CS2223: Algorithms
Sorting Algorithms, Heap Sort, Linear-time sort, Median and Order Statistics

1 Sorting

1.1 Problem Statement

You are given a sequence of n numbers $< a_1, a_2, \ldots, a_n >$. You need to output a re-ordering (permutation) of the input sequence as $< a'_1, a'_2, \ldots, a'_n >$, such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

1.2 Insertion Sort

Idea: Just like we sort a set of cards in hand – start with 0 cards in hand, take a card and put it in the right position; now take the next card and insert it in its right position, continue till all the cards are in their correct position.

Algorithm:
insertionSort (A)
    Input: A - the sequence of numbers $< a_1, a_2, \ldots, a_n >$ to be sorted
    Output: no output, but A will be sorted.
    for j = 2..n { // insert $A[j]$ in correct order in $A[1, 2, \ldots, j]$
        $key = A[j]$
        $i = j - 1$
        while $i > 0$ and $A[i] > key$ {
            $A[i + 1] = A[i]$
            $i = i - 1$
        }
        $A[i + 1] = key$
    }

Proof of Correctness: We will use the following loop invariant property – before the start of the $j = k$th for loop the array $A[1, 2, \ldots, k - 1]$ is sorted. We will prove the loop invariant property by induction.

Base case: The loop invariant is true before the start of the first loop, when $k = 2$. Now the array consists of only $A[1]$, which is sorted.

Induction Hypothesis: Let the loop invariant be true for $j = k - 1$. We will show that the loop invariant is true for $j = k$.

Induction Step: As the loop invariant is true for $j = k-1$, before the $k-1$th for loop, $A[1, 2, \ldots, k-2]$ is sorted. In the $k-1$th for loop, we take $A[k-1]$, and insert in in the right position. Therefore before the start of the $j = k$th for loop, $A[1, 2, \ldots, k - 1]$ is sorted.

Overall proof: Therefore before the start of the $j = n$th for loop, $A[1, 2, \ldots, n-1]$ is sorted. During loop $j = n$, $A[n]$ is inserted into the correct position, and therefore at the end of this loop $A[1, 2, \ldots, n]$ is sorted.

Analysis

Best-case: When the given array is already sorted, each time we take a new element, we need to compare only once. This is done $(n-1)$ times and hence the best case complexity is $(n - 1)$ comparisons.

Worst-Case: When the given array is in reverse-sorted order, each time we take a new element, we need to compare it with all the existing elements. The worst-case complexity is therefore given by $1+2+\ldots+(n-1) = (n-1)n/2$.

As we are interested in worst-case complexity, we say that insertion sort complexity is $O(n^2)$.

1.3 Merge Sort

Idea: Use **Divide and Conquer**. Divide the given sequence into two equal subsequences, sort each of them individually (conquer), and merge the two sorted subsequences (combine).

Algorithm:
mergeSort (A, p, r)

Input: A - the sequence of numbers $A[1, 2, \ldots, n]$, p and r, such that we need to sort the elements $A[p, p+1, \ldots, r]$

Output: empty, but the elements $A[p, p+1, \ldots, r]$ are sorted.

if $p < r$ then {
  $q = \lfloor (p + r)/2$
  mergeSort (A, p, q)
  mergeSort $(A, q+1, r)$
  merge (A, p, q, r)
}

The merge procedure takes in an array A and three numbers p, q, r such that $A[p, p+1, \ldots, q]$ and $A[q+1, q+2, \ldots, r$ are sorted, and returns the array A such that $A[p, p+1, \ldots, r]$ is sorted.

merge (A, p, q, r)

Input: A - the sequence of numbers $A[1, 2, \ldots, n]$, p, q and r such that $A[p, p+1, \ldots, q]$ and $A[q+1, q+2, \ldots, r$ are sorted

Output: empty, but $A[p, p+1, \ldots, r]$ is sorted

Create two arrays L and R such that L consists of $A[p, p+1, \ldots, q]$ and R consists of $A[q+1, q+2, \ldots, r$. Add a very large number say ∞ as the last element of L and R

$i = 1$; $j = 1$; // i and j are used to iterate through L and R respectively.
for $k = p, p+1, \ldots, r$ {
  if $L[i] \leq R[j]$ then { $A[k] = L[i]$; $i = i + 1$; }
  else { $A[k] = R[j]$; $j = j + 1$; }
}

Proof of Correctness of merge operation: We need to show that merge takes two sorted sequences and combines them into one sorted sequence.
Note that each step, merge takes the smallest number remaining and puts it in the correct position.

