ESO211 (Data Structures and Algorithms): Lecture Notes Set 5

Shashank K Mehta

1 Comparison Based Sorting Algorithms

For each sorting algorithm we assume that the input is an array $A[1 : n]$ and the key associated with each element $A[i]$ is $A[i].key$. It must be kept in mind that if $i \neq j$, then $A[i] \neq A[j]$ but it is possible that $A[i].key = A[j].key$. We sort in non-decreasing order in each case.

In comparison based sorting algorithms, given any two objects $A[i]$ and $A[j]$, we only ask "is $A[i].key \leq A[j].key$?" or $A[i].key < A[j].key$?". We do not assume any thing about the kind of keys are involved. It is possible the objects are humans and $A[i].key < A[j].key$ means $A[j]$ has greater faith in God than $A[i]$ does.

1.1 Insertion Sort

```
for $i := 2$ to $n$ do
    $j := i - 1$;
    $temp := A[i]$;
    while $j \geq 1$ & $A[j].key > temp.key$ do
        $A[j + 1] := A[j]$;
        $j := j - 1$;
    end
    $A[j + 1] := temp$;
end
```

Algorithm 1: Insertion Sort

1.2 Invariant to Prove Correctness of a Loop

To prove that a loop structure in an algorithm is indeed doing its job, a standard technique is to design an assertion $S(k)$. An assertion is a Boolean function of some parameters and evaluates to *true* or $false$. $S(k)$ must be a Boolean function of the parameters of the loop. The invariant $S(k)$ should be evaluated using the values of the loop parameters at the end of the k-th pass of the loop.

The invariant should be so designed that $S(0)$ must be true. Further, if the l-th pass is the last pass, then the truth of $S(l)$ must imply the correctness of the loop structure.

Let us take the For-loop in the Insertion Sort algorithm as an example. The For-loop in this case is the entire algorithm. The goal the loop is to sort the contents of $A[.]$ which were in arbitrary order initially. Let us denote the value stored in $A[j]$ after k passes be denoted by a_j^k. Consider the following assertion as $S(k)$:

$$\{a_1^0, \ldots, a_{k+1}^0\} = \{a_1^k, \ldots, a_{k+1}^k\} \text{ AND } a_1^k \le a_2^k \le \cdots \le a_{k+1}^k$$

Observe that $S(0)$ is trivially true. The loop will end after $n-1$ passes. If $S(n-1)$ is true, then it implies that $a_1^{n-1} \le a_2^{n-1} \le \cdots \le a_n^{n-1}$, which is precisely the goal of the loop. Now to prove the correctness of the loop we only need to prove that $S(k-1) \Rightarrow S(k)$ for any $1 \le k \le n-1$. Using this claim, the fact that $S(0) = true$, and the induction principle we can conclude that $S(n-1) = true$.

Exercise: $S(k-1) \Rightarrow S(k)$ for any $1 \le k \le n-1$.

1.2.1 Analysis

In the worst case after iteration i, $i-1$ comparisons and assignment operations will be done. So the worst case time complexity of the algorithm is $c.\sum_{i=2}^{n} i-1 = c.\sum_{i=1}^{n-1} i = c.n(n-1)/2 = O(n^2)$.

It requires only $O(1)$ space in addition to the input array.

1.3 Bubble Sort

```
for $i := 1$ to $n-1$ do
    for $j := n$ down to $i+1$ do
        if $A[j-1].key > A[j].key$ then
            $Swap(A[j-1], A[j])$;
        end
    end
end
```

Algorithm 2: Bubble Sort

Invariant of the outer for-loop is that at the end of iteration $i = k$ all elements in $A[1:k]$ are at their final position. The inner loop brings the smallest element in $A[i+1:n]$ to $A[i+1]$. The worst case time complexity is $O(n^2)$. It also requires only $O(1)$ space in addition to the input array. This is also a stable sort.

