**Assignment 4**

1. (Weight: 10%)Trace the execution of the call mystery(4) for the following recursive function. What does this function do?

int **mystery**(int n) {

  if (n == 0)

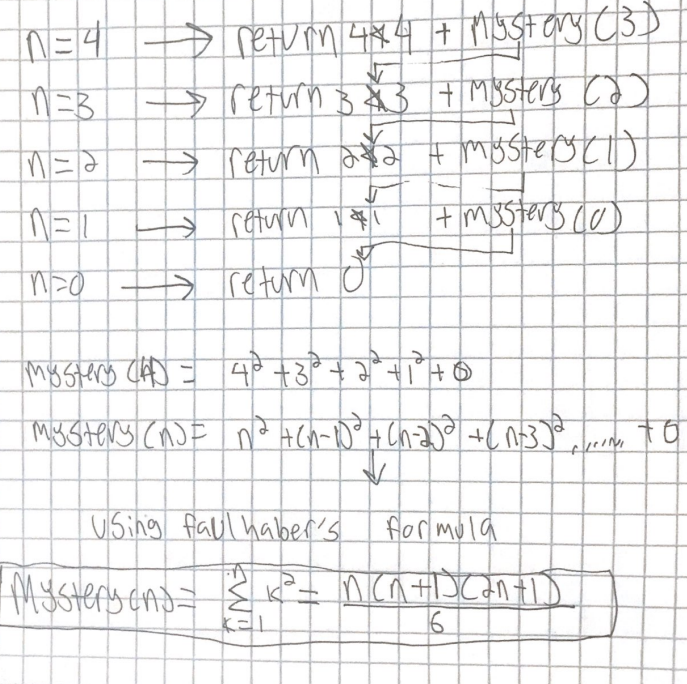
    return 0;

  else

    return n \* n + mystery(n – 1);

}

The function is a recursive call that give us the sum of the square of n positive numbers.



1. (Weight: 20%) **Programming:** Write a recursive function **to\_number** that forms the integer sum of all digit characters in a string. For example, the result of to\_number("3ac4") would be 7.

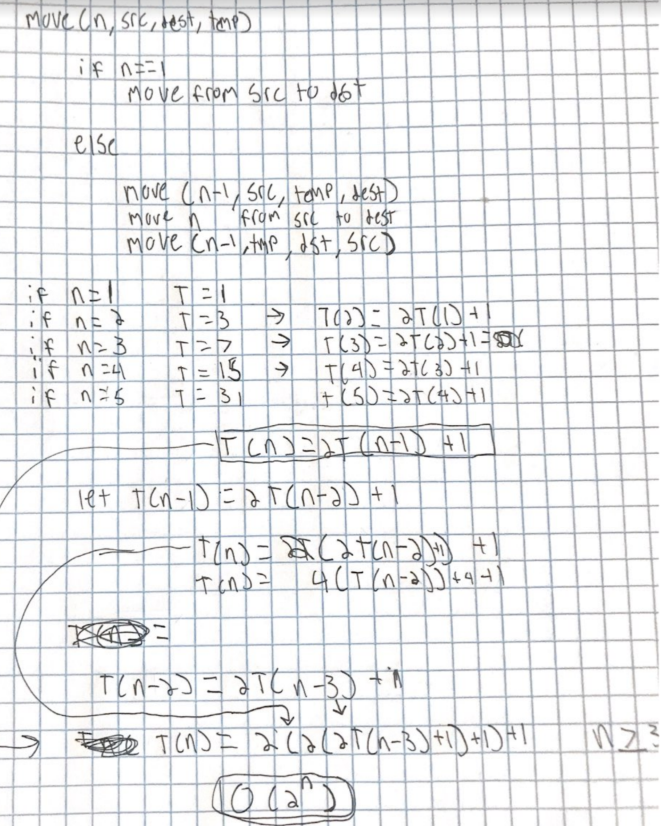
**Hint:** If next is a digit character ('0' through '9'), functionis\_digit(next) in header <cctype> will return true.

|  |
| --- |
| int to\_number(string word) {    int length = word.length();  int ret = 0;  if (word.length() == 0) {  return 0;  }  char next\_char = word[0];  if (isdigit(next\_char)) {  ret += (int)next\_char-48; //Since 48 is the ASCII value of "0"    }  return ret + to\_number(word.substr(1));  } |