Proof of Correctness of mergeSort: Use induction on the number of elements in the array.

Base step: Check that mergeSort works correctly when the number of elements in the array is $1, 2$.

Induction Hypothesis: Assume that mergeSort works correctly when the number of elements in the array is $\leq (n-1)$.

Induction Step: We need to show that mergeSort works correctly when the number of elements in the array is n. mergeSort splits into two sets, which are both $< n$, and does mergeSort on them (which means the two arrays are sorted). Then merge merges the two lists correctly. Hence mergeSort works correctly when number of elements in the array is n.

Analysis: See that $T(n) = 2T(n/2) + \Theta(n)$. Solving this, we get complexity of mergeSort is $\Theta(nlog\ n)$.

1.4 Quick Sort

Idea: It also uses divide and conquer. Choose an element in the array p as pivot, partition the given array into two sub-arrays – L which has all elements $\leq p$, and R which has all elements $> p$. Sort L and R using quick sort. and the result is L sorted, followed by p, followed by R sorted.

quickSort (A, p, r)

Input: A - the sequence of numbers $A[1, 2, \ldots, n]$, p and r, such that we need to sort the elements $A[p, p+1, \ldots, r]$

Output: empty, but $A[p, p+1, \ldots, r]$ is sorted.

if $p < r$ then { q = partition (A, p, r) // partition returns q where $A[q]$ is the pivot used to parition and $A[q]$ is in the correct position

quickSort $(A, p, q-1)$

quickSort $(A, q+1, r)$

}

partition(A, p, r)

Input: A - the sequence of numbers $A[1, 2, \ldots, n]$, p and r, such that $A[p, p+1, \ldots, r]$ is partitioned by $A[q]$ such that all elements in $A[p, p+1, \ldots, q-1]$ are $\leq A[q]$, and all elements in $A[q+1, q+2, \ldots, r]$ are $> A[q]$.

Output: q such that all elements in $A[p, p+1, \ldots, q-1]$ are $\leq A[q]$, and all elements in $A[q+1, q+2, \ldots, r]$ are $> A[q]$.

$x = A[r]$ // We choose $A[r]$ as the pivot
$i = p$ // i denotes the current index where we will place a number $\leq x$
for $j = p..r-1$ {
  if $A[j] \leq x$ then { swap $A[j] and$A[i];i = i + 1; }
}
swap $A[i]$ and $A[r]$; return i;

Proof of Correctness: We can show by induction that quicksort sorts correctly.

Analysis: $T(n) = T(n_1) + T(n - n_1 - 1) + n$ where n_1 is the size of the first partition.

We can see that the worst case complexity of quicksort is when the pivot always partitions into two arrays such that the size of one of them is $(n-1)$ and the size of the other is 0 (when the pivot is the largest element).

The best case complexity is when the pivot partitions the array equally.

From the above, we say the complexity of quicksort is $O(n^2)$ and $\Omega(nlog\, n)$.

2 HeapSort

Rooted Tree: A rooted tree has a node designated as root. A node can have 0 or more children. Every node except root has exactly 1 parent, root has 0 parents. From the root, you can reach every node in the tree. Depth of a node – depth of root is 0, depth of a non-root node is depth of its parent + 1. Height of the tree is the maximum depth over all nodes.

Binary Tree: A rooted tree where a node can have maximum of 2 children. We distinguish between left child and right child of a node. It is possible for a node to have a right child when it has no left child.

(Binary) Heap: A Binary tree which has no "empty space", and which satisfies the "heap property". All the levels except the leaf level are completely filled, the leaf level is filled from left to a point and the rest are empty.