1.4 Merge Sort

We assume here that n is a power of 2. In case it is not, then add sufficient number of "infinities" so the total number of elements become a power of 2, say $n = 2^k$. For convenience we assume that input is $A[0:n-1]$. Here we present the algorithm in iterative form.

```
for $r := 0$ to $k-1$ do
    $l := 2^r$;
    for $j := 0$ to $n-2l+1$ insteps of $2l$ do
        $Merge(A[j:j+l-1], A[(j+l:j+2l-1])$;
    end
end
```

Algorithm 3: Merge Sort

The invariant for the outer loop at the start of iteration $r = \log_2 l$ is that each sequence $A[p.l : (p+1).l-1]$ is sorted for $p = 0, \ldots, (n/l)-1$. Each merge operation performs $O(l)$ comparisons. Hence the inner loop makes $O(n)$ comparisons and total cost is $O(n \log n)$ comparisons. The additional space required is $O(n)$. The merge process is described below.

```
p := a;
q := a + l;
r := a;
while r < a + 2l do
    if p = a + l then
        B[r] := A[q]; q := q + 1; r := r + 1;
    end
    else
        if q = a + 2l then
            B[r] := A[p]; p := p + 1; r := r + 1;
        end
        else
            if A[p].key ≤ A[q].key then
                B[r] := A[p]; p := p + 1; r := r + 1;
            end
            else
                B[r] := A[q]; q := q + 1; r := r + 1;
            end
        end
    end
end
for i := a to a + 2l − 1 do
    A[i] := B[i];
end
```

Algorithm 4: $Merge(A[a : a + l - 1], A[a + l : a + 2l - 1])$

1.5 Heap Sort

```
Create a Min-Heap H;
BuildHeap(A, n);
for i := 0 to n − 1 do
    B[i] := MinDelete(H);
end
for i = 0 to n − 1 do
    A[i] := B[i];
end
```

Algorithm 5: Heap Sort

BuildHeap takes $O(n)$ time and each *MinDelete* takes $O(\log_2 HeapSize)$. So total time is $T(n) \leq c_1.n + c_2 \sum_{j:=n}^{1} \log_2\ j \leq c_3 \log_2\ (n!) = O(n.\log\ n)$. The additional space for Heap etc is $O(n)$.

1.6 Quick Sort

```
Data: Array $A[1:n]$ of totally ordered elements with multiplicity and $1 \le i \le j \le n$
Result: If $i \le j$, then Permute the elements in $A[i:j]$ so that they are sorted in
        non-decreasing order
if $j = i$ then
    return;
end
Randomly select an element, $pivot$, from $A[i:j]$;
$k := Partition(A, i, j, pivot)$;
/* function $Partition$ puts all elements in $A[i:j]$ which are less than or equal
   to $pivot$ at locations $A[i:k]$ and remaining integers in $A[k+1:j]$ for a
   suitable $k$. It returns $k$, the position where $pivot$ is placed after
   partitioning                                                                  */
$QuickSort(A, i, k-1)$;
$QuickSort(A, k+1, j)$;
```

Algorithm 6: $QuickSort(A, i, j)$

1.6.1 Correctness of Algorithm

Exercise.

1.6.2 Complexity Analysis

Worst Case Analysis

Let $T(k)$ denote the time to sort a sequence of size k. Then $T(k) \le c_1.k + T(k_1) + T(k_2)$ where $k_1 + k_2 = k - 1$. Since $T()$ is at least linear, $T(k_1) + T(k_2) \le T(0) + T(k-1) = T(k-1)$. So $T(n) \le c_1.n + T(n-1) \le c.n + c.(n-1) \cdots + c.1 + T(0) = c.n.(n+1)/2 = O(n^2)$.

Average Case Analysis

Some times the worst cases are very few and remaining cases cost much less than what worst cases do. If the input is unbiased, then average case analysis gives much better picture of the performance.

The crux of this algorithm is selecting a pivot p from the given unsorted set S and split it into three subsets (1) set of elements of S which have key are less than or equal to $p.key$, (2) $\{p\}$, (3) set of elements with key greater than $p.key$. This way the set gradually gets refined. At each stage we have a sequence of subsets.

For example suppose initial set is $\{38, 17, 5, 16, 88, 2, 49, 22, 72, 13, 7, 4, 31\}$. Suppose 17 is chosen as the pivot. The we reduce the set to a sequence of sets $\{5, 16, 2, 13, 7, 4\}, \{17\}, \{38, 88, 49, 72\}$. Next suppose 5 was selected as pivot in the first set. The the set sequence becomes $\{2, 4\}, \{5\}, \{16, 13, 7\}$, $\{17\}, \{38, 88, 49, 72\}$. The entire process is to refine this sequence so that each set becomes a singleton. Then the set sequence will become the sorted sequence.

The cost of the algorithm is precisely the cost of all the partitioning. The total cost all the partitioning is a constant times the total number of comparisons. To measure this, let us define a variable $X_{i,j}$ which take value zero if i-th and the j-th elements are never compared but it takes value 1 if they are compared. So the time of the algorithm is $T(n) = \sum_{i<j} X_{ij}$. SO the problem reduces to finding X_{ij} for all i, j.

