CO322: Data Structure and Algorithms

Lab 02 – Sorting

March 03, 2016

• MergeSort

Merge sort is based on Divide and conquer method. It takes the list to be sorted and divide it in half to create two unsorted lists. The two unsorted lists are then sorted and merged to get a sorted list. The two unsorted lists are sorted by continually calling the merge sort algorithm; we eventually get a list of size 1 which is already sorted. The two lists of size 1 are then merged.

Algorithm:

1. Divide the input which we have to sort into two parts in the middle. Call it the left part and right part.
2. Sort each of them separately. Note that here sort use the same function recursively (e.g., `mergeSort()`)
3. Then merge the two sorted parts (e.g., `merge()`).

`mergeSort()` **function**: it takes the array, left-most and right-most index of the array to be sorted as arguments. Check `if (left < right)`; we have to sort only when `left < right` because when `left = right` it is naturally sorted (1 item). Sort the left part by calling `mergeSort()` function again over the left part `mergeSort(array,left,mid)` and the right part by recursive call of `mergeSort()` function as `MergeSort(array,mid + 1, right)`. Lastly merge the two arrays using the `merge()` function.

`merge()` **function**: It takes the array, left-most, middle and right-most index of the array to be merged as arguments. A temporary array (`tempArray[right-left+1]`) is required to store the new sorted part. The current index position (`pos`) of the temporary array is initialized to 0. The left index position (`lpos`) is initialized to left and right index position (`rpos`) is initialized to mid+1, of the array. You have to consider 3 situations when merging the two already sorted arrays:

a) ***Until lpos < mid and rpos < right***:
Compare array[lpos] & array[rpos], and store the lower value at the current index position (pos) of tempArray; increments the position index (pos) the respective left or right index position (lpos or rpos) accordingly.

b) ***Until (lpos <= mid)*** i.e. elements in the left part of the array are left:
Copy rest of the left array (from `lpos` until `mid`) to the `tempArray`.

c) ***Until (rpos <= right)*** i.e. elements in the right part of the array are left:
Copy rest of the right array (from rpos until right) to the tempArray.

Finally copy back the sorted array to the original array.

Exercise 1

1.1. Implement MergeSort() in C to sort an array of integers

1.2. Write a program to demonstrate the functionality of your MergeSort()

1.3. Create an array of 1,000,000 random integers, and measure the time taken to sort this array using your implementation of MergeSort

(hint: in Linux, you can use the built-in command *time* to measure the time taken for your entire program, OR use *clock()* function in <time.h> before the start and after the end of your sorting function to get an accurate CPU time measurement)

Do this 30 times (i.e., running your MergeSort on 30 different integer arrays of 1,000,000 entries) and get the average running time of your MergeSort.

• QuickSort

Quick Sort is divide and conquer algorithm like Merge Sort. Unlike Merge Sort this does not require extra space. i.e., it sorts in place. Here, the dividing step is to choose a pivot and partition the array such that all elements less than or equal to pivot are to the left of it and all the elements which are greater than or equal to the pivot are to the right of it. Then, it recursively sort the left and right parts.

Algorithm:

1. Call the *`partition()`* function to partition the input and return the element in the middle (the pivot) such that, the items to the left of the pivot are not greater than the pivot and items to the right of the pivot are not lesser than the pivot. Thus, the *`partition()`* function should perform the following within the function:
    a. **Select the pivot**: this could be as simple as selecting the first (or last) element in the list, or a lot more complex as selecting the *nінther*(recursive median-of-three).
    b. **Weak sort (partition):** move the elements such that elements lesser than or equal to the pivot are in the left side, and the elements greater than or equal to the pivot are in the right side. Move the pivot element to the middle.
    c. **Return the pivot location**

2. Call *`QuickSort()`* function on the left-side (i.e., from *`low`* to *`pivot-1`*) and the right-side (i.e., *`pivot+1`* to *`high`*) of the pivot

3. When the two recursive *`QuickSort()`* functions return, the larger array is naturally sorted.

The key to the algorithm is the partition procedure. The wrong selection of the partition element (the pivot) could lead to the quadratic [O(n^2)] worst case performance of Quicksort. However, there are many ways to improve the likelihood of equal partitioning of the list to be sorted (e.g., randomized, median-of-three, etc.).

