Assignment 3 Design Document

Alex Yeh

February 5, 2023

1 Description of Assignment

I have implemented four sorting algorithms which are shell sort, batcher sort, heap sort, and quick sort and have utilized another program called sorting.c to test the sorts.

2 Files to be included in directory "asgn2":

1. batcher.c
    - This C file implements Batcher sort.
2. batcher.h
    - This header file specifies the interface to batcher.c.
3. shell.c
    - This C file implements Shell sort.
4. shell.h
    - This header file specifies the iterface to shell.c.
5. heap.c
    - This C file implements Heap sort.
6. heap.h
    - This header file specifies the interface to heap.c.
7. quick.c
    - This C file implements recursive Quicksort.
8. quick.h

- This header file specifies the interface to quick.c.

9. set.c
    - This C file implements bit-wise Set operations.
10. set.h
    - This header file implements and specifies the interface for the set ADT.
11. stats.c
    - This C file implements the statistics module.
12. stats.h
    - This header file specifies the interface to the statistics module.
13. sorting.c
    - This C file contains main() and contains other functions necessary to complete the assignment.
14. Makefile
    - This file directs the compilation process of sorting.c.
15. README.md
    - This file is in Markdown format and describes how to use my program and Makefile. It also lists and explains the different command-line options that my program accepts.
16. DESIGN.pdf
    - This file is a PDF version of this design document for assignment 3. It describes my design and design process for my program with pseudocode and images.
17. WRITEUP.pdf
    - This file is a PDF version of my writeup for assignment 3. I wrote about what I learned from the different sorting algorithms. I also included graphs explaining the performance of the sorts on a variety of inputs and an analysis of the graphs I produced.

3 Pseudocode/Description of Programs

3.1 shell.c

Shell Sort in Python

```
def shell_sort(arr):
    for gap in gaps:
        for i in range(gap, len(arr)):
            j = i
            temp = arr[i]
            while j >= gap and temp < arr[j - gap]:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
```

Above is a picture of the python pseudocode for shell sort from the assignment 3 pdf which I used to write shell.c. Because I translated the code from python to C, I will not be providing all of the pseudocode of my C code because is practically the same thing. I will, however, provide pseudocode below of what I needed to implement in my function to translate the first for-loop of the python pseudocode.

For-loop to iterate through GAPS
    Set uint32_t gap to gaps of index i of the for-loop

To use “GAPS” and “gaps”, I had to download the python file from the assignment 3 resources repository on git and run it in order to generate gaps.h. I then was able to include the gaps.h file in my shell.c file. Another thing that is different from the pseudocode and my code is that shell_sort has a three parameters: Stats *stats (to keep track of the stats), uint32_t *arr (the array to be sorted), and uint32_t length (the length of the array to be sorted). To keep track of the number of moves and comparisons made by shell sort, I included stats.h and used the move function and cmp function in stats.c. The move function adds 1 to the number of moves and the cmp function adds 1 to the number of comparisons.

3.2 heap.c

Heap maintenance in Python

```
1  def max_child(A: list, first: int, last: int):
2      left = 2 * first
3      right = left + 1
4      if right <= last and A[right - 1] > A[left - 1]:
5          return right
6      return left
7
8  def fix_heap(A: list, first: int, last: int):
9      found = False
10     mother = first
11     great = max_child(A, mother, last)
12
13     while mother <= last // 2 and not found:
14         if A[mother - 1] < A[great - 1]:
15             A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
16             mother = great
17             great = max_child(A, mother, last)
18         else:
19             found = True
```

Heapsort in Python

```
1  def build_heap(A: list, first: int, last: int):
2      for father in range(last // 2, first - 1, -1):
3          fix_heap(A, father, last)
4
5  def heap_sort(A: list):
6      first = 1
7      last = len(A)
8      build_heap(A, first, last)
9      for leaf in range(last, first, -1):
10         A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
11         fix_heap(A, first, leaf - 1)
```

Above is a picture of the python pseudocode for heap sort from the assignment 3 pdf which I used to write heap.c. Because I translated the code from python to C, I will not be providing the pseudocode of my C code because is practically the same thing. One thing that is different from the pseudocode and my code is that each function uses an extra parameter called Stats *stats (to keep track of the stats) and heap_sort uses another parameter called uint32_t n which is the length of the array to be sorted. Another thing that is different in my code is in my heap_sort function, I set last equal to n for line 7 of the pseudocode because n represents the length of the array in the function's parameter. To keep track of the number of moves and comparisons made by heap sort, I included stats.h and used the swap function and cmp function in stats.c. The swap function adds 3 to the number of moves and the cmp function adds 1 to the number of comparisons.

3.3 quick.c

Partition in 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

```
# 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))
```

Above is a picture of the python pseudocode for quick sort from the assignment 3 pdf which I used to write quick.c. Because I translated the code from python to C, I will not be providing the pseudocode of my C code because is practically the same thing. One thing that is different from the pseudocode and my code is that each function uses an extra parameter called Stats *stats (to keep track of the stats) and quick_sort uses another parameter called uint32_t n which is the length of the array to be sorted. To keep track of the number of moves and comparisons made by quick sort, I included stats.h and used the swap function and cmp function in stats.c. The swap function adds 3 to the number of moves and the cmp function adds 1 to the number of comparisons.