Observe that there is some randomness inherent in the algorithm. The way we select a pivot is random so X_{ij} may differ in different runs of the algorithm and in different input set of n elements.

So we assume that X_{ij} is a random variable and we will only try to compute the expected value of $T(n)$. So $E[T(n)] = E[\sum_{i<j} X_{ij}] = \sum_{i<j} E[X_{ij}]$. Let $Pr(i,j)$ denote the probability of i-th and j-th elements getting compared in the process. Then $E[T(n)] = \sum_{i<j}(1.Pr(i,j) + 0.(1 - Pr(i,j)) = \sum_{i<j} Pr(i,j)$. So the question reduces to "How to find $Pr(i,j)$ for any pair of i,j?"

Observe that the algorithm does not depend on the absolute values of the keys. In stead it only depends on the relative values (which one is bigger than which). This is the characteristic of all the algorithms discussed so far. So it is sufficient to label each element uniquely by the number at which it will occur in the final (sorted) stage. In the above example $a_1 = 2, a_2 = 4, a_3 = 5, a_4 = 7, a_5 = 13, a_6 = 16, a_7 = 17, a_8 = 38, a_9 = 49, a_{10} = 72, a_{11} = 88$.

Claim 1 *Any two elements a_i and a_j will remain together in a set if and only if all the consecutive elements $a_i, a_{i+1}, a_{i+2} \ldots, a_{j-1}, a_j$ remain in a single set. Further, until a pivot a_k is selected such that $i \leq k \leq j$, a_i and a_j will remain together and subsequently they will be separated.*

Suppose in the set sequence $S_1, S_2, \ldots, S_r$, all the elements $a_i, a_{i+1}, a_{i+2} \ldots, a_{j-1}, a_j$ belong to a set S_t. Suppose S_t is the smallest set in which these elements are together. Note there may be elements in S_t other than these. Clearly, from the above Claim the pivot selected for S_t must be some a_k such that $i \leq k \leq j$.

Claim 2 *If the smallest set which contains all of the elements $a_i, a_{i+1}, a_{i+2} \ldots, a_{j-1}, a_j$ is S_t, then the pivot selected for S_t must be in the range $[a_i, \ldots, a_j]$.*

Nest claim says that if S_t is the last set containing $a_i, a_{i+1}, \ldots, a_j$, then a_i and a_j may be compared only during the partitioning of S_t.

Claim 3 *If S_t is smallest set that contains a_i and a_j, then the pivot for S_t will be a_k with $i \leq k \leq j$. Further, a_i and a_j may get compared only during the partitioning of S_t. Finally, a_i and a_j will be compared if and only if $a_k = a_i$ or $a_k = a_j$.*

Claim 4 $Pr(i,j) = 2/(j - i + 1)$.

Proof $Pr(i,j)$ is equal to the probability that the pivot for S_t is a_i or a_j. Suppose that the pivot for S_t is smaller than a_i or greater than a_j. Then S_t cannot be th smallest set containing a_i and a_j. But that is not the case. So a_k must have been in the range $a_i : a_j$. So probability that the pivot is a_i subject to the condition that pivot was selected in the range $a_i : a_j$ is $1/(j - i + 1)$ under uniform probability assumption. Similarly a_j will be the pivot in S_t is also $1/(j - i + 1)$. So $Pr(i,j) = 2/(j - i + 1)$. ■

Plugging this value in the expression for $E[T(n)]$ we get the expected time equal to

$$
\begin{aligned}
&c.\sum_{i<j} 2/(j-i+1))\\
=&2c.\sum_{i=1}^{n-1}\sum_{r=1}^{n-i} 1/(r+1)\\
=&2c.\sum_{i=1}^{n-1}(H_{n-i+1}-1)\\
\leq&2c.\sum_{i=1}^{n-1}(1+\log\ (n-i+1)-1)\\
=&2c.\sum_{i=1}^{n-1}\log(i+1)\\
=&2c.\log\ n!\\
=&O(n.\log\ n).
\end{aligned}
$$

1.7 Stable Sorting

Definition 1 *A sorting algorithm is said to be* stable *if in the input sequence, $a_1, a_2, \ldots, a_n$, elements a_i and a_j have the same key and $i < j$, then a_i precedes a_j in the output sequence as well.*

Example: Suppose all students of a class are standing in a queue. The queue is sorted by the first name using a stable sorting algorithm. Suppose Abhay Kumar was initially standing at position 10 and Abhay Jain at position 20. Then after sorting Abhay Kumar will still stand ahead of Abhay Jain. Thus we must do stable sorting.

