**1) Describe a recursive algorithm for finding the maximum element in an array, A, of n elements. What is your running time and space usage?**

Time Complexity O(n)

Space Complexity O(n)

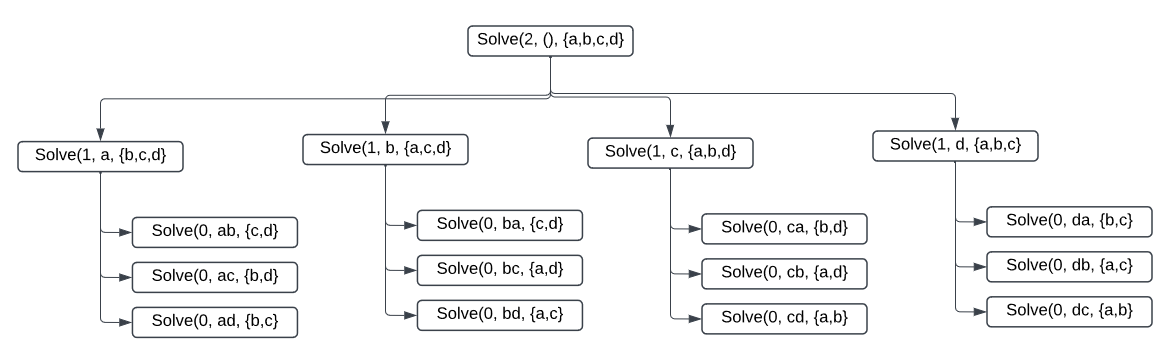
* Function int max(int array, int position)
  + If position < 0 (Invalid input)
    - Throw an error
  + Else if position < 1
    - Return array[0] (the first position in the array)
  + Int returned = max(array, position -1)
  + If(returned > intArr[position]
    - Return returned
  + Return intArr[position]

This recursive call would get to the base case of position less than 1 and return the first element in the array. As it goes back up the call stack, it is compared against another value because returning the max between the two. As we reach the top, the max of all the comparisons is return.

**5) Draw the recursion trace for the execution of reverseArray(data, 0, 4), from Code Fragment 5.3.2, on array data = {4, 3, 6, 2, 6}.**

* Initial Call
* Reverse Array then call function again
  + |
  + +----> [6,3,6,2,4]
    - First recursive call
    - Reverse Array then call function again
    - |
    - +----> [6,2,6,3,4]
      * Third recursive call
      * Reverse Array then call function again
      * |
      * +----> [6,2,6,3,4]
        + Final recursive call
        + |
        + +----> Low now > high. Return data as is

**6) Draw the recursion trace for the execution of method Solve (2,S,U), from Code Fragment**[**5.3.6**](https://learn.zybooks.com/zybook/HARPERCOLLEGECSC216JAVAJamesFall2024/chapter/5/section/3?content_resource_id=97762991)**, where S is empty and U = {a, b, c, d}.**



**7) Describe a recursive algorithm for computing the nth Harmonic number defined as Hn = n, k=1, 1/k**

* Function name summation (int n)
  + if n == 1
    - return 1
  + return 1.0 / n + summation(n -1)

This recursive algorithm will first do 1/n, then recursively call itself with n – 1.

As we reach the bottom of the call stack where n = 1, it will return 1. As we move up the call stack, it will return doubles and add itself into the above call before that call returns. So and so forth

**8) Describe a recursive algorithm for converting a string of digits into the integer it represents. For example, '13531' represents the integer 13,531.**

* Function name toInt (string)
  + If string length is 0 // base case
    - Return 0
  + Character lastChar saves the last character of the array
  + Return (int) lastChar + (10 \* toInt(substring of string without last character)

When we get to the base case where string length is 0, we turn 0. As we move up the call stack, we should have the largest digit in the ending integer because we kept the lastChar in each recursion. So as we return, we multiply the return value by 10 so that it is moved to the correct place in the integer and add the values moving back up to the original call

**10) Describe a way to use recursion to compute the sum of all the elements in an n x n (two-dimensional) array of integers.**

* Function arraySum (2Darray, int lengthOfArray)
  + If n < 0 // base case
    - Return 0;
  + Initialize int rowSum = 0
  + For (int index = 0; index < array[lengthOfArray].length; index++)
    - rowSum = rowSum + array[lengthOfArray][index]
  + return rowSum + arraySum(array, n-1) // Recursive call with reduced n

Base case of less than 0 means there are no more rows in the 2D array to add and returns 0. We use a for loop to traverse the row we’re add and sum all values into a variable rowSum. We then return rowSum against the values that are passed up by the recursive call of the same arraySum function with a reduced second input parameter. N will be reduced until we reach -1 when the base case will trigger.

**11) Describe a recursive algorithm to compute the integer part of the base-two logarithm of n using only addition and integer division.**

* Function log2 (int n)
  + If n < 2
    - Return 0
  + Return 1 + log2 (n /2)

Base case is when n is less than 2 where we will simply return 0. If n is not less than 0, it will return 1 + a recursive call of log(n/2). The first 1 here is because the number is at least of value 2 and thus the integer value is at least 2 to the power of n=1. As we divide by two, we eventually get to the base case of the log2 function being passed something less than 2. We then go up the call stack adding up the amount of times we were able to divide original int n by 2 before being less than 2 and return that value.

