Computer Science 1 Exam 1

String Functions

#include <string.h> for all functions

- **int strlen(string):** Calculates the length of the string up to but not including the terminating null character '\0'

  Example:

  (Input) Strings

  (Output) Size: 6

- **char* strcpy(destination, source)**: Overwrites the destination string with the source string

  Example:

  (Input) String1, String2

  (Output) String2

- **char* strcat(destination, source)**: Appends the source string to the end of the destination string

  Example:

  (Input) String1, String2

  (Output) String1String2

- **int strcmp(string1, string2)**: Compares the first difference in characters at the same index in string1 and string2 and returns (ascii value of char1 - ascii value of char2)

  Example:

  (Input) Coding, Computers

  (Output) -9

Dynamic Memory

Dynamic Memory Functions

#include <stdlib.h> for all functions

- **void* malloc(size in bytes):** Allocates a dynamic memory block of the given size in bytes and returns a pointer to it
- **void* calloc(number of items, size of item)**: Allocates dynamic memory for a given number of elements and initializes all of the data with either a null or 0 value
- **void* realloc(pointer to old memory block, new size)**: Attempts to resize the memory block that was previously allocated with a call to malloc or calloc and returns a pointer to it
- **void free(pointer)**: Deallocates the memory previously allocated by a call to malloc, calloc, or realloc

Memory Violations

- **Dereferencing NULL**: Trying to access a null pointer, including an array index when the array is set to NULL (Segmentation fault)
- **Memory Leak**: Losing the address of allocated memory, including not freeing dynamic memory (Computer will behave poorly when trying to clean memory)
- **Use After Free**: Trying to dereference memory after it is freed (Segmentation fault or memory protection faults)
- **Double Free**: Attempting to free already freed memory (Unpredictable behavior)
- **Dangling Pointer**: When a pointer is pointing to unallocated memory or memory is out of scope (Segmentation fault or memory protection faults)
- **Free Static Memory**: Freeing memory for static data types (Unpredictable behavior)
- Memory protection faults can cause variables to randomly change values

Searches

Linear Search

Searches for an element in list by checking each element of the list until the key element is found or the entire list has been traversed

```
#include <stdio.h>
#include <stdlib.h>

int linearSearch(int * array, int size, int key)
{
    int index;

    for (index = 0; index < size; index++)
        if (array[index] == key) break;

    return (index < size) ? index : -1;
}
```

Example:

(Input) array: [8, 4, 2, 5, 1, 9, 10]

size: 7

key: 6

(Output) -1

(Input) array: [4, 8, 2, 9, 5]

size: 5

key: 2

(Output) 2

Runtime

Best	Average	Worst
O(1)	O(n)	O(n)

Binary Search

Searches for an element in a sorted list by repeatedly dividing the search range in half until key has been found or the search range is 0

```
int binarySearch(int * array, int low, int high, int key)
{
    if (low > high) return -1;

    int mid = (low + high) / 2;

    if (key == array[mid])
        return mid;
    else if (key < array[mid])
        return binarySearch(array, low, mid - 1, key);
    else
        return binarySearch(array, mid + 1, high, key);
}
```

Example:

(Input) array: [1, 2, 3, 4, 5]

low: 0

high: 4

key: 6

(Output) -1

(Input) array: [1, 2, 3, 6, 7, 9, 13]

low: 0

high: 6

key: 3

(Output) 2

Runtime

Best	Average	Worst
O(1)	O(log n)	O(log n)

Time Complexity

Order of Growth: 1 < log n < √n < n < n log n < n^2 < n^3 < … < 2^n < 3^n < … < n! < n^n

Runtime Estimate: Rate * Operations = Time

Recurrence Relations

- O(n) — T(n) = T(n-1) + C
- O(n^2) — T(n) = T(n-1) + n
- O(n log n) — T(n) = T(n-1) + log n
- O(2^n) — T(n) = 2T(n-1) + C
- O(log n) — T(n) = T(n/2) + C
- O(n) — T(n) = T(n/2) + n
- O(n log n) — T(n) = 2T(n/2) + n
- O(n^2) — T(n) = T(n/2) + n^2

