Algorithms Are Everything.

dlee181

February 2023

1 Introduction

This design doc will cover multiple sorting algorithms including

1. Heap Sort
2. Batcher Sort
3. Shell Sort
4. Quick Sort

Although this is the focus of the activity, we also learned how to implement a set-based parsing system in our main testing harness that takes in parameters that trigger set functions that allow us to modify a set integer so that instead of taking 8 bits for a true and false, instead we can use an integer which will store 32 bits and can be modified to true or false for each bit.

2 Files to be included in the directory for "asgn3"

1. batcher.c
2. batcher.h
3. quick.c
4. quick.h
5. heap.c
6. heap.h
7. shell.c
8. shell.h
9. set.c
10. set.h

11. stats.c
12. stats.h
13. sorting.c
14. gaps.h
15. Makefile
16. README.md
17. DESIGN.pdf
18. WRITEUP.pdf

3 Pseudo-Code / Structure

Shell Sort: $\mathcal{O}(n^{5/3})$
Shell sort is similar to insertion sort (and technically is at some point) but the main point of shell sort is that similar to insertion, it takes a pair of elements, but instead of them being adjacent to each other, instead it takes gaps based off of the Pratt sequence and moves things out of place closer to their correct position in the sorted list. Then after each iteration, the gap gets smaller becoming more and more precise.

Steps:

1. Begin by including:
   (a) shell header file
   (b) gaps header file which will contain the gap sequence
   (c) stats header in case you want to recreate the exact same project
   (d) standard int library to have consistent integer sizes
2. Create a shell_sort function that doesn't have a return type that takes into its parameters
   (a) a Stats struct pointer to stats
   (b) unsigned integer 32-bit array
   (c) unsigned integer 32-bit variable which contains the length of the array
3. Create a for loop which iterates, my example iterable will be *gap* through every value in the Pratt sequence located in the gaps header file
4. Create a second for loop that iterates, my example iterable will be i starting from *gap* to the length of the array
5. Create an integer variable j and set it equal to i

6. Create another integer variable named *temp* and set it to value of the array at index i
7. Create a while loop that checks if j is greater than or equal to the gap as well as *temp* is less than the array at index j minus gap
8. move the value of the array at index j minus gap into the array at value j
9. reduce j by the gap size
10. Lastly set the value of the array at index j to *temp*

Heap Sort: $\mathcal{O}(n \log n)$
Heap Sort implements the heap data structure. We will be using the max heap version of a heap which essentially states that any parent's node value is greater than or equal to the value of the children. We will represent the heap as an array and any index k will have the index of the left child as $2k$ and the index of the right child as $2k + 1$.

The primary Job of a heap sort is to maintain two jobs:

1. Build a heap
2. Fix a heap

To build this we will require three helper functions as well as the main function *heap_sort* We will require the following:

1. *max_child* This function will take an array, and two other integer values named $first$ and *last*. And what this function will do is compare them and based on whether the right child is larger or the left is larger, will return the larger child.
2. fix_heap This function will take the same arguments as *max_child* and now we will create three variables, set f to false, set an integer, *parent*, and set it to $first$, and set another integer, *great*, equal to the max child. Then you will try to loop until the heap is essentially fixed with the parent being the largest at the top and the structure following to the bottom ultimately turning the f to true and breaking the loop.
3. *build_heap* This function takes the same arguments of the previous two functions and loops in the range of *last* divided by two floored, to the $first$ value minus one. Although the other functions don't specify there is some swapping and comparing which includes -1 to take into account for 0-based indexing.
4. *heap_sort* This function takes an array as a parameter and builds a heap with the first value being 1 and the last value being the length of the array. Then it loops through the leaves starting with the last and ending with the first leaf, iterating backward by one. Lastly, it will swap the values of the array at index first - 1 and the array at the leaf value - 1 after which it builds the heap again until the array is sorted

Essentially this is the goal
[1 8 5 13 0] : Random Array
[13 5 8 0 1] : Heap / Binary Tree
[1 5 8 0 13] : The first iteration takes the whole array and swaps the biggest from smallest

