)$ or quadric algorithm would be one that uses a nested loop to iterate over the same array. In the example below the array is iterated and call the process function n2* where n is the count. If the count is 1000, the outer for loop is going to execute 1000 and every time it executes it’s going to execute the inner for loop 1000 times. That means process will be called 1 million times. An algorithm of quadratic growth gets really expensive really quickly. While an input size of 1 has a cost of 1, an input size of 1 million will have a cost of 1 trillion.

```
void quadratic(char[] input, int count) {
  for (int i = 0; i < count; i++)
      for (int x = 0; x < count; x++)
          process(input, i, x);
}
```

Some methods have more than one input, which contributes to growth. A nested loop that iterates over two distinct collections of data might indicate an O(nm) algorithm. The issue with this algorithm is that is n is very big and m is very small, it's the basically an O(n) algorithm. Because of this uncertainty it's helpful to look into the problem space to understand if the current domain is biased towards one extreme or another to decide if an algorithm is appropriate.

If a programmer needs to search through one million records, and each time they compare a record it takes 1ms. The constant time algorithm requires only 1ms, and the logarithmic algorithm requires just 6ms. The linear algorithm will spend just over 16 1/2 minutes, and the O(nm) algorithm takes 16 1/2 minutes multiplied by whatever m is. The $O(n^2)$ algorithm will take over 11 1/2 days to finish. Even though that seems like a long time, imagine an $O(n^3)$ algorithm that would take 31 million years to complete. This is why we use Big-O notation to find the worst-case behaviour when analysing algorithms' complexities in data structures. The common sorting algorithm quick sort has a worst case complexity of $O(n^2)$, but an average complexity of O(n log n), which makes it a good general purpose sorting algorithm.

6.3 In complexity classes, the P class stands for polynomial time. It is made out of decision problems with a yes or no answer that can be solved by a deterministic Turing machine in polynomial time. A lot of problems can be solved with brute forcing, but the issue with this solution is that it usually requires a huge amount of time. If the problem has a polynomial time algorithm, it would be solved more efficiently than using brute force. In P class such a problem can be solved in n^k time for constant k, where n is the size of the input. In other words, a the solution could be found in a reasonable amount of time as the input grows, i.e. worst case scenario O(n^k). A feature of this class is that the solution is easy to find. P is regularly a class of computational problems, which are solvable and tractable. This means the problems can be solved in theory as well as in practice. If there's a solution in theory but not in practice, it is intractable. A known issue with P class is determining the biggest common divisor, finding a maximum match, and decision versions of linear programming. Some problems that belong to this complexity class can be found in sorting and searching algorithms. Sorting a collection of data in a specific order can be solved by a deterministic algorithm in polynomial time.

6.4 The Non-deterministic Polynomial Time class, or NP class is a collection of decision problems that are solvable by a non-deterministic machine in polynomial time. This includes problems that are solvable in polynomial time up to problems which are solved in exponential time.

Although they might need a larger amount of time to be solved, the solutions in NP can be verified by a Turing machine. In other words, a given yes answer to a NP problem are verifiable in polynomial time. This answer is also referred to as a certificate. Another feature of the NP class is although the problems are hard to solve, the solutions are easy to verify.

A real life example of a NP class problem is the map colouring problem. Say you have a map of every country in the world and you need to figure out how many colours are needed to make sure each country coloured in are not the same as a neighbour. Trying to figure out how many colours you need is impractical. However verifying that a fully coloured map is reasonable. You count the total amount of colours in the solution. This problem scales linearly with the number of countries on the map, meaning it is a nondeterministic polynomial time problem.

6.5 NP-complete (or CoNP class), stands for complement of NP class. A problem of this class is an extension of the last class, but also NP-hard. If the solution to a problem in CoNP is no, then there's proof it can be checked in polynomial time. In other words, if a problem X is in NP, then its complement X is also in CoNP. There is no need to verify all answers at once in polynomial time, but it is necessary to check one answer, either yes or no in polynomial time for a problem to be in NP or CoNP. Some real life examples of this type of problem are checking for prime numbers and integer factorisation.

Sources

- GeeksforGeeks. (2023). Types of Complexity Classes P NP CoNP NP hard and NP complete. GeeksforGeeks. https://www.geeksforgeeks.org/types-of-complexity-classes-p-np-conp-np-hard-and-np-complete/

- Complexity Classes | Brilliant Math & Science Wiki. (n.d.). https://brilliant.org/wiki/complexity-classes/

- Southard, A. (2018, June 11). Dynamic Programming — Recursion, Memoization and Bottom Up Algorithms. Medium. https://medium.com/@andrewsouthard1/dynamic-programming-recursion-memoization-and-bottom-up-algorithms-61c882d1c7e

- Prof. Dr. Gupta, R. (2023) Computability and Complexity [Presentation slide]. Høyskolen Kristiania.

- Horvick, R. (2020) Algorithms and Data Structures [Online course]. Pluralsight. https://app.pluralsight.com/library/courses/algorithms-data-

structures-part-one/table-of-contents

- Own work and compendium (2023). Algorithms and Data Structures PG4200 [Unpublished]. Høyskolen Kristiania.