Assignment 3 DESIGN.pdf

Ning Jiang

Description of Program: In this assignment we need to do 4 sorting methods: Insertion Sort, Batcher Sort, Heap Sort, and recursive Quicksort, and calculate the number of comparisons and moves for each Sort.

Files to be included in directory "asgn3":

1. • batcher.c implements Batcher Sort.
   • batcher.h specifies the interface to batcher.c.
   • insert.c implements Insertion Sort.
   • insert.h specifies the interface to insert.c.
   • heap.c implements Heap Sort.
   • heap.h specifies the interface to heap.c.
   • quick.c implements recursive Quicksort.
   • quick.h specifies the interface to quick.c.
   • set.h implements and specifies the interface for the set ADT.
   • stats.c implements the statistics module.
   • stats.h specifies the interface to the statistics module.
   • sorting.c contains main() and may contain any other functions necessary to complete the assignment.

2. Makefile:
• CC = clang must be specified.
• CFLAGS = -Wall -Wextra -Werror -Wpedantic must be specified.
• make must build the sorting executable, as should make all and make sorting.
• make clean must remove all files that are compiler generated.
• make format should format all your source code, including the header files.

3. README.md: describe how to use your program and Makefile. It should also list and explain any command-line options that your program accepts. Any false positives reported by scan-build should be documented and explained here as well.

4. DESIGN.pdf: This design document must describe your design and design process for your program with enough detail.

5. WRITEUP.pdf:
• What you learned from the different sorting algorithms. Under what conditions do sorts perform well? Under what conditions do

sorts perform poorly? What conclusions can you make from your findings?

• Graphs explaining the performance of the sorts on a variety of inputs, such as arrays in revers order, arrays with a small number of elements, and arrays with a large number of elements.

Your graphs must be produced using either gnuplot. You will find it helpful to write a script to handle the plotting. As always, awk will be helpful for parsing the outputof your program.

• Analysis of the graphs you produce.

Pseudocode / Structure:

From asgn3.pdf

Insertion Sort

Insertion Sort in Python

```python
def insertion_sort(A: list):
    for i in range(1, len(A)):
        j = i
        temp = A[i]
        while j > 0 and temp < A[j - 1]:
            A[j] = A[j - 1]
            j -= 1
        A[j] = temp
```

Heap sort

Heap maintenance in Python

```python
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left

def fix_heap(A: list, first: int, last: int):
    found = False
    mother = first
    great = max_child(A, mother, last)

    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
```

Heapsort in Python

```python
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
        fix_heap(A, father, last)

def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, first, leaf - 1)
```

Quick Sort

Partition in Python

```python
def partition(A: list, lo: int, hi: int):
    i = lo - 1
    for j in range(lo, hi):
        if A[j - 1] < A[hi - 1]:
            i += 1
            A[i - 1], A[j - 1] = A[j - 1], A[i - 1]
    A[i], A[hi - 1] = A[hi - 1], A[i]
    return i + 1
```

Recursive Quicksort in Python

```python
# A recursive helper function for Quicksort.
def quick_sorter(A: list, lo: int, hi: int):
    if lo < hi:
        p = partition(A, lo, hi)
        quick_sorter(A, lo, p - 1)
        quick_sorter(A, p + 1, hi)

def quick_sort(A: list):
    quick_sorter(A, 1, len(A))
```

Batcher’s sort

Merge Exchange Sort (Batcher's Method) in Python

```python
def comparator(A: list, x: int, y: int):
    if A[x] > A[y]:
        A[x], A[y] = A[y], A[x]

def batcher_sort(A: list):
    if len(A) == 0:
        return

    n = len(A)
    t = n.bit_length()
    p = 1 << (t - 1)

    while p > 0:
        q = 1 << (t - 1)
        r = 0
        d = p

        while d > 0:
            for i in range(0, n - d):
                if (i & p) == r:
                    comparator(A, i, i + d)
            d = q - p
            q >>= 1
            r = p

        p >>= 1
```