3.4 batcher.c

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
```

Above is a picture of the python pseudocode for batcher sort from the assignment 3 pdf which I used to write batcher.c. Because I translated the code from python to C, I will not be providing the pseudocode of all my C code because is practically the same thing. One thing that is different from the pseudocode and my code is that each function uses an extra parameter called Stats *stats (to keep track of the stats) and batcher_sort uses another parameter called uint32_t n which is the length of the array to be sorted. Also, to translate line 10 of the python pseudocode, I had to create a function to calculate the bit length. The pseudocode for that is below.

uint8_t bit_length(uint32_t x)
    uint8_t t = 0;
    While n doesn't equal 0
        Right shift X by 1
        Increment t by 1
    Return t

To keep track of the number of moves and comparisons made by quick sort, I included stats.h and used the swap function and cmp function in stats.c. The swap function adds 3 to the number of moves and the cmp function adds 1 to the number of comparisons.

3.5 set.c

set_empty(void)
return 0

set_universal(void)
Set each bit to 1

set_insert(Set s, uint8_t x)
Use the OR operation and set the first operand to set s and the second operand to the value by left shifting 1 by x number of bits

set_remove
Use the AND operation and set the first operand to set s and the second operand to the negation of 1 shifted to the same position as x

set_member
Use the AND operation and set the first operand to set s and the second operand to the value obtained by left shifting 1 by x number of times

set_union
Use the OR operation and set the first operand to set s and the second operand to set t

set_intersect
Use the AND operation and set the first operand to set s and the second operand to set t

set_difference
Use the AND operation and set the first operand to set s and the second operand to the negation of set t

set_complement
Flip all the bits of set s using the NOT operator

3.6 sorting.c

define OPTIONS “ahbsqr:n:p:H”

Create static void function called “program_usage” to print out help message

Create static void function called “create_array” to create a new array
Use srandom and seed to get array of pseudorandom numbers
For-loop from i=0 while i is less than size

Bit-mask the pseudorandom numbers to fit in 30 bits using bit-wise AND

The following function is commented out in my code, but I uncommented it and commented the function above when I needed to create an array with numbers in reverse order for my writeup.
Create static void function called create_reverse_array to create an array of numbers in reverse order

Use srandom and seed to get array of pseudorandom numbers
For-loop from i=0 while i is less than size
Set the index of the array equal to the size of the array minus the index

int main(int argc, char **argv)
Set default uint32_t seed to 13371453
Set default uint32_t size to 100
Set default uint32_t number of elements to 100
Set int opt equal to 0
Initialize random array
Initilaize enumeration used for sets
Define character array of sort names
Initialize Set s to set_empty function to create an empty set
While loop while opt is not equal to -1
Switch statement for opt
Case 'a':
Set s to set_insert with parameters "s" and "HEAP" from enumeration
Set s to set_insert with parameters "s" and "BATCHER" from enumeration
Set s to set_insert with parameters "s" and "SHELL" from enumeration
Set s to set_insert with parameters "s" and "QUICK" from enumeration
break
Case 'h':
Set s to set_insert with parameters "s" and "HEAP" from enumeration
break
Case 'b':
Set s to set_insert with parameters "s" and "BATCHER" from enumeration
break
Case 's':
Set s to set_insert with parameters "s" and "SHELL" from enumeration
break
Case 'q':
Set s to set_insert with parameters "s" and "QUICK" from enumeration
break
Case 'r':
Set seed to strtoul function with parameters optarg, NULL, and 10 (this repre-

sents base 10)
break
Case 'n':
Set size to strtoul function with parameters optarg, NULL, and 10 (this represents base 10)
break
Case 'p':
Set number of elements to strtoul function with parameters optarg, NULL, and 10 (this represents base 10)
break
Case 'H':
Call program_usage function
return 0
Case 'default':
Call program_usage function
return 1
If statement for if set s equals 0
Print "Select at least one sort to perform." to stderr with a new line after
Call program_usage function
return 1
If statement for if the number of elements is greater than the size
Set the number of elements to the size
Allocate memory for the random array using malloc. Use size and sizeof(uint32_t) as the two parameters
For loop from i = 0 while i is less than 4
If statement for if the set_member function with parameters "s" and "i" from enumeration is true
Reset stats
Call create_array function with parameters "random_array", "size", and "seed"
The function called below is commented out in my code but I uncommented it when I needed to create an array with numbers in reverse order for my writeup and commented the function call above:
Call create_reverse_array function with parameters "random_array", "size", and "seed"
If statement for if i is equal to "SHELL"
Call shell_sort function with parameters "&stats", "random_array", and "size"
Else if statement for if i is equal to "HEAP"
Call heap_sort function with parameters "&stats", "random_array", and "size"
Else if statement for if i is equal to "BATCHER"
Call batcher_sort function with parameters "&stats", "random_array", and "size"
Else if statement for if i is equal to "QUICK"
Call quick_sort function with parameters "&stats", "random_array", and "size"
Print out the sort name, number of elements, the number of moves, and the number

of comparisons
        Loop through each element in the sorted array
            If statement for if i modulo 5 is equal to 0 and the iteration is not equal to 0
                Print a new line for every 5 elements
            Print each element in the sorted array with a width of 13
            If statement for if the index is equal to the number of elements minus 1
                Print a new line after the last element
    Free the memory allocated for the random array using free
    Return 0

3.7 Makefile

Set EXECBIN equal to “sorting”
Set CC equal to “clang”
Set CFLAGS equal to “-Wall -Wpedantic -Werror -Wextra -Ofast -g -gdwarf-4”
Set SOURCES equal to all available .c files
Set OBJECTS equal to all available .o files
Set PHONY targets to “all”, “clean” and “format”
Set “all” to EXECBIN
Set “sorting” dependencies on all object files
Set default rule for creating .o from .c files
Set rule for make clean
Set rule for make format