**12) Describe an efficient recursive algorithm for solving the element uniqueness problem, which runs in time that is at most O(n^2) in the worst case without using sorting.**

* Function unique (array, int pos1, int pos2) // pos1 and pos2 start be 0 for initial run
  + If pos1 == (array – 1) AND pos1 == pos2 // base case
    - Return true // we have reached the end of the array without finding dupe
  + If pos1 == pos2
    - Recursively call unique (array, pos1, pos2 + 1) // Same index, nothing to check
  + Else If pos2 == length of array
    - Recursively call unique (array, pos1 + 1, 0) // pos2 moved back to 0 and pos1 incremented
  + Else if array[pos1] == array[pos2]
    - Return false // a duplicate has been found
  + Else
    - Return unique(array, pos1, pos2 + 1)

In this recursive function, we intialize with an array and pos1 and pos2 which should both be initialized at 0. Our base case is where pos1 and pos2 have both reached the end of the array meaning no duplicates have been found. At this point we return true. From the top of the call stack to each recursive call, we move through every possible match to check for equivalency only ignoring when pos1 and pos2 are referencing the same position.

This will unfortunately run O(n^2) every single time

**13) Give a recursive algorithm to compute the product of two positive integers, m and n, using only addition and subtraction.**

* Function fakeMultiply (int m,int n)
  + If m equals 0 OR n equals 0 // base case
    - Return 0
  + If (n less than 0 AND m less than 0) OR (n less than 0 AND m greater than 0)
    - n = n – n – n // if both negative, makes both positive
    - m = m – m – m // if only n negative, flips n to positive and makes m negative
  + return m + recursive cal fakeMultiply(m, n -1)

When we reach base case where m or n equals to 0, we move up the call stack where we begin to add m to itself n amount of times.

**16) In the Towers of Hanoi puzzle, we are given a platform with three pegs, a, b and c, sticking out of it. On peg a is a stack of n disks, each larger than the next, so that the smallest is on the top and the largest is on the bottom. The puzzle is to move all the disks from peg a to peg c, moving one disk at a time, so that we never place a larger disk on top of a smaller one. See figure below for an example of the case n=4. Describe a recursive algorithm for solving the Towers of Hanoi puzzle for arbitrary n.**

* Function towerOfHanoi (diskNumber, fromPeg, otherPeg, toPeg)
  + If diskNumer is equal to 1
    - Move disk 1 from fromPeg to toPeg
    - Return;
  + Recursive call towerOfHanoi(diskNumber – 1, fromPeg, toPeg, otherPeg) *// Please note the different order. This means that in the next call, otherPeg will be the toPeg*
  + Move disk from fromPeg to toPeg
  + Recursive call towerOfHanoi(diskNumber -1, otherPeg, fromPeg, toPeg) *// Again, please note the diferent order. This means that* *in the recursive call the otherPeg will be the toPeg, fromPeg will be the otherPeg*

I need help on this one. Please note below sources

Source One:

Program for Tower of Hanoi Algorithm. (2024, May 9). Geeksforgeeks. Retrieved November 22, 2024, from https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/

Source Two. Used this website to play around with ideas:

https://www.mathsisfun.com/games/towerofhanoi.html

First Call 1:

Before it does anything else, it recursively calls itself once

Recursive Call 2:

Again, it recursively calls itself before doing anything

Recursive Call 3:

n is now equal to 1. It prints out the statement saying that it is moving disk 1 from A to C. It then returns back to Iteration 2

Recursive Call 2 Continued:

We are back on iteration 2 with disk 2. It prints out saying that we are Moving disk 2 from A to B becuase of the order we placed our A, B, C pegs into the recursive call

Recursive Call 4:

We enter the second recursive call in recursive call 2 to enter recursive call 4. We have the pegs once again ordered differently in the call. This moves disk 1 from C to B and return

First Call 1:

We come back up to the first call. Disk 3 gets moved from A to C and then enters the recursive call at the bottom of the algorithm

From here forward it is largely a repeat of the logic above except with

Different movements due to the changes of the peg order inputs

Recursive Call 5:

We enter and are immediately moved to recursive call 6.

Recursive Call 6:

Disk 1 is move from B to A and then returns

Recursive Call 5 Continued:

Disk 2 is move from B to C. Then we enter another recursive call

Recursive Call 7:

Disk 1 is moved from A to C. We return all the way to the top of the method and terminate the method

**18) Write a short recursive Java method that takes a character string a and outputs its reverse. For example, the reverse of pots&pans would be snap&stop.**

* Function reverse (String inpString)
  + If inpString length is 0 // base case
    - Return inpString
  + Return recursive call reverse(inpString from char index 1 to end) + character at index 0

Each time we call the string, we reduce the input string from the front and recursively call it without the first character and add that character to the end of the string that will be returned from the recursive call. When we get to the base case, it returns nothing. And as we move from base case upward, the previous characters get concatenated to the end of what is returned by the recursive call.