Heap property: A **max-heap** has the property that the value of a node is $\geq$ the value for its children. A **min-heap** has the property that the value of a node is $\leq$ the value for its children. The maximum element of a max-heap is at its root, the minimum element of a min-heap is at its root.

Heaps are used for priority queues – consider a scheduler that wants to schedule the maximum priority job. The operations that need to be supported are – extractMax to extract the job with the highest priority, insert to insert a new job, heapify to maintain the heap after the highest priority job is removed. All these can be done in $O(logn)$. We will also look at how heaps can be used to sort in $O(nlog\ n)$, and how to build a heap from a given set of numbers in $\Theta(n)$.

A heap of n elements can be stored as an array such that $A[1]$ is the root. $A[2i]$ and $A[2i+1]$ are the children of $A[i]$, and $A[\lfloor i/2 \rfloor]$ is the parent of $A[i]$.

We will look at different heap operations and how they perform one by one. The first one is maxHeapify, which takes in a binary tree, where the left child of the root and the right child of the root are max heaps, and returns a complete max heap. (Note: We actually require the following properties as well. Let h_l denote the height of the left child tree and h_r denote the height of the right child tree. We require that h_l is either $= h_r$ or $= h_r + 1$. If $h_l = h_r$, we require the left binary tree to be complete (every non-leaf node has exactly 2 children). If $h_l = h_r + 1$, we require that the right binary tree to be complete.)

maxHeapify (A, i)

Input: A the input array, and i we want the tree rooted at $A[i]$ to be a heap. In the input, we require that the trees rooted at $A[2i]$ and at $A[2i+1]$ are both heaps.

Output: empty, but the tree rooted at $A[i]$ is a heap.
if $((2 * i \leq n)$ and $A[2 * i] > A[i])$ then $largest = 2 * i$;
else $largest = i$
if $((2 * i + 1 \leq n)$ and $A[2 * i + 1] > A[largest])$ then $largest = 2 * i + 1$;
if $(largest \neq i)$ then { swap $A[i]$ & $A[largest]$; maxHeapify $(A, largest)$; }

Analysis: Complexity is given by $O(log\ n)$ (note that in the best case, the complexity could be a constant – if $A[i]$ was already a heap).

The following algorithm builds a max heap given an array $A[1, 2, \ldots, n]$ of numbers.

buildMaxHeap (A)
  Input: A, the array $A[1, 2, \ldots, n]$
  Output: empty, but A will be a max-heap
  for $i = \lfloor n/2 \rfloor..1$
    maxHeapify (A, i)

Proof of Correctness: Note that the elements $A[\lfloor n/2 \rfloor + 1, \ldots, n]$ are leaves and hence are already max heaps. The above algorithm at any step takes a non-leaf node such that its children are already max-heaps and builds a max heap using maxHeapify. Hence correct.

Analysis: As complexity of maxHeapify is $O(log\ n)$, we can obtain that the complexity of buildMaxHeap is $O(nlog\ n)$. But we can do a better analysis.

Observation: The complexity of maxHeapify given a node is the height of the tree rooted at the node. At lower levels of the tree, there are more nodes and the heights of these trees are smaller.

Consider a node at a "height" of k in the tree. The complexity of maxHeapify for such a node is given by k.

The number of nodes at a height of k is given by $2^{h-k} = 2^h/2^k$.

Therefore the complexity is given by

$$\sum_{k=1}^{h} k{*}2^h/2^k = 2^h.\sum_{k=1}^{h} k/2^k \leq 2^h.\sum_{k=1}^{\infty} k/2^k \leq 2^h.(1/2{+}2/2^2{+}3/2^3{+}4/2^4){+}2^h.\sum_{k=1}^{\infty} 2^{k/2}/2^k$$

As $2^h = \Theta(n)$, the above analysis gives the overall complexity of buildMaxHeap is $\Theta(n)$.