Examples of Closed Forms for Summations

$$\sum_{i=1}^{n} c = cn \rightarrow O\left(n\right) \qquad \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \rightarrow O\left(n^2\right)$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \rightarrow O\left(n^3\right) \qquad \sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4} \rightarrow O\left(n^4\right)$$

Recursion

Recursive Structs

A structure where one of the variables in it is a pointer to itself. A structure can not refer itself directly because the structure has not completed its definition prior to using it inside the struct definition so the only way to create a recursive struct is to use pointers

```
struct Node {
    int data;
    struct Node * next;
};
```

Permutations

Every rearrangement of the elements of an ordered list into a one to one correspondence with itself and doing something for each rearrangement

```
void permute(int size, int index, int * used, int * perm)
{
    if (index == size)
    {
        // Do something with the permutation
        return;
    }

    // Loop through all values
    for (int value = 0; value < size; value++)
    {
        // Check if the current value is already used
        if (used[value] == 1) continue;

        // Put this value in the current location of the permutation
        used[value] = 1;
        perm[index] = value;

        // Try all possible permutations after this index
        permute(size, index + 1, used, perm);

        // Remove value from permutation
        used[value] = 0;
        perm[index] = -1;
    }
}
```

Array Sorting

Some functions will use

```
// Swap values that num1 and num2 point to
void swap(int * num1, int * num2)
{
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

Runtime

Sorting Algorithm	Best Case	Average Case	Worst Case
Selection	O(n^2)	O(n^2)	O(n^2)
Insertion	O(n)	O(n^2)	O(n^2)
Bubble	O(n)	O(n^2)	O(n^2)
Quick	O(n log n)	O(n log n)	O(n^2)
Merge	O(n log n)	O(n log n)	O(n log n)

Selection Sort

The selection sort works by first finding the element that will come first in the array, and placing it in the first spot of a "new array" which will just be the first indexes of the array, repeating this process for each next smallest value until the original subarray is empty

Initial	4	2	1	5	3
First Pass	1	2	4	5	3
Second Pass	1	2	4	5	3
Third Pass	1	2	3	5	4
Fourth Pass	1	2	3	4	5

Selection Sort Code

```
void selectionSort(int * array, int size)
{
    int lowest;

    // Repeating process for every sub array[i..size-1]
    // which is the unsorted part of the array
    for (int i = 0; i < size - 1; i++)
    {
        // Find lowest value in the unsorted part of the array
        lowest = i;
        for (int j = i; j < size; j++)
            if(array[j] < array[lowest]) lowest = j;

        // Put lowest value in the unsorted part to the front
        swap(&array[i], &array[lowest]);
    }
}
```

Insertion Sort

The insertion sort will insert the values from the original array one by one to the sorted subarray which is just the beginning of the array

Initial	4	2	1	5	3
First Pass	2	4	4	5	3
Second Pass	1	2	4	5	3
Third Pass	1	2	4	5	4
Fourth Pass	1	2	3	4	5

Insertion Sort Code

```
void insertionSort(int * array, int size)
{
    int i, val;

    // Checking each value that is unsorted
    for(int curr = 1; curr < size; curr++)
    {
        // Takes first value of unsorted part of the array
        val = array[curr];

        // Shifting all values greater than the current value to the right
        for(i = curr - 1; i >= 0 && array[i] > val; i--)
            array[i + 1] = array[i];