The partitioning (or weak-sort) could also be implemented in many ways. The partitioning algorithm devised by **Hoare** is given below (source: https://en.wikipedia.org/wiki/Quicksort):

```
algorithm partition(A, lo, hi) is
    select pivot element
    swap A[pivot] with A[lo]
    pivot := A[lo]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i >= j then
            return j
        swap A[i] with A[j]
```

A common optimization technique of *QuickSort()* is to use a non-recursive sorting algorithm such as *Insertion Sort* when the number of elements is below some threshold (say 10?). i.e., the *QuickSort()* function will partition input & make the recursive *QuickSort()* calls only if the number of elements are above a given "threshold". Otherwise, it will just call *InsertionSort()* on the input.

Exercise 2

2.1. Implement *QuickSort()* in C to sort an array of integers (you are free to use any method to select your pivot, as well as for partitioning). Make this version fully recursive (i.e., do not use the *InsertionSort()* optimization mentioned above).

2.2. Implement *QuickInSort()* in C to sort an array of integers such that it uses recursive Quick Sort calls until the input array is greater than a given threshold.

2.3. Create an array of 1,000,000 random integers, and measure the time taken to sort this array using your *QuickSort()*. Do this 30 times and get the average running time of your *QuickSort()*.

2.4. Similarly measure the average running time of your *QuickInSort()*.

Bonus marks: Vary the threshold value of your *QuickInSort()* and plot the average running time of your *QuickInSort()* algorithm against the threshold value. Vary your threshold such that you can clearly identify the optimal value or range-of-values for the threshold (e.g., starting from 10 moving up until necessary).

- ## Heap Sort

Heapsort uses the property of Heaps to sort an array. The Heap data structure is an array object that can be viewed as a complete and balanced binary tree. Min (Max)-Heap has a property that for every node other than the root, the value of the node is at least (at most) the value of its parent. Thus, the smallest (largest) element in a heap is stored at the root, and the subtrees rooted at a node contain larger (smaller) values than does the node itself.

Algorithm:

It starts with building a heap by inserting the elements, and letting those rest in its place.

1. Increase the size of the heap as a value is inserted (i.e., increment position of last element)
2. Insert the entered value in the last place.
3. Compare the inserted value with its parent, swap the parent & the child repeatedly, until heap property is satisfied and the inserted value is placed at its right position (If the array starts with 0, the parent of child *i* is given by $\lfloor(i-1)/2\rfloor$)

Once the heap is built, remove the elements from the heap, while maintaining the heap property to obtain the sorted list of entered values.

1. Remove `heap[1]`, as it is the minimum element. Size of the heap is decreased.
2. Now heap[1] has to be filled. We put the last element in its place and see if it fits. If it does not fit (i.e., the value moved to `heap[1]` is larger than at least one of its children), take minimum element among both its children and replaces parent with it.
3. Repeat until the value moved to `heap[1]` finds its rightful place.

Heapsort can be performed in place. The array can be split into two parts, the sorted array and the heap. In fact, building the heap could also be done in place; the heap can grow to insert one element from the non-heap part at a time, until the entire heap is built.

Exercise 3

3.1. Implement a Max-Heap data structure. Your implementation should have the following functions:

- *heapCreate* (Create an empty Heap Data Structure)
- *heapDestroy* (Destroy/clean Heap Data Structure)
- *heapShiftUp* (move a node up in the tree to restore heap condition after insertion)
- *heapShiftDown* (move a node down the tree to restore heap condition after deletion)
- *heapify* (create a heap out of a given array of elements, in place – should be designed using repeated calls to *heapInsert()* function)
- *heapInsert* (add a new element to the heap – should call the *heapShiftUp()* function to restore the heap property)
- *heapRemove* (remove the max value from the max-heap – should call the *heapShiftDown()* function to restore the heap property)

3.2. Implement *HeapSort()* to sort an array of integers, in-place, using the Max-Heap data structure you implemented in 3.1.

3.3. Create an array of 1,000,000 random integers, and measure the time taken to sort this array using your *HeapSort()*. Do this 30 times and get the average running time of your *HeapSort()* implementation.