Main file: Sorting.c

```
#include <stdio.h>
#include <inttypes.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <stdint.h>
#define OPTION "ahbiqHr:n:p:"
```

```
#include <unistd.h>
#include <math.h>
#include <stdint.h>
#include "set.h"
#include "batcher.h"
#include "heap.h"
#include "insert.h"
#include "quick.h"
#include "stats.h"
// include all the files we need to use

#define OPTION "ahbiqHr:n:p:"
//define the case options

#define ASA 1
#define HS 2
#define BS 3
#define IS 4
#define QS 5
// include the variables that we need to use with set method.

// define the usage page.

// define main function.
int main(int argc, char **argv){
        The default number of elements  = 100
        The default size = 100.
        The default seed = 13371453.
        srand(seed);
        Set arguments = empty_set();
        int opt = 0;
while ((opt = getopt(argc, argv, OPTION)) != -1):
     switch (opt):
     case 'a':
          arguments = insert_set(ASA, arguments); break;
     case 'h':
          arguments = insert_set(HS, arguments); break;
     case 'b':
           arguments = insert_set(BS, arguments); break;
     case 'i':
           arguments = insert_set(IS, arguments); break;
     case 'q':
            arguments = insert_set(QS, arguments); break;
     case 'r':
```

```
            seed = (uint32_t) strtoul(optarg, NULL, 10); break;
            srand(seed);
      case 'n':
            size = (uint32_t) strtoul(optarg, NULL, 10); break;
      case 'p':
           elements =(uint32_t) strtoul(optarg, NULL, 10); break;
      case 'H':
            usage(argv[0]); return EXIT_FAILURE;



if(size < elements):
           elements = size;
// If the size of the array is less than the specified number of
elements to print, print out the entire array and nothing more.

uint32_t *A;
    A = (uint32_t*)malloc(size * sizeof(uint32_t));
    for(int i = 0; i < size; i = i + 1):
           A[i] = rand() & (int)(pow(2,30) - 1);

    // create a random list random() should be bit-masked to fit
in 30 bits.

uint32_t n = size; // n = len(array)

Stats *sortingstats = malloc(sizeof(Stats));

If arguments = 2 in member_set:
           reset(sortingstats); // first reset the sortingstats to
                                // let moves and compares to 0
           heap_sort(sortingstats, A, n); // run heap_sort
           print("Heap  Sort,  n  elements,  sortingstats->moves
moves, sortingstats->compares compares\n");
           for (int j = 0; j < elements; j = j + 1):
               printf(A[j] PRIu32);
               if((j + 1) % 5 == 0):
                   printf("\n");
           free(A); // remember to free the random list after using
malloc.

Else If arguments = 3 in member_set:
          Do the same thing as before instead, run batcher_sort
   function, remember to free A after done with using the random
```

list.

Else If arguments = 4 in member_set:
Do the same thing as before instead, run insertion_sort function, remember to free A after done with using the random list.

Else If arguments = 5 in member_set:
Do the same thing as before instead, run quick_sort function, remember to free A after done with using the random list.

Else If arguments = 1 in member_set:
Do the same thing as before instead, run all of the four sorting functions.remember to free the random list and create it again after done with using the random list. Finally free the sortingstats.

Notes:
1. How to use random and srandom to create a pseudorandom list?
2. How to meet the requirements: random() should be bit-masked to fit in 30 bits.

```
uint32_t *A; // create a pointer
    A = (uint32_t*)malloc(size * sizeof(uint32_t));
/* Use malloc() and free() beacause we can't do something like arr[size] in C instead, we should use malloc to allocate a block of memory on the heap and free it after we done with this radom list.*/
    for(int i = 0; i < size; i = i + 1):
           A[i] = rand() & (int)(pow(2,30) - 1);
             /*The pseudorandom numbers generated by random() should be bit-masked to fit in 30 bits. */
```

3. how to calculate bit_length in C?

```
int t = log2(n) + 1
```

Error Handing:

1. I am confused with how to define a move or a compare, I attend the tutoring section on Tuesday 9:00 am to 11:00 am with Miles. I know how to use stats without cover, and the compare value increases only when the elements in the sequence are swapped, and

the move value increases when the elements in the series are moved

2. I am confused with how to use set and I attend the section on Thursday to get some information.

Credit:
1. I learned how to create a random list.
2. I learned how to create a logarithmic scale for the x and y axis.
3. I learned how to use malloc() and free() to allocate a block of memory on the heap.