Sorting Sorting Algorithms: Design Process

Isabella Phung

January 2023

1 Introduction

Sorting is one of the key problems in computer science. Since computers often work with such large swaths of data, it becomes nigh impossible to work with this data without some manner of organization. This project will focus on implementing a variety of sorting algorithms, shell sort, heap sort, quick sort, and batcher sort.

2 Big O Notation

An easy way to compare sorting algorithms is in their speed. We can do so using a measure of duration known as Big O. The above sorting algorithms are generally known to be close to n*log(n) for their Big O measurement. But they may differ in speed due to some constants. As a hypothesis, I predict the slowest will be batcher sort. This algorithm is known to perform well utilizing parallel processes. In this case we'll be performing the algorithm step by step rather than performing it in parallel. Next I predict to be heap sort, followed by shell sort, then quick sort. Quick sort is known to be the fastest comparison sorting algorithm. Heap sort, while efficient I think requires more time to create the heap first in comparison to shell sort.

3 Pseudocode

The pseudocode is heavily based on the python code provided by the CSE13S instructors.
We were also given stats.h and stats.c which have to be fed into each sorting algorithm to keep track of the number of swaps and comparisons that are performed.
Shell Sort:
We'll be utilizing a given gaps.h header file with values provided as the gaps.
these gaps refer to the gap in indexes, not the gap in actual value.
So, for example, suppose you have an array of 5 elements. The header file may go all the way up to 1000, but the sorting algorithm should cut down on the

gaps list until you only have any gap values less than 5.
Suppose you have a list of length 8 with a gap value of 4. You find pairs of 4 index distance apart in the list and swap if needed. You then continue down the list of gaps until you get down to a gap of one.
for gap in gaps #runs through list of gaps in gaps.h file
for i in range(gap, len(arr)) #creates range from gap value to the length of the array, so no gap values greater than the length of the array are used.
j=i #values in between gap and len(arr)
temp = arr[i] #actual value taken from list
while j ¿= gap and temp is less than arr[len-gap] #j¿=gap prevents segmentation fault error from going too far. Then checks value gap distance away. If in correct order, skip, if in wrong order, put the gap-distance-away value in current spot. It continues down until can't go any further

Heap Sort:
Even though heaps are represented as a binary tree, they can be represented as an array. The unsorted array must get put into heap form. Note that for any index k, index of left child is 2k, index of right child is 2k+1. Parent node of any node k is k/2.
Functions:
max_child:
accepts list, first and last elements. Returns bigger child.
fix heap:
calls max child and assigns it to var. Traverses tree, cursor looks at bigger child. If child bigger than cursor, replace node, otherwise, ignore the child build heap: Effectively takes the list and calls it a heap, dumps this heap into fix heap which handles fixing everything such that it follows the rules of a max heap, where parents are larger than their children.
heap sort:
The python code for this algorithm and all the other algorithms, typically start from a 1 index and subtract 1 to get the index for the array. For quick sort, I tweaked the code such that this wouldn't be necessary. For heap sort, I initially tried to do the same thing, but found that it made handling the children nodes more confusing and less intuitive. Children nodes are found via the following algorithm, assuming the index of the parent node is k, the left child index is 2k and the right child is 2k+1. But if one were to assert that the index the heap starts at 0, then the index of k may be 0, which would result in 0 for the left child. So I decided to stick with starting the indices at 1 rather than 0. heap_sort calls build_heap, then calls fix_heap. You should now have a proper heap. Once you have everything in heap form, you know the top value is the biggest value. In order to get a sorted list, drop that element to the bottom and push the smallest value to the top. Why? Because we know that the top element is the largest element, so we know that this is supposed to be at the end of the list. We now have fix heap handle the rest, BUT only input 1 through n-1 elements now that you've put the biggest value at the back. Repeat the same process, swapping the largest values from the heap with the smallest value

at the end of the tree, while sectioning off the sorted big values. The nature of the max heap is that the largest value goes to the top. Unfortunately there's no rules as to exactly where the lower values go in the heap so it's not possible to simply traverse the tree.

Quick Sort:
Worse case: Given a sorted array, does a lot of comparisons to double check.
Partitions array around a pivot, in this case, it will be the last element. Elements less than pivot go to left, greater than pivot go to right. Then left and right sections get fed back into quick sort. This algorithm is typically recursive.
Functions:
partition:
For i in range(lo, hi) #rather than create new arrays, partition will receive indices that indicate the start and end of the sections that need to be sorted
this function returns a partition value after moving all values less than the partition value to the left and values greater it to the right.
quick_sorter:
Feeds partitioned array back into itself for recursive purposes.

Batcher Sort:
Batcher.c only consists of one function, batcher_sort. Sorts in "rounds". For example, with 16 elements, it performs an 8 sort, a 4 sort, 2 sort, then a 1 sort. It's similar with shell sort except the gaps are set as 2 to the k. It's extremely fast if performed in parallel since it generally consists of bitwise operations.

In addition to the sorting algorithms, we were tasked with making set.c which would be used to handle user input. This file defines a set as an integer that encodes a bit vector, which contains all of the options the user can select. This library contains the set functions required for manipulating these sets.
Functions:
set_empty: creates and returns empty set.
set_universal: creates and returns a set with all options set to 1.
set_insert: inserts a 1 at the indicated set index.
set_remove: inserts a 0 at the indicated set index.
set_member: returns true if 1 at indicated set index.
set_union: performs set union on two sets using bitwise OR.
set_intersect: performs set intersection on two sets using bitwise AND.
set_difference: performs a set difference on two sets by getting the complement of one of the sets and ANDing the two sets together.
set_complement: returns set complement.