        // Inserting the current value in the right
        // spot in the sorted part of the array
        array[i + 1] = val;
    }
}
```

Bubble Sort

The bubble sort moves through the array from the beginning to end, swapping two values if the second value is less than the first, so that the highest value will be at the highest index, repeating this process until array is sorted

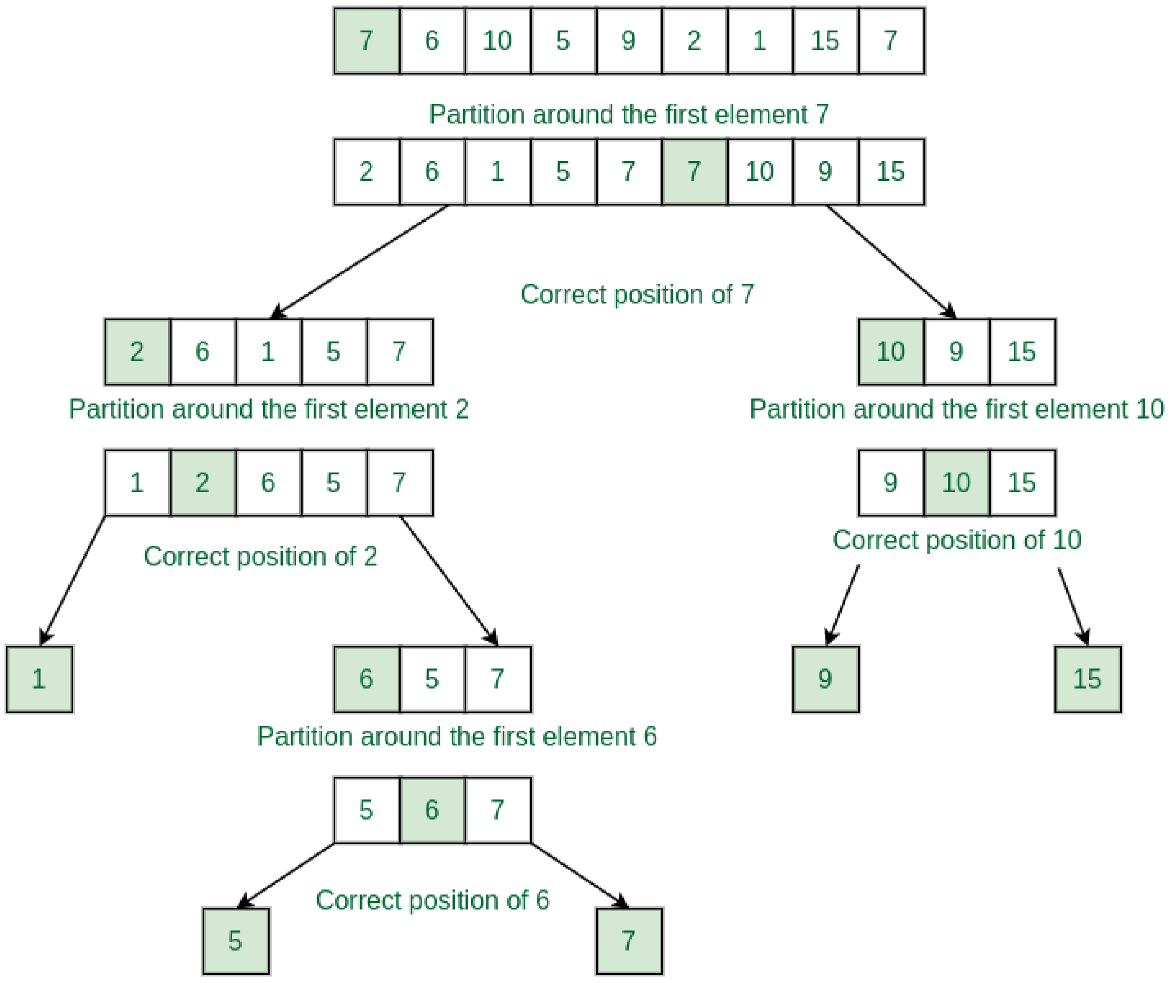
4	2	1	5	3
2	4	1	5	3
2	1	4	5	3
2	1	4	5	3
2	1	4	3	5

Bubble Sort Code

