Wendi Tan
wtan37@ucsc.edu

CSE13s Winter 2021
Assignment 5: Sorting

Pre-lab Part1:

1. 21 rounds.
2. The worst case would be an array in a reverse order, for example {5, 4, 3, 2, 1}. And the comparison would be n*(n-1)/2, where n is the length of the array.
3. Changing the swap condition: if(array[i] < array[i+1]) {swap array[i] and array[i+1]}.

Pre-lab Part2:

1. The best time complexity is O(n*log(n)), and the worst case is O(n*(log n)^2). Since the sequence of gap can determine the round of comparison in shell sort, we need to choose a more efficient way.

Pre-lab Part3:

1. Quick sort is implemented by picking a pivot in the array, and putting smaller numbers on its left, larger numbers on its right, then recursively sort two sub-groups with the same steps. The worst case emerges when the pivot is the largest or the least number in the unsorted array for every round. For example, if we choose the leftmost pivot in a reverse-ordered array (or the rightmost pivot in a sorted array), we need to iterate n times for each sort times n elements need to be sorted, then get the O(n^2). However, we now do this by choosing the middle element in the array and sub-groups as pivots, and this would decrease the chance of encountering the worst case.

Pre-lab Part4:

1. I planned to create a file for public use, containing functions swap, counting moves and comparison. These functions can be called by all source files for sorting methods.

Design

This program can implement four sorting methods: bubble sort, shell sort, quick sort, and heap sort. Since we've been given pseudocodes, this document contains more of structures and my implementation.

Part 1. Sorting algorithms: Bubble sort, Shell sort, Quick sort, and Heap sort.

1). Bubble sort:

This sorting function implemented every round by comparing two adjacent elements from head to the end of the list, in order to shift the largest element to the end of the array among the unsorted range of list. During each round, we check whether the list make changes. If there's no changes, then the array is in a sorted order.

Basic structure:

1. Set a variable "swapped" as "true", which allows it to enter the while loop.
2. Make a while loop (loop 1), the condition should be "swapped==true." Because this loop implemented for each whole swap, which means each two-elements-pair had been compared and swapped. (If swapped==false, then the array is in a sorted order)
3. Make a for loop to compare and swap the adjacent elements, arr[k+1] with arr[k]. If arr[k+1] is smaller, swap two elements. Set "swapped" as true, so that the loop 1 can continue executing.
4. Decrease the unsorted range by one, so that we don't have to compare the rest elements with sorted part.
5. Repeat step 2~4.

2). Sell sort:

The method starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. By starting with far apart elements, it can move some out-of-place elements into position faster than a simple nearest neighbor exchange (cited from Wikipedia: https://en.wikipedia.org/wiki/Shellsort).
First of all, we create a gap, start comparing element arr[gap] with arr[0] and swap if needed. Then compare arr[gap+1] with arr[1], arr[gap+2] with arr[2], … arr[gap+n] with arr[n], arr[n-gap], arr[n-2gap], …, until arr[gap+n] hits the end of the array. Then we decrease the gap and repeat steps above till the gap decreased to zero.

Basic structure:

1. I firstly find the largest available gap (the largest gap can fit in the unsorted array) by using a while loop and set the number to a variable "index."
2. Make a while loop (loop 1) implementing the parry gaps from gaps[index] to the end of the array gaps.
3. Inside loop1, build another loop (loop 2) for each element that will be compared (from arr[gap] to arr[gap + n], where gap+n is array size -1). More specifically, set a variable

pointing to the element which we want other elements(arr[n], arr[n-gap], arr[n-2gap], …) to compare to. And rewrite it after each loop 2 ends.

4. Make a loop (loop 3) inside loop 2, to make arr[gap+n] compare with arr[n], arr[n-gap], arr[n-2gap], ……until n-m*gap equal to zero or cannot be diminished any more.

3). Quick sort:

This sorting method divide the array into two sub-arrays by selecting a pivot point at the midst of the array, putting elements less than pivot point to its left, and elements greater or equal to the pivot are moved to its right. Then implementing the same method in two sub-arrays. We used stack in Quicksort for pushing the lowest and highest indices of the partition to sort.

