ESO211 (Data Structures and Algorithms) Lectures 14 to 16

Shashank K Mehta

1 General Sorting Algorithms: Algorithms that use comparison as the base step

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 $A[i].key$. We sort in non-decreasing order in each case.

1. **Insertion Sort**

```
for i := 2 to n do
    j := i − 1;
    temp := A[i];
    while j ≥ 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

Invariant of the for-loop is that at the start of iteration $i = k$, $A[1:k-1]$ is sorted. Worst case complexity is $O(n^2)$. It requires only $O(1)$ space in addition to the input array. It is a stable sort, i.e., if $A[i] = x$ and $A[i+j] = y$ at the start and if $x.key = y.key$, then after sorting x remains to the left (lower-side) of y.

2. **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.

3. **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 do
        Merge(A[2.l.j : 2.l.j + l − 1], A[(2.l.j + l : 2l.j + 2.l − 1]);
    end
end
```

Algorithm 3: Merge Sort

The invariant for the outer loop at the start of iteration $l = k$ is that each sequence $A[l.j : l.j+l-1]$ is sorted for $j = 0, \ldots, (n/l) - 1$. 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: $Merge(A[a : a + l - 1], A[a + l : a + 2l - 1])$

4. **Heap Sort**

Since insert and MinDelete take at most $\log n$ comparisons, this algorithm takes $O(n \log n)$ time. The additional space for Heap etc is $O(n)$.

5. **Quick Sort**

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

Algorithm 5: Heap Sort

```
Data: Array A[1 : n] of integers with multiplicity and 1 ≤ i, j ≤ n
Result: If i ≤ j, then Permute the numbers in A[i : j] so that they are sorted in increasing
        order
if j ≤ i then
 |  return;
end
Randomly select a number, pivot, from A[i : j];
k := Partition(A, i, j, pivot);
/* function Partition puts all numbers 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: $QuickSort(A, i, j)$

The worst case time complexity is $O(n^2)$, but the average case complexity is $O(n \log n)$.

Average Case Analysis

Let a_i denote the i-th value in the sorted sequence for all i. Observe that in the entire execution of this algorithm any pair of a_i and a_j will be compared if and only if the first element which was selected a as the pivot from the range $a_i, a_{i+1}, \ldots, a_j$ is either a_i or a_j. Let X_{ij} be a random variable which takes value 1 when a_j and a_j are compared, else it takes value 0. So the total cost of the algorithm is $O(\sum_{i<j} X_{ij})$. So expected cost is $(E[\sum_{i<j} X_{ij}]) = O(\sum_{i<j} E[X_{ij}])$. From the above observation $E[X_{ij}]$ is the probability that a_i and a_j are compared. So $E[X_{ij}]$ is $Pr(next\ pivot\ is\ a_i | First\ time\ a\ pivot\ will\ be\ selected\ from\ a_i : a_j) + Pr(next\ pivot\ is\ a_j | First\ time\ a\ pivot\ will\ be\ selected\ from\ a_i : a_j)$. Since all elements can be selected as pivot with equal probability, each term on the right hand side is $1/(j-i+1)$. So plugging this value we get the average time equal to $O(\sum_{i<j} 2/(j-i+1)) = O(2\sum_{i=1}^{n}\sum_{r=1}^{n} 1/r) = O(2.n.H_n) = O(n.\log n)$, where H_n denotes the Harmonic series $1/1 + 1/2 + \cdots + 1/n$ which is between $\log_e n$ and $1 + \log_e n$. So the average cost is $O(n.\log n)$.

2 Sorting Algorithms Which Do Not Use Comparison

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.

6. **Counting Sort**

Suppose $A[0..n-1]$ contains objects and their keys are integers in the range $0..k$. The objective is to stably sort the array with a method which is efficient if k is small.

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.

```
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

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

Note: This algorithm is linear in n if $k = o(n)$ (small-oh of n means $ck \leq n$ for some constant c.)

7. **Radix Sort**

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

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

Time complexity = $O(d.(n + b))$.

