DESIGN

Wenyu Liang

February 6, 2023

1 Program Description

This program will implement Shell Sort, Batcher Sort, Heap Sort, and recursive Quicksort. In addition, the number of comparisons and swaps will also be returned. Users of this program are expected to provided several command-line arguments to sort arrays using specific algorithms. Inside the program, a set is used to track which command-line options are specified.

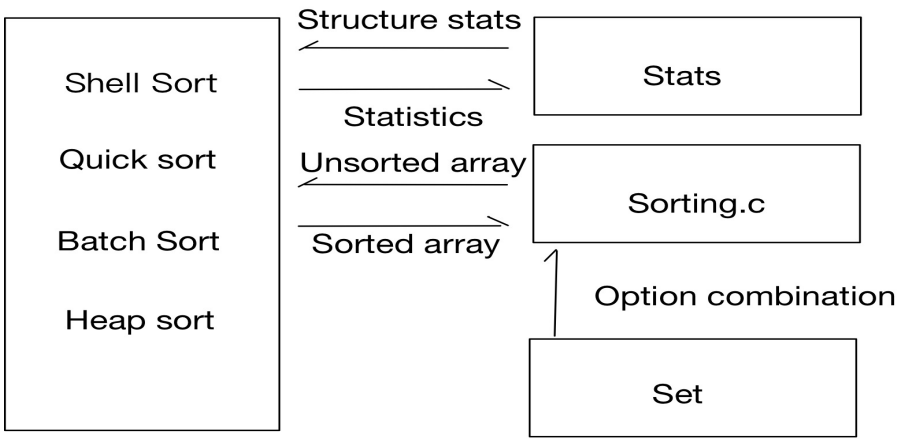
2 Files to be included in directory asgn3

1. **batcher.c**: implements BatcherSort
2. **batcher.h**: specifies the interface to batcher.c.
3. **shell.c**: implements ShellSort
4. **shell.h**: specifies the interface to shell.c.
5. **heap.c**: implements HeapSort
6. **heap.h**: specifies the interface to heap.c.
7. **quick.c**: implements recursive Quicksort
8. **quick.c**: specifies the interface to quick.c
9. **set.c**: implements and specifies the interface for the set ADT.
10. **stats.c**: implements the statistics module.
11. **stats.h**: specifies the interface to the statistics module.
12. **sorting.c**: contains main() and may contain any other functions necessary to complete the assignment.
13. **Makefile**: This file is provided and directs the compilation process of this program.
14. **README.md**: Describe how to use the script and Makefile.

15. **DESIGN.pdf**: Describes design for the program with pseudocode and visualization.

16. **WRITEUP.pdf**: Describes how the program works in detail, codes included.

3 Structure



4 Pseudocode

1. **sorting.c**

```
Stats stat = {0, 0};
Stats *stats_ptr = &stats;
Set set;
set = set_empty();
int fill_random(*Array, n, seed ){
    for i in range(n){
        Array[i] = random() & 0x3FFFFFFF used to bit-masked
    }
}

int print_array(*Array, n, seed){
    for i in Array{
        print(Array[i];
        if (i+1)%5 == 0:
            print "\n"
    }
}
```

```
int main(){
    while ((opt = getopt(argc, argv, OPTIONS)) != -1){
      switch (opt)
        case 'a': set = set_insert(set, 0); break;
        case 'h': set = set_insert(set, 1); break;
        case 'b': set = set_insert(set, 2); break;
        case 's': set = set_insert(set, 3); break;
        case 'q': set = set_insert(set, 4); break;
        case 'r': r = (uint32_t) strtoul(optarg, NULL, 10); break;
        case 'n': n = (uint32_t) strtoul(optarg, NULL, 10); break;
        case 'p': p = (uint32_t) strtoul(optarg, NULL, 10); break;
        case 'H': usage(argv[0]); break;
    }
    if (set_member(set, 0) == 1) //all is set {
        for (i in range(0:5) {
            set = set_remove(set, i);
        for sort in (shell, quick, barcher, heap){
            calloc();
            fill_random();
            run *.sort;
            reset(stats_ptr);
            free(array)
        }
  } else:
        uint8_t bit_move = 0;
        uint8_t check_opt = 1;
        while (bit_move < 5 ){
            if bit_move == (1,2,3,4){
                run corresonponding sort;
                reset(stats_ptr);
                    print_array(Array, n, p);
                free(Array)
            }
        }
```

2. **quick.c**

```
int partition(*stats, *A,low, high){
  lower to left;
  greater to right;
  cmp() when elements are compared;
  swap() when elements are swapped;
  return (partition index);
}
void quick_sorter(*stats, *A, low, high){
  if (low < high){
```

```
    partition(stats, A, low, high);
    quick_sorter(stats, A, leftstart, leftend);
    quick_sorter(stats, A, rightstart, rightend);
  }
}
void quick_sort(Stats *stats, uint32_t *A, uint32_t n){
  quick_sorter(stats, A, 1, n);
}
```

3. **shell sort**

```
void shell_sort(*stats, *arr, n) {
    for gap in gaps{
        for (gap:n){
            j = 1
            while j >= gap && temp < arr[j-gap]{
                arr[j] = arr[j-gap]
                j-=gap
            }
        }arr[j] =temp
    }
}
```

4. **batcher sort**

```
void comparator(*stats, *A, x, y){
    if (A[x] > A[y]) {
        swap(stats, &A[x], &A[y]);
}

void batcher_sort(*stats, *arr, n){
    for (i=0 : log2(n)) //log2(n) implemented by n<<1
        for (j= 0 : n - 1 by 2^i)
            comparator(arr, j, j + 2^i - 1)
}
```

5. **heap sort**

```
int max_child(*stats, *A, first, last){
    compare the children;
    return the larger one
}
void fix_heap(*stats, *A, first, last){
    while(mother <= last / 2 && found == 0){
        swap parent and max_child if structure is wrong
    }
}
```

```
void build_heap(*stats, *A, first, last){
    for(uint32_t father = last/2; father > first - 1; father--){
                fix_heap(stats, A, father, last);
}

void heap_sort(*stats, *A, first, last){
    first = 1;
        last = n;
        build_heap(stats, A, first, last);
        for(uint32_t leaf = last; leaf > first; leaf--){
                swap(stats, &A[first - 1], &A[leaf - 1]);
                fix_heap(stats, A, first, leaf-1);
}
```

5 Credit

I used gaps.h.py to generate gap sequence and learn from calloc_example.c to generate random numbers and dynamically allocate memory.