Basic structure:

a). Stack: We create a stack APT used for pushing and popping elements to/from the stack.

b). Quick sort:

1. Build a function called "partition" for placing elements that less than pivot on its left side and elements that greater or equal to it on its right, set parameters arr[], low and high indices of the partition:

- Set i and j as the partition's leftmost and rightmost array indices, pivot as midst element of the partition.
- Build a while loop for placing elements in the partition. Loop ends when two indices meet.
- If arr[i] is less than pivot, increases i. If arr[j] is greater than pivot, increases j. These two steps are iterative until we found one element on pivot's left which is greater than it, and another one on pivot's right which is smaller than it.
- Swap two elements we found above if i < j.
- Return j.

2. In main function of quick sort:

- Set "left" as 0, "right" as n-1, which are the indices of array's head and tail. Set variables "hi" and "lo" for places to put "partition's" parameters.
- Create a stack and push "left" followed by "right." (stack is to store boundaries of each partition).
- Create a loop as stack is not empty. Pop boundaries to "low" and "hi" respectively. Then call the function "partition" to sort the array in this boundary.
- Then push new boundaries of the partition and repeat steps above.

4). Heapsort:
heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region(cited from Wikipedia: https://en.wikipedia.org/wiki/Heapsort).

Max_child(arr, first, last):

This function is used to get one node's greater child. If the parent is the kth element (1-based indexing) its children are 2k (left) and 2k+1 (right). But the computer starts counting index from 0, so we need to compare two elements arr[2k-1] (left) and arr[2k] and get the greater one. The reason why we set parameters "first" and "last" is that if the remaining unsorted array only has two nodes (one parent and one child), the (2k+1)th element won't exist, and we only need to compare the left child and the parent. So, we have to set the if condition comparing the first and last elements' index.

Fix_heap():

This function is to switch the largest number among three elements of the parent-children heap to the parent node. Suppose parent and children are k, 2k, and 2k+1 respectively. Fix_heap would call max_child() to find the max child and then compare it to the parent. If parent is smaller than the greatest child, swap two numbers. Then set "parent" as the switched child, do the above recursively. So that each parent's children are less than it. Thus, we get a heap.

Build_heap():

This function can be considered as step one. We build the heap three for the array.

Heap_sort():

This is the main sorting function. First of all, we need to get the index of the first and the last element. Then,

1. we call build_heap(), making the max element to the root (arr[0]);
2. setting the max number to the end of the array (swapping the last element and the root element);
3. leaving it fixed by decreasing the sorting range by one;
4. finding the max number with the function fix_heap();
5. and repeat step 2~4 by putting them in a loop.

Finally, we can get the sorted array using heap sort.

Part 2. Global functions for counting comparisons and moves.

I created an APT called “globe” inside a source file called globe.c and its header file. “globe” has members “comp” and “moves” for counting comparisons of array’s elements and the swapping times. This file also has functions “swap()”, and “comparison()” to swap elements and compare elements in an array. Every time I call these functions in other files that include “globe.h”, “comp” will increase by one, and “moves” will increases by 3. “print_array()” is another function to print the array. “glob.c” also contains basic functions like “g_create()” and “delete_g” for creating a globe pointer and delete it preventing memory leak.

In “bubble.c”, “shell.c”, “quick.c”, and “heap.c,” I created one globe pointer separately so that each of them has their “comp” and “moves.”

Part 3. Set.

“set.c” and its header file use bits for tracking which command-line options are specified. I only used “set_empty()” for setting the byte zero at the beginning of the main program; “set_insert()” for inserting bits’ indices: 0, 1, 2, 3, which represent bubble, shell, quick , and heap respectively; “set_member()” for checking whether a certain sorting algorithm is specified in the command-line.

Part 4. Main function in “sorting.c”.

1. Define an enumerated type, setting bits’ indices for bubble, shell, quick , and heap.
2. Set default numbers for seed, size, and printed elements. Use getopt() to fetch options in command-line.
3. Create a dynamic allocated array. And build a function for putting random numbers in that array.
4. Use “set_member()” to identify each sorting algorithm we are going to use, and call them.