Quick sort: $\mathcal{O}(n \log n)$
Quick sort is a divide-and-conquer algorithm that partitions the unsorted arrays into two smaller arrays based on a pivot. It will sort accordingly and then repeat until the list is ultimately sorted. Steps:

1. Create a partition function, quick sorter function, and quick sort function
2. In the partition function we will take a pivot being the last value in the array, then using the value before the pivot we compare if that value is greater than the value at the first index. Then we continue this process until it is in order and the pivot has moved into the right spot.
3. Next regarding the *quick_sorter* function, it takes the same three arguments as the last function but this function then calls partition if low is less than hi after which it will set an integer p to the partition with the array, the value of lo and hi, and recursively call itself with the array, low, p - 1, and recursively call itself again with the array, p + 1, and hi.
4. Lastly, the *quick$_s$ort* function takes an array and the length of the array as its argument. Then it calls *quick_sorter* with: array, 1, and length as its parameters.

Batcher's Odd-Even Merge Sort $\mathcal{O}(n \log n)$
Because Batcher is so complex, I don't understand it, but what I do know is that it is extremely good for things that are close to hardware and especially CPU chips. The reason is related to parallel computing. Also for this last one, I will just be giving the general pseudo-code of my code translated from C.

1. Create 2 helper functions one will be called comparator() and the other will be called bit_len()
2. comparator() takes: integer Array, integer x, integer y, and swaps values by comparing if the value at x is greater than the value at y in the array. It will be their values if so.
3. bit_len() takes: the length of the array and tracks the number of bits in the integer by right-shifting by one and adds to a counter variable until it reaches zero. Once the value has reached zero, it will return the counter variable which is ultimately the bit length.
4. Lastly batcher_sort() takes: an integer array and the length of the arrary. The function is recursive and the base case is when the length of the array is 0.

5. Create 3 variables, n equals len, t = bit_len of n and p which is t -1 left shifted by 1.

6. While p is greater than 0, create an integer value q is equal to t - 1, left shifted by 1, create an integer r, which is equal to 0, and create integer value d which is equal to p.

7. Nest in the while loop another while loop that checks if d is greater than 0

8. Nest in the nested while loop a for loop which iterates from integer i equals 0, i is less than n - d, and increment i by 1.

9. In the for loop check if i bitwise anded by p is equal to r. if so call comparator() with the arguments: array, i, and i + d.

10. outside the for loop, set d equal to q - p, left q by 1 and set r equal to p.

11. Lastly, in the outermost while loop, right shift p by 1.

4 Credits

1. I'd like to thank Ben, Ben found a mistake in my heap_sort function, helped me with bit-masking, checked my quick_sort and always being active on discord, posting on piazza like his invalid write/read post. There are many small things that Ben does but isn't credited so I am sorry.

2. I'd like to thank Lev, Lev helped me with a part of my Makefile, debugging code, creating the bit_len() function, and overall teaching and being patient as a tutor.

3. I'd like to credit miles for debugging and also for helping me create my set.c file. He helped me conceptualize the bit values and turn them into set functions which ultimately helped me manipulate my testing harness and integer set-array. He also helped with the testing harness and taught me how to use set_insert to make a bit a member in my set.

4. I'd like to credit Darrel Long, because for every function I used his python pseudo code to translate everything. Furthermore, the tips and lessons he teaches us on how to conceptualize the functions helped a lot with this assignment.

5. I'd like to credit Dev, Dev helped me conceptualize the code in a manner that was clear and easy to envision. The sorting algorithms that he assisted me with were heap sort, quick sort,

6. Lastly, I'd like to credit Omar, Omar I went to one of his sections in which he taught me how to apply multiple things to my code. He taught me how to use Calloc to dynamically allocate memory for my array. He taught

me how to seed randomly and generate the array. And he also taught me how to prevent memory leaks by using val grind and remembering to free memory every time I finish using it.

5 Images

Will be provided with explanations and comparisons in WRITEUP.