Example: Thousands of people are waiting in a queue for buying grocery at a shop and the shopkeeper decides that the older people will be served first. So the queue must be sorted in decreasing order of age. But if two persons, A and B, of same age are in the queue with A ahead of B, then we want to ensure that A remains ahead of B after sorting. This requires stable sorting.

Exercise: Analyze each of the preceding algorithm and find which are stable.

2 Sorting Algorithms Which Do Not Use Comparison

Unlike comparison based algorithms here we will assume about the nature of the keys.

2.1 Counting Sort

Suppose $A[0..n-1]$ contains the objects to be sorted, and their keys are integers in the range $0..k$. The objective is to stably sort the array with a method which is efficient when k is small. Note that each object is distinguishable but their keys are from the integer set $\{0, 1, \ldots, k\}$.

First consider the following situation. Suppose we are given an array $C[0..k]$ of integers such that $C[j]$ is the number of objects in A with key less than or equal to j. Then the right most element in A with key value j, must go to position $C[j]$ in the output array, the second right most element with key value j must go to position $C[j]-1$, and so on. Observe that this results is a stable sorting.

The algorithm is as follows.

Time complexity $= O(n+k)$.

```
for $j := 0$ to $k$ do
    $C[j] := 0$;
end
for $i := 0$ to $n-1$ do
    $C[key(A[i])] := C[key(A[i])] + 1$;
end
for $j := 1$ to $k$ do
    $C[j] := C[j-1] + C[j]$;
end
for $j := n-1$ to $0$ do
    $B[C[key(A[j])]] := A[j]$; $C[key(A[j])] := C[key(A[j])] - 1$;
end
```
Algorithm 7: Counting-Sort

Note: This algorithm is linear in n if $k = O(n)$. It can be generalized to sorting a sequence of n totally ordered objects with k distinct values if we can map them to numbers $1 : k$ respecting their ordering.

7. **Radix Sort**

Problem: To sort a sequence of n base-b integers, each of which have d digits.

Given two integers $A = a_d a_{d-1} \ldots a_1$ and $B = b_d b_{d-1} \ldots b_1$, $B < A$ if and only if there exists r such that $a_d = b_d, a_{d-1} = b_{d-1}, \ldots, a_r = b_r$ and $b_{r-1} < a_{r-1}$. This fact leads to the following sorting algorithm.

```
for $j := 1$ to $d$ do
    Stably sort the integers using $j$-th least significant digit as the key.
end
```

If we use counting sort in each iteration, then each iteration will cost $O(n+b)$. So the time complexity of this algorithm is $O(d.(n+b))$.

Show that the above algorithm correctly and stably sorts the integers.

8. **Bucket Sort**

Problem: Sort $A[0..n-1]$ having numbers in the open interval $[0, 1)$.

```
Define $B[0..n-1]$ to be an array of pointers pointing to lists of numbers;
for $j := 0$ to $n-1$ do
    $B[j] := nil$;
end
for $j := 0$ to $n-1$ do
    Insert a node containing $A[j]$ into head of the list of $B[\lfloor n.A[j] \rfloor]$;
end
for $j := 0$ to $n-1$ do
    Insertion-sort(list pointed by $B[j]$);
end
Concatenate the sorted lists of $B[0], B[1], ..., B[n-1]$ in that order.
```
Algorithm 8: Bucket Sort

Note that if the numbers were uniformly spread in the interval $[0, 1)$, the average number of elements per bucket must be just one. Hence we do not care to efficiently sort each bucket.

Let there be n_i nodes in the linked list of $B[i]$. Then the insertion-sort for $B[i]$ will cost $O(n_i^2)$. Thus the total time complexity is $O(n + \sum_i(n_i^2))$.

In the worst case one bucket may have all the numbers and others are empty. Then $n_i = n$ for some i and $n_j = 0$ for each $j \neq i$. Then the time complexity will be $O(n^2)$.

Let us now determine the average time complexity under the assumption that all the input numbers are uniformly distributed in the range $[0, 1)$.