```
void bubbleSort(int * array, int size)
{
    int temp;

    // Decrease size of unsorted subarray after every iteration until size is 1
    for(int max = size; max > 1; max--)
    {
        // Iterate through subarray array[0..max - 1],
        // swapping values if current value is greater than next value
        for (int j = 0; j < max - 1; j++)
            if (array[j] > array[j + 1])
                swap(&array[j], &array[j + 1]);
    }
}
```

Quick Sort

The quick sort works by reducing the array into two smaller pieces and recursively sorting each piece. The array is split by choosing some value in the array to become a pivot. The elements less than the pivot are moved to the left side and elements greater than the pivot are moved to the right side. When a subarray has a size of 1 or 0, the array can be returned without swapping any values

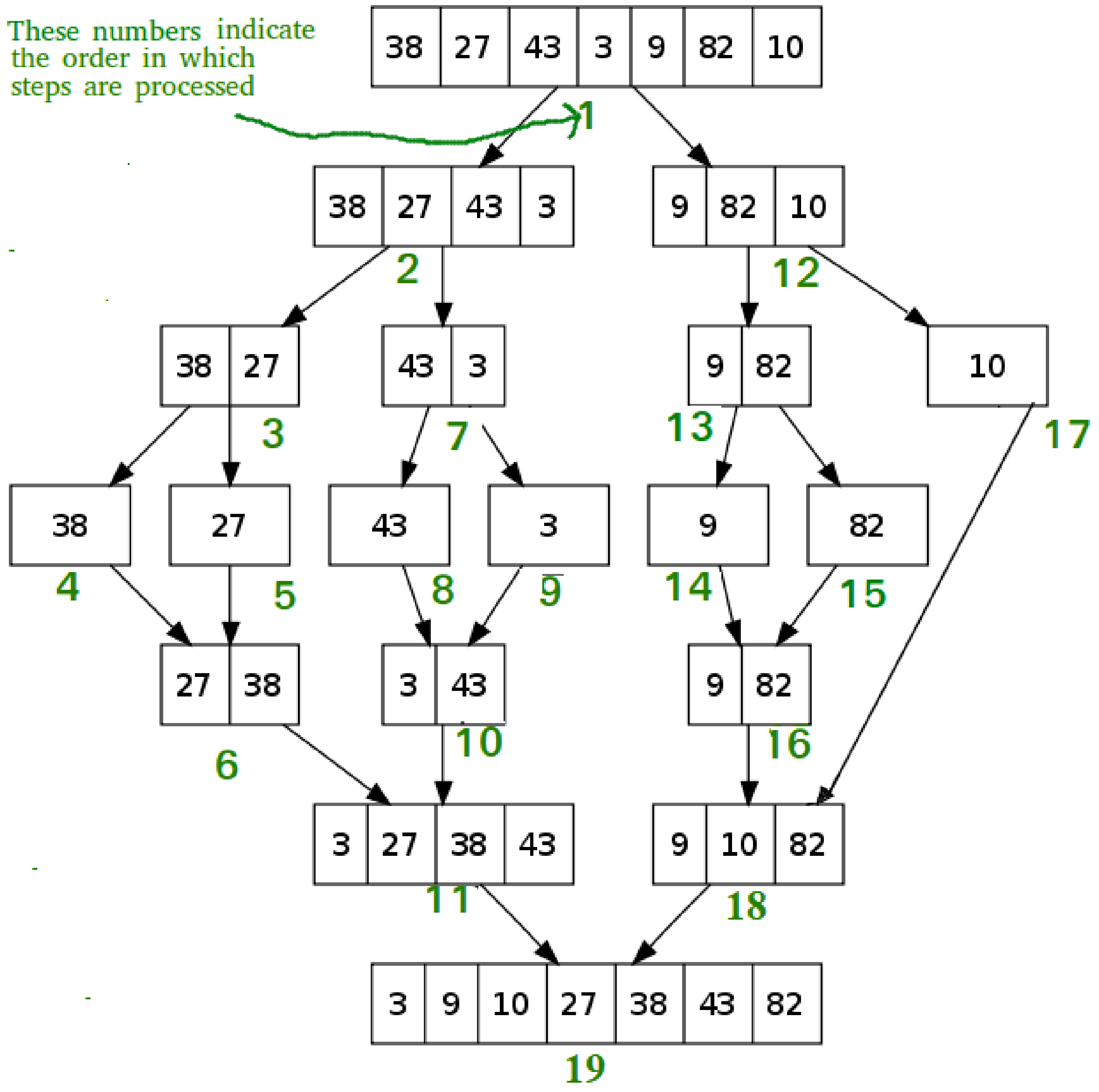


Quick Sort Code

