String functions

- **strcmp(str1, str2)** returns 0 if equal, returns > 0 if str1 ascii value is greater, returns < 0 if str2 is greater.
- **strcpy(dest, src)** copies string from src to dest including null char
- **strcat(str1, str2)** appends str2 onto the end of str1, applies value to str1
- **strlen(str)** returns length of string up to the null char

Memory functions

- **malloc(int bytes)** returns a pointer to the allocated memory, or NULL if the request fails.
- **calloc(number_of_items, size_of_one_item)** allocates the requested memory and returns a pointer to it, same as malloc but returns zero
- **realloc(*ptr, newsize)** returns a pointer to the newly allocated memory, or NULL if the request fails
- **free(*ptr)** deallocates memory of pointer

Searches

- Linear – checks each and every item slow, O(N) runtime
- Binary – needs a ordered array, splits array using midpoints to get closer and closer to the value
  Runtime of binary is sort + (binary O NlogN)

Runtimes

Algorithm	Best	Average	Worst
Insertion	$\Theta(N)$	$\Theta(N^2)$	$\Theta(N^2)$
Quick	$\Theta(N\log(N))$	$\Theta(N\log(N))$	$\Theta(N^2)$
Bubble	$\Theta(N)$	$\Theta(N^2)$	$\Theta(N^2)$
Selection	$\Theta(N^2)$	$\Theta(N^2)$	$\Theta(N^2)$
Merge	$\Theta(N\log(N))$	$\Theta(N\log(N))$	$\Theta(N\log(N))$

```c
int binarySearch(int arr[], int l, int r, int x){
    if (r >= l) {
        int mid = l + (r - l) / 2;
        // If the element is present at the middle itself
        if (arr[mid] == x)
            return mid;
        // If element is smaller than mid, then it can only be present in left subarray
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
        // Else the element can only be present in right subarray
        return binarySearch(arr, mid + 1, r, x);
    }
    // We reach here when element is not present in array
    return -1;
}
```

Sorts

- Insertion – start at beginning keep checking if the next is smaller then current if so insert/swap, even multiple positions if it needs to jump multiple times.

```
void insertionSort(int arr[], int n){
    int i, key, j;
    for (i = 1; i < n; i++){
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key){
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

- Quick – uses pivots, a item in the array that is in correct position of final array all items to left are smaller all the right are larger

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex - 1)
    quickSort(array, pivotIndex, rightmostIndex)

partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
  swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

- Bubble – bubbles the highest values to the end looping whole array each time essentially.

```
void bubbleSort(int arr[], int n){
    int i, j;
    for (i = 0; i < n - 1; i++)
        // Last i elements are already in place
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
}
```

- Selection – searches array from beginning for smaller item than current item and swaps if so

```
void selectionSort(int arr[], int n) {
```

```
    int i, j, min_idx;
    // One by one move boundary of
    // unsorted subarray
    for (i = 0; i < n-1; i++){
        // Find the minimum element in
        // unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++){
          if (arr[j] < arr[min_idx]) min_idx = j;
        }
        // Swap the found minimum element
        // with the first element
        if (min_idx!=i) swap(&arr[min_idx], &arr[i]);
    }
}
```

- Merge – splits into smaller arrays of two elements each sorted, then merges into bigger and bigger arrays

Recursion

- **Combinations – distinct items** - the elements of the subset can be listed in any order
- **Permutations – distinct arrangements** - the elements of the subset are listed in a specific order

Merge

```
void merge(int arr[], int left, int mid, int right){
    int i, j, k;
    int s1 = mid - left + 1;
    int s2 = right - mid;

    int left_arr[s1], right_arr[s2];
    for (i = 0; i < s1; i++)
        left_arr[i] = arr[left + i];
    for (j = 0; j < s2; j++)
        right_arr[j] = arr[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    while (i < s1 && j < s2) {
        if (left_arr[i] <= right_arr[j]) {
            arr[k] = left_arr[i];
            i++;
        }
        else {
            arr[k] = right_arr[j];
            j++;
        }
        k++;
    }
```

```
    while (i < s1) {
        arr[k] = left_arr[i];
        i++;
        k++;
    }
    while (j < s2) {
        arr[k] = right_arr[j];
        j++;
        k++;
    }
}

void merge_sort(int arr[], int left, int right){
    if (left < right) {

        // finding the mid value of the array.
        int mid = l + (right - left) / 2;

        // Calling the merge sort for the first half
        merge_sort(arr, left, mid);

        // Calling the merge sort for the second half
        merge_sort(arr, mid + 1, right);

        // Calling the merge function
        merge(arr, left, mid, right);
    }
}
```