If the input numbers are uniformly distributed in the range $[0, 1)$, i.e., a number goes to i-th bucket with probability $p = 1/n$, then $Prob(n_i = k) = \binom{n}{k}p^k(1-p)^{n-k}$. Then $E[n_i^2] = \sum_{k=0}^{n} Prob(n_i = k).k^2 = \sum_{k=0}^{n} k^2.\binom{n}{k}.(1/n)^k(1-1/n)^{n-k} = \sum_{k=1}^{n} k^2.\binom{n}{k}.(1/n)^k(1-1/n)^{n-k}$.

By expressing $\binom{n}{k} = n!/(k!.(n-k)!)$ and simplifying the expression we get $E[n_i^2] = 1 + \sum_{k=1}^{n-1}\binom{n-1}{k}p^k(1-p)^{n-1-k}.k$. Repeating similar step we get $E[n_i^2] = 1 + (n-1).p = 2 - 1/n$.

So the expected time is $O(n + E[\sum_i(n_i^2)]) = O(n + \sum_i E[n_i^2]) = O(n + \sum_i(2 - 1/n)) = O(3n) = O(n)$.

Theorem 1 *The expected time complexity of Bucket Sort under uniform distribution of the numbers in the range* $[0, 1)$ *is* $O(n)$.

Note: This algorithm is applicable to any set of reals if we normalize the key, i.e. divide the key of each A[i] by $K + 1$ where K is the largest key value..

3 Lower Bound for Sorting

In this section we will determine a lowerbound for algorithms for sorting which are based on comparisons.

3.1 Decision Tree

This is a general method to lay down all possibilities while trying to take a decision. We express then in form a tree. The root-node of the tree is the starting point which is unbiased/unconditioned. At this stage we enumerate all possibilities and express each as a child node of the root. Assume that the possibility 1 has been realized, then enumerate all the subsequent possibilities and make them the child nodes of the first child of the root. This way we complete the tree.

Let us suppose we have an smart algorithm which sorts a set of n numbers $a_1, \ldots, a_n$ given as input. In various stages it picks two numbers and asks which is smaller. By "smart" we mean that it does not ask irrelevant questions. Suppose it begins with a_1 and a_2 and asks which is smaller. Then we will make three child nodes. The first is associated with $a_1 < a_2$, second with $a_1 = a_2$ and the last with $a_2 < a_1$. Under the first node it may choose some two numbers a_i and a_j and again ask the question and make three child nodes of the first child of the root. So the node will correspond to $a_1 < a_2$ and $a_i < a_j$. eventually there will be a stage when a node will emerge when all comparisons are resolved and input numbers can be sorted. Figure **??** shows a possible decision tree for some set of 4 numbers $\{a_1, a_2, a_3, a_4\}$.

In the following theorem we will show that in a decision tree if each node has at most b child nodes and there are at most N final possibilities, then the path from root to a leaf node has length at least $\log_b N$.

Theorem 2 *If in a decision tree each question has at most b possible answers and there are at most N final possibilities (i.e., there are N distinct leaf nodes) asked has at most b possible answers. Then in the worst case there will be a path with length at least $\lceil log_b N \rceil$.*

Proof If the length of each path from the root to the leaf-nodes is at most k, then the most number of leaf nodes can be accommodated in the tree if each node, other than the leaf-nodes, has b children. In this case the tree will be complete b-nary tree. In this case there will be b^k leaf nodes. Since they must accommodate N distinct nodes, $b^k \geq N$. So $k \geq \log_b\ N$. ■

Corollary 1 *Any algorithm for sorting based on only comparisons must have time complexity $\Omega(n. \log_3\ n)$ where n is the number of elements to be sorted.*

Proof In a decision tree reflecting the decisions taken by the algorithm in identifying the correct order (in which the numbers are sorted) each comparison question can have only 3 possible answers. So $b = 3$. The leaf nodes in this case must have resolved all comparisons so a unique order must have been established among the input numbers. Since there are $n!$ possible permutations of n numbers, there must be at least $N = n!$ leaf nodes. It is possible that there may be more than $n!$ nodes if same order appears in more than one leaf node. Thus the longest path in the tree must have length at least $\log_3\ n!$.

Sterling's approximation for $n!$ is $\sqrt{2\pi} n^{n+0.5}.e^{-n}$. Hence we have the following result. So the longest path has length is at least $c.(n. \log_3 n - n)$. So the algorithm has worst case time complexity $\Omega(n. \log_3\ n)$. Since this can be any algorithm, the problem has the lower bound for the worst case time complexity $\Omega(n/ \log_3\ n)$. ■