```
// low --> Starting index
// high --> Ending index
int partition(int * array, int low, int high)
{
    int pivot = array[low]; // First element as pivot
    int i = low; // Location for pivot point

    // Traverse each element of the array and compare with pivot
    for (int j = low + 1; j <= high; j++)
    {
        // If current element smaller than pivot is found,
        // swap it with current location for pivot
        if (array[j] <= pivot)
        {
            i++;
            swap(&array[i], &array[j]);
        }
    }

    // Swap the lowest value with the new pivot location
    swap(&array[i], &array[low]);

    // Return the index for the pivot point
    return i;
}

void quickSort(int * array, int low, int high)
{
    // Don't sort anything if subarray has a size of 1 or 0
    if (high - low <= 1) return;

    // Sets pivot point in right place, partioning the smaller and larger values
    int pivot = partition(array, low, high);

    // Separately sort elements before pivot and after pivot
    quickSort(array, low, pivot - 1);
    quickSort(array, pivot + 1, high);
}
```

Merge Sort

The merge sort repeatedly splits the array in half and then merges the two subarrays in a sorted order



Merge Sort Code

```
// Merges and sorts two subarrays of elements where first
// subarray is array[L..M] and second subarray is array[M+1..R]
void merge(int * array, int low, int high, int mid)
{
    int length1 = mid - low + 1;
    int length2 = high - mid;

    // Create temp arrays for left and right subarrays
    int * arrL = (int *) malloc(sizeof(int) * length1);
    int * arrR = (int *) malloc(sizeof(int) * length2);

    // Copy left and right subarray elements into temp arrays
    for(int i = 0; i < length1; i++)
        arrL[i] = array[i + low];

    for(int i = 0; i < length2; i++)
        arrR[i] = array[mid + 1 + i];

    int i = 0; // Initial index of first subarray
    int j = 0; // Initial index of second subarray
    int k = low; // Initial index of merged subarray

    // Merge temp arrays back into array[L..R]
    while (i < length1 && j < length2)
    {
        // If arrL[i] is less than or equal to arrR[j] then add arrL[i] to the
        // next index in the merged subarray and move to the next value in arrL[]
        if (arrL[i] <= arrR[j])
            array[k++] = arrL[i++];
        else
            array[k++] = arrR[j++];
    }

    // Copy any remaining elements in arrL[]
    while (i < length1)
        array[k++] = arrL[i++];

    // Copy any remaining elements in arrR[]
    while (j < length2)
        array[k++] = arrR[j++];

    // Free temp arrays
    free(arrL);
    free(arrR);
}
```

```
// Repeatedly splits array until each subarray has a
// length of 1 or 0 and then merges and sorts the subarrays
void mergeSort(int * array, int low, int high)
{
    // If size of subarray is 0 or 1, do nothing
    if(low >= high) return;

    int mid  = (low + high) / 2; // Midpoint of subarray

    // Sort left and right halves of array
    mergeSort(array, low, mid);
    mergeSort(array, mid + 1, high);

    // Merge both halves together
    merge(array, low, high, mid);
}
```