Observe that given two integers $A = a_d a_{d-1} \dots a_1$ and $B = b_d b_{d-1} \dots b_1$, if $a_d = b_d, a_{d-1} = b_{d-1}, \dots, a_r = b_r$ and $b_{r-1} < a_{r-1}$, then $B < A$. Show that the above algorithm correctly and stably sorts the integers.

Note: Again, the complexity is dependent on the values of the input integers.

8. **Bucket Sort**

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

```
Declare B[0..n − 1] be an array of pointers pointing to list 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[⌊n.A[j]⌋];
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 be empty. Then $n_i = n$ for some i and $n_j = 0$ for each $j \neq i$. Then the time complexity will be $O(n^2)$.

Exercise: Given that $E[n_i] = 1/n$ show that $E[n_i^2] = 2 - 1/n$.

So the expected time is $O(n + 2n) = O(n)$.

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 the largest key plus 1.

Note: This algorithm will perform well when the integers are uniformly distributed in the value range.

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 search an element from a set which satisfies a criterion. We ask a question which has at most b possible answers. For example, suppose the set is all the students in the campus and the question is: what is the first alphabet (English) of your first name. There are 26 possible answers and the entire set of students partitions into 26 subsets. Based on the answer we select that subset for which this answer holds true. One question allows us to reduce the search space. By repeated questions we can zero-in onto the desired element. To implement this method we also need a suitable data-structure which allows us to store all the information received from the answers efficiently.

The binary search is a simple example of this method. We ask whether the current element's value is greater or lesser than the element being searched. From the answer of each query we reduce the search space. Another example of this method is the game of "Twenty Questions".

Next we will show that if the each question has at most b possible answers, then no matter what strategy we adopt in selecting questions, the number of questions required to be asked in the worst case will be $\lceil log_b N \rceil$ where N denotes the number of elements in the initial set.

Theorem 1 *Suppose in a decision-tree based search, each question asked has at most b possible answers. Then in the worst case there will be at least $\lceil log_b N \rceil$ question-answer rounds, where N denotes the number of elements in the initial set.*

Proof To prove this claim we consider the following game played between two adversaries: X and Y. Here X will try to guess the element of the set which Y has chosen from the set. Y plays a trick. She does not select any element at the start of the game. When X asks the question with possible answers $A_1, A_2, \ldots, A_b$, associated with each answer is a subset which conforms to that answer. Clearly there will be at least one set having at least N/b elements in it. Suppose one such answer is A_j. Then Y replies A_j. Y repeats this trick at each question. After r rounds the reduced set size cannot be smaller that N/b^r. Hence the set size will reduce to one after at least $\lceil log_b N \rceil$ questions. After the set reduces to a singleton, Y accepts that element as the one she had selected.

Y can always play this trick no matter what strategy is used by X in asking questions. Hence we have shown that in all cases the worst case number of questions will be at least $\lceil log_b N \rceil$. ■

3.2 Comparison Based Sorting

In sorting problem we are given a sequence of numbers $a_1, \ldots, a_n$. We are supposed to find that permutation $a_{j_1}, \ldots, a_{j_n}$ such that $a_{j_i} \leq a_{j_{i+1}}$ for all i. A comparison, a_p with a_q, can result into one of the three answers: $a_p < a_q$; $a_p = a_q$; and $a_p > a_q$. At any point in time that set is being considered which satisfies all the answers given earlier. From the above theorem irrespective of the strategy used in selecting the pair for comparison, there always exists a permutation which can be identified only after at least $\lceil log_3 n! \rceil$ comparisons because there are $n!$ possible elements in the set at the start. Sterling's approximation for $n!$ is $\sqrt{2\pi} n^{n+0.5}.e^{-n}$. Hence we have the following result.

Theorem 2 *The worst case performance of any comparison based sorting algorithm requires at least $c_1 + (n+0.5)\log_3 n - c_2 n$ comparisons for some instance. Equivalently its worst case time complexity is $\Omega(n \log n)$.*