Nicolas Ayala
nmayala@ucsc.edu

Pre-Lab 1)

1) 5 rounds, the number of swaps it takes to bring the 5 to the beginning
2) O(N^2), if the smallest element is at the end of the list

Pre-Lab 2)

1) Worst time complexity is O(N^2), if gap is simply 1 then it is identical to bubble sort
2) By choosing a sequence of gaps which has either been proven to have a good time complexity, or has been empirically determined to be fast.

Pre-Lab 3)

1) Behavior depends on the pivot choice, if the chosen pivot partitions the array by about a half every time then the complexity is O(Nlog(N)). For randomized input arrays, the pivot is expected to partition the arrays pretty evenly. If the pivot is chosen to be the midway element of the array, then inputting a sorted array will produce best case, rather than worst case, behavior. If the pivot is randomly chosen in the array, then the time complexity is almost always O(Nlog(N)) for all inputs.

Pre-Lab 4)

1) Binary search has a time complexity of O(log(N)), repeating this for every element means repeating it N times. This makes the complexity O(Nlog(N))

Pre-Lab 5)

1) Make each sorting algorithm return an array of size 2 (it will really return a pointer) which holds the number of comparisons it made and the number of moves it made

Design of program

There will be a main function that accepts 8 different options, which may appear in any combination, or it may have no options at all. The options are

- -b, employ bubble sort on an array
  - sets bubble_flag to true;
- -s, employ shell sort on an array
  - sets shell_flag to true;
- -q, employ quicksort on an array
  - sets quick_flag to true;
- -i, employ binary insertion sort on an array
  - sets insertion_flag to true;
- -A, the program will employ every sort
  - sets bubble_flag to true;
  - sets shell_flag to true;
  - sets quick_flag to true;

sets insertion_flag to true;

-p <value>, sets the number of elements to print to <value>, default is 100

sets n to <value>, n is 100 initially

-r <value>, sets the random seed to <value>, default is 8222022

sets seed to <value>, seed is 8222022 initially

-n <value>, sets the number of elements to <value>, default is 100

sets c to <value>, c is 100 initially

An array of size c is initialized using

(uint32_t *)calloc(c,sizeof(uint32_t))

The random number generator from stdlib.h has its seed set using

srand(seed)

The array is filled, each value is set to

rand() & 0x3FFFFFFF

Then, for each sort with a true flag, the array is copied and passed to them to be sorted.

For each sort, it must return a pointer to an array of 2 integers where the first number is the number of comparisons, and the second number is the number of moves.

Those numbers will be printed, as well as the first n (set by -p flag) elements of the sorted array

The sorts are implemented in their own .c files and header files, the pseudo code for each is as follows

```
bubble_sort(int[] arr){
	int a = length(arr) -1;
	while(a>0){
		int b = a;
		a = 0;
		for(int i = 0;i<b;i++){
			if(arr[i] > arr[i +1]){
				swap arr[i], arr[i+1] //using xor swap
				a = i;
			}
		}
	}
}

shell_short(int[] arr){
	for(int gap = floor(5*length(arr)/11); gap>0;gap = floor(5*gap/11) ){
		for(int r = 0; r < gap; r ++){
			int a = length(arr) - gap + r;
```

```
                    while(a>0){
                            int b = a;
                            a = 0;
                            for(int i = r;i<b;i += gap){
                                    if(arr[i] > arr[i + gap]){
                                            swap arr[i], arr[i+1] //using xor swap
                                            a = i;
                                    }
                            }
                    }
            }
        }
}

quick_sort(int[] arr){
        if(length(arr) < 2){
                return;
        }
        int left = 0;
        int right = length(arr) - 1;
        int[length(arr)] new;
        for(int i = 1; i<length(arr); i++){     //0th element is index
                if(arr[i] < arr[0]){
                        new[left] = arr[i];
                        left++;
                }else{
                        new[right] = arr[i];
                        right--;
                }
        }
        new[left] = arr[0]
        arr = new;//copy array, do not simply change direction of pointer arr
        if(left>1){
                leftarray = pointer to (0 to left) elements of arr;
                quick_sort(leftarray);
        }
        if(right < length(arr) - 2){
                rightarray = pointer to (right to length(arr)-1) elements of arr;
                quick_sort(rightarray);
        }
}
```

```
binary_insertion_sort(int[] arr){
	for(int i = 1; i < length(arr); i++){
		int left = 0;
		int right = i;
		while(left<right){
			mid = (left+right)/2
			if(arr[i]>=arr[mid])
				left = mid +1;
			else
				right= mid;
		}
		for(int j = i; j>left; j++)
			swap( arr[j], arr[j-1])
	}
}

merge_sort(int[] arr){
	leftarr = first half of arr
	rightarr = second half of arr
	merge_sort(leftarr);
	merge_sort(rightarr);

	int a = 0;//index for left arr
	int b = 0;//index for right arr
	int i = 0;//index for main array
	while(a<length(leftarr) && b<length(rightarr)){
		if(leftarr[a] < rightarr[b]){
			arr[i] = leftarr[a];
			a++;
		}else{
			arr[i] = rightarr[b];
			b++;
		}
		i++;
	}
	if(b == length(rightarr){ // we reached the end of the second array first
		while(i<length(arr){
			arr[i] = leftarr[a];
			i++;
			a++;
		}
	}
	//if we reached the end of the first array first, then nothing needs to be done
```

```
        //because the second array is already where it should be
}
```