**22) Given an unsorted array, A, of integers and an integer k, describe a recursive algorithm for rearranging the elements in A so that all elements less than or equal to k come before any elements larger than k. What is the running time of your algorithm on an array of n values?**

* Function bubbleSort(int array, int lengthOfArray)
  + If lengthOfArray == 0 OR lengthOfArray == 1 // base case
    - Return array
  + For (int element = 0; element < lengthOfArray – 1; element++)
    - If array[element] is greater than array[element + 1]
      * Temp = array[element]
      * Array[element] = array[element + 1]
      * Array[element + 1] = temp
  + Return recursive call of bubbleSort(array, lengthOfArray – 1)

This is a recursive implementation of a bubble sort. We start at the further right position in the initial call of the lengthOfArray and then recursively call itself with lengthOfArray – 1 since the last element of the array should be sorted at the point of the next recursive call. By the time we reach the base case, every element should have gone through the sorting algorithm and thus we can just return the array.

Run time is O(n^2).

Other recursive option that is faster is quick sort. However, I am not confident in explaining that in pseudocode so I we will get back to that in later assignments.

**24) Describe a recursive algorithm that will check if an array A of integers contains an integer A[i] that is the sum of two integers that appear earlier in A, that is, such that A[i] = A[j] + A[k] for j,k < i.**

* Function twoSum(int[] array, int target, int lead, int lag) //initial call will be target=2 lead=0 lag=1
  + // Array is too small or we have reached the end of checkable values
  + // and there are no figures that meet criteria
  + If array length is 3 OR array length is equal to target // base case
    - Return false
  + // Check if two previous values add up to array[target]
  + If array[target] = array[lead] + array[lag]
    - Return true
  + // Lead and lag are the first two values in the array. Increase target
  + // by one and move lead and lag to just below target
  + Else if (lead == 0 AND lag == 1)
    - Increment target by 1
    - Lead = target – 2
    - Lag = target – 1
    - Return twoSum(array, target, lead, lag)
  + // Lead is 0 but lag is not yet at 1. Decrement lag by 1. Put
  + // lead one below lag
  + Else if(lead == 0)
    - Lag = lag – 1
    - Lead = lag -1
    - Return twoSum(array, target, lead, lag)
  + // Lead is not yet at 0 and nothing is equal as of yet. Decrement lead
  + Else
    - Return twoSum(array, target, lead – 1, lag)

1. Initialize method with array, target at 3rd element because you need at least two elements behind it to add up to the third. Lead and lag are initialized at 0 and 1

2. Check if array.length is at least 3. As noted, you need three elements for this problem. Also check if target == array.length and thus out of bounds of the array. This means whole array has been checked and check is false.

3. Check for equality and if criteria is true and needs to return true

4. If lead and lag take up the first two elements of the array, need to push target one forward in array to check further elements. Recursively call the function after making adjustments to target and setting lead and lag back to the two elements before target

5. Check if lead is 0. Decrement lag by one and then set lead to one below that and call function recursively with new parameters

6. No other ifs triggered, decrement lead and keep checking / calling

**26) Describe a fast recursive algorithm for reversing a singly linked list L, so that the ordering of the nodes becomes opposite of what it was before.**

* Function reverse (linkedList list, Node head)
  + Head.next == null
    - Set head to be head
    - Return
  + Create tempNode equal to head.next
  + Set head.next = null
  + Reverse(list, tempNode)
  + Set tempNode.next equal to head
  + Set the Tail to the head

1. Our base case is if head.next is null. Here, we will call the list.setHead(head) function so that the last item becomes the new head.

2. We hold the head.next in a tempNode.

3. We call recursively down to the end of the linkedlist until the new head is set and we return.

4. The tempNode where we held the reference to the next node has its .next pointer set to current node.

5. We then set the current node to the tail. This will eventually go back up to the top call and the first node is set to the tail finishing up reversing the linked list

**27) Give a recursive definition of a singly linked list class that does not use any Node class.**

* public class RecursiveLinkedList
  + private RecursiveLinkedList next // It holds a pointer of the next object in the list
  + public int data // The data is held in the linked list itself instead of a node
  + Constructuor RecursivedLinkedList (int data)
    - Stores data in itself
    - Sets pointer to next is null
  + Append function (int data) // Appends. If this.next is null, appends here. If it isn't, recursively calls this.next.append until it finds node with this.next null
    - Creates new RecursiveLinkedList newEntity
    - If pointer to next is null
      * Set pointer to newEntity
    - Else
      * Call next.append(data) // Recursive call to continue down the linked list
  + getSize() // Recursively calls itself down the linked list adding 1 to the return value until this.next is equal to null
    - if pointer to next is null
      * return 1
    - else
      * return 1 + next.getSize()
  + search() // Searches for the data provided. If it finds it, it returns back true all the way back up the chain. If it gets to this.next == null without finding the value, it returns false. If this.next still exists it will continue to search down the list for the value
    - If current link data is equal to what we’re looking for
      * Return true
    - Else if pointer to next is null
      * Return false
    - Else
      * Call next.search(data)