1. (Weight: 10%) **Programming:** Write a recursive linear search function with a recursive step that finds the last occurrence of a target in a vector, not the first. You will need to modify the linear search function (You can find it in the slides) so that the last element of the vector is always tested, not the first.

|  |
| --- |
| Int reverse\_search( vector<type> &items, type& target, size\_t pos\_last){  If(pos\_last==0) //if empty list or if the list does not contain target  Return -1;  If (target == items[pos\_last]) //if item is found  Return pos\_last;  Else  Return reverse\_search(items, target, pos\_last-1);//if item is not found  } |

1. (Weight: 5%) What is the big-O for the Towers of Hanoi as a function of n, where n represents the number of disks? Compare it to the function 2n.



Since the time complexity is exponential, every time that n doubles, the number of operations is squared.

1. (Weight: 40%) **Programming:** Programmatically generate three type of arrays (or vectors) of size 10,000: an array that is sorted in ascending order, a reversed array (an array that is sorted in descending order), and a random array.

* Programmatically sort the three arrays using bubble sort, selection sort, insertion sort, shell sort, merge sort, and quick sort (the regular one and the improved one ) algorithms. For each sorting algorithm, calculate the number of exchanges (or shifts), number of comparisons, and the usage of extra space for these sorting algorithms. Summarize the result in tables (see below).
* Discuss whether your results match the big O of these algorithms.
* **Note:** you can use the sorting algorithms on Blackboard. For quick sort, the improved one is called "middle\_quick\_sort". **Your program will crash when you try to sort a sorted array with the regular quick sort. That’s okay.**
* **Submission:** Please submit the source code (.cpp files) as well as tables that summarize the results: This is what the table should look like:

**Comparing the number of comparisons**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Random Array** | **Sorted Array** | **Reversed Array** |
| Selection Sort | 49995000 | 49995000 | 49995000 |
| Bubble Sort (improved) | **49992722** | **9999** | **49995000** |
| Insertion Sort | 25351212 | **9999** | **49995000** |
| Shell Sort | 223610 | 110018 | 160696 |
| Merge Sort | 120425 | **69008** | **64608** |
| Quick Sort | 171321 | n/a | Stack overflow |
| Quick Sort (Improved) | 148761 | 131343 | 131343 |

**Comparing the number of exchanges:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Random Array** | **Sorted Array** | **Reversed Array** |
| Selection Sort | 9993 | 0 | 5000 |
| Bubble Sort (improved) | **25341218** | **0** | **49995000** |
| Insertion Sort (shifts) | 25341218 | 0 | **49995000** |
| Shell Sort (shifts) | 119393 | 0 | 60630 |
| Merge Sort(copies) | 133616 | 133616 | 133616 |
| Quick Sort | 31421 | n/a | Stack overflow |
| Quick Sort (Improved) | 44074 | 11808 | **16812** |

**Comparing the usage of extra space:**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Random Array** | **Sorted Array** | **Reversed Array** |
| Selection Sort | 0 | 0 | 0 |
| Bubble Sort (improved) | 0 | 0 | 0 |
| Insertion Sort | 0 | 0 | 0 |
| Shell Sort | 0 | 0 | 0 |
| Merge Sort | 133616 | 133616 | 133616 |
| Quick Sort | 0 | n/a | Stack overflow |
| Quick Sort (Improved) | 0 | 0 | 0 |

**6.** (Weight: 15%) **Programming:** Modify the source code for the insertion sort so that it can sort a list of integers (You can find the source code on Blackboard and the slides). Is there a difference in performance (compared to the vector-based one)?

|  |
| --- |
| void insertion\_sort\_list(int arr[], int size ) {      int temp,i;    for (int j = 1; j < size; j++) {      temp = arr[j];  i = j - 1;  comparisons++;  while (i >= 0 && arr[i] > temp) {  exchanges++;  arr[i + 1] = arr[i];  i = i - 1;  }  arr[i + 1] = temp;  }      } |

The performance of the insertion\_sort algorithm is the same whether you are using vectors or arrays.