Let us now examine how we can sort using heaps. We will give the intuition only. First build the heap. Now take the maximum element from

the heap and this is the largest element in A. Now take the last element in the heap and put it as the root, and call maxHeapify $(A, 1)$. This will give a new heap with one less element. Now the second highest element in A will be the root of the new heap. Repeat this till we are left with only one element in A.
Analysis: buildHeap has complexity $\Theta(n)$. We call maxHeapify n times, and the complexity of each maxHeapify is $O(log\ n)$. Therefore the overall complexity is $O(nlog\ n)$. Note that in best case heap sort has complexity $\Omega(n)$.

Let us consider how to insert an element into the heap. Again, we will give the intuition only. Insert it into the first free position in A, say $n + 1$. Check if this value is greater than its parent, if yes, swap. Continue this till we reach the root of the heap.

The complexity of insert is given by $O(log\ n)$.

3 Linear Time Sort

Note that the above sort algorithms use comparisons, where they compare 2 numbers to decide which one is smaller than the other. We can actually show that for any comparison sort algorithm, the worst case complexity is $\Omega(nlog\ n)$. However, if we make some assumptions about the input, we can sort in less time.

An example is known as counting sort, where we assume that the input is in the range $[0, k]$ where $k = O(n)$. In this case, we keep a bucket for each value in the range $[0, k]$ (which is $k + 1$ buckets). We scan the input, and each time we increment the counter for the corresponding bucket. After the entire input is scanned, we need to just scan the buckets, see the counts associated with each bucket and return the sorted list. This algorithm takes $\Theta(n)$ time.

4 Median and Order Statistics

If we are given a sorted array of n elements, we can find the ith smallest element (called the ith order statistic) in $\Theta(1)$ steps. You can also search whether an element with value k is present in $O(log\ n)$ using binary search.

The algorithm is given below.
binarySearch (A, p, q, k)

Input: $A[p, p+1, \ldots, q]$ the sorted array, k is the number to find if k exists in $A[p, p+1, \ldots, q]$

Output: True if k is present, otherwise false.
if $p = q$ then {
if $A[p] = k$ then return true; else return false; }
if $A[\lfloor (p+q)/2 \rfloor] = k$ then return true;
if $A[\lfloor (p+q)/2 \rfloor] > k$ then return binarySearch $(A, p, \lfloor (p+q)/2 \rfloor - 1, k)$;
else return binarySearch $(A, \lfloor (p+q)/2 \rfloor + 1, k)$;

The worst case time complexity of the above algorithm is given by the recurrence $T(n) = T(n/2) + \Theta(1)$, whose solution is $T(n) = O(log\ n)$.

We now define the **selection problem** as given an array A of n distinct numbers (need not be sorted) and a number i with $1 \leq i \leq n$, find the ith smallest element in A.

Note that we know how to find the minimum and maximum in $\Theta(n)$ time. How do we solve it for any i? We can do it in linear time, the intuition is as follows. Find a number x as the pivot, and partition A into two sets based on x. Now we need to look in one of the two sets. This algorithm takes worst case when the pivot is not a good one – the pivot is say the largest element.

To ensure that we can do selection in linear time, we can choose a good pivot. Let us see how to choose a good pivot. For simplicity we assume that n is a power of 5 !!

1. divide A into $n/5$ groups, each having 5 elements.

2. find the median of each of the $n/5$ groups.

3. find the median of these $n/5$ medians recursively using the selection algorithm. Use this median of medians as the pivot.

4. partition A according to the pivot, and now recursively do the selection problem on the correct partition.

Let us examine why the above pivot is a good one. First observe that $(n/5 - 1)/2$ medians will be $>$ the median of medians. Similarly $(n/5 - 1)/2$ medians will be $<$ the median of medians.

This means at least $(3n/10 - 3/2)$ elements in A are $>$ the median of medians. Similarly at least $(3n/10 - 3/2)$ elements in A are $<$ the median of medians. This means when we call the selection again, the size of the partition will be atmost $7n/10 + 1/2$.

We can therefore write the recurrence for the worst case complexity as $T(n) = T(n/5) + T(7n/10 + 1/2) + \Theta(n)$. Solving this, we get the time complexity of the above algorithm is $\Theta(n)$.