Indian Institute of Information Technology Design and Manufacturing, Kancheepuram, Chennai 600 127, India
An Autonomous Institute under MHRD, Govt of India
http://www.iiitdm.ac.in
COM 209T Design and Analysis of Algorithms -Lecture Notes

Instructor
N.Sadagopan
Scribe:
P.Renjith

Average Case Analysis

Algorithm analysis are broadly classified into three types such as

- Best case analysis : Gives a lower bound on the run-time analysis over all inputs.
- Worst case analysis : This analysis gives an upper bound on the run-time analysis over all inputs.
- Average case analysis : This analysis gives the average bound on the run-time analysis over all inputs. Also called as probabilistic analysis.

Let D be the set of inputs for the problem under consideration and $I \in D$, an instance of the problem. Let $t(I)$ be the number of basic operations performed by the algorithm on input I. It is important to note that there are infinitely many inputs for any given problem. Suppose, we assume that the input comes from the set of positive integers, then any finite subset of $\mathbb{I}^+$ is a valid input. The number of possible finite subsets of $\mathbb{I}^+$ is countably infinite. That is, any finite subset can be expressed as a vector $(a_1, \dots, a_k)$, for some fixed integer k, and the number of such subsets has an enumeration and has one-one mapping with the set of natural number. Since, the number of inputs is countably infinite, computing simple average (the ratio of the sum over all inputs to the number of inputs) is not feasible one. This calls for a probabilistic analysis based approach to model the input distribution. Due to this, we shall shift our focus on weighted average analysis instead of simple average analysis. Further, the probability based analysis helps in partitioning the set of inputs into a finite set of equivalence classes satisfying some property.

- The best case time complexity of an algorithm is $T(n) = \min(t(I))$, where $I \in D$
- The worst case time complexity of an algorithm is $T(n) = \max(t(I))$, where $I \in D$
- For the average case time complexity of an algorithm, $T(n) = \sum P(I) \cdot t(I),\ I \in D$, where $P(I)$ is the probability of occurrence of the input I and summation is taken over all inputs. This number is the weighted average over all inputs.

Remarks:
1. Best case time complexity is denoted using Ω notation. i.e., if an algorithm has time complexity $\Omega(n)$, then every input to the algorithm incurs at least $c \cdot n$ comparisons.
2. Worst case time complexity is denoted using O notation. i.e., if an algorithm has time complexity $O(n^2)$, then every input to the algorithm incurs at most $c \cdot n^2$ comparisons.
3. We use theta (θ) notation to analyze the run-time of a specific input with respect to an algorithm. For example, an already sorted input is a best case input to insertion sort whose run-time for this input is denoted as $\theta(n)$. Similarly, with respect to quick sort, the same input acts as a worst case input and its run-time is $\theta(n^2)$.
4. If an algorithm takes $\theta(n^2)$, then for all inputs, the algorithm incurs $\theta(n^2)$.

Note: The worst case and best case analysis of an algorithm can sometimes yield the same asymptotic bounds (for instance; merge sort) or can have different bounds (for example; insertion sort). For algorithms whose asymptotic time complexity of best case and worst case inputs are different, it is natural to look at the average case analysis of the algorithm. Further, for such algorithms, it is interesting to investigate whether the average case is close to the best case bound or worst case bound.
In this section, we shall analyze the average case of some classical algorithms.

- **Linear Search**
  Recall that linear search searches the given array A of size n linearly to check the containment of element x. Therefore in worst case we need n comparisons ($x \notin A$) and in the best case $\theta(1)$ comparisons ($x = A[1]$). Note that for best case x is present in $A[1]$ or one of the few locations close to $A[1]$. Similarly, for worst case, either $x \notin A$ or it is present in one of the locations close to $A[n]$. We shall now analyze average case by considering the probable position for x. For average case analysis, we shall focus on the probable position for x, rather than the elements of the array A.
  Since, our focus is on x, we classify the set of inputs into n equivalence classes. Equivalence class E_1 consists of those inputs such that x is present in $A[1]$ and equivalence class E_i consists of all inputs such that $A[i] = x$. For a successful search on any input A, the element x is equally likely to be present in any location of A or any equivalence class. Therefore, $Pr[A[i] = x] = \frac{1}{n}, 1 \leq i \leq n$.

  We do average case analysis by dividing the outcome of search into successful and unsuccessful search.

  - Cost of unsuccessful search $= n$. Clearly, after performing n comparisons, we would know that $x \notin A$.

  - Cost of successful search: since each location is equally likely, the cost of accessing the i^{th} location is i and the associated probability is $\frac{1}{n}$. Therefore, the cost is $\frac{1}{n} \cdot 1 + \frac{1}{n} \cdot 2 + \ldots + \frac{1}{n} \cdot n$ $= \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$

  - Since for an arbitrary input, we do not know whether it falls into successful/unsuccessful search, we bring in one more probability measure q that denotes the probability of successful search. Thus, the average cost $= q\times$ cost for successful search $+ (1-q)\times$ cost for unsuccessful search

  - $q \cdot \frac{n+1}{2} + (1-q) \cdot n$, where q is the probability of a successful search.
  - Intuitively, it is clear that if $q = 1$ (always success), then one has to look at 50% of locations of A. Due to this reason, the cost of successful search is $\frac{n+1}{2}$.

  Observations:

  1. The above analysis works fine as along as the input array contains distinct elements. If the array contains duplicates, then the argument 'x is equally likely to be in any location'

fails, and the probability varies for each location. As far as algorithm is concerned, the algorithm stops searching once it finds the first occurrence of x and does not bother about multiplicities of x.

2. If suppose the element x occurs r times in A, then the probability of occurrence of x is $\frac{r}{n}$. As part of search, the search terminates at the first appearance of x.
3. Therefore, the cost of successful search is
$$\sum_{i=1}^{n} p_i \cdot i$$
where p_i is the probability that x appears in $A[i]$. If q is the probability of success, then the average cost is
$$q \cdot \sum_{i=1}^{n} p_i \cdot i + (1-q) \cdot n$$
.

- **Binary Search**
In binary search, the sorted array A of size n is first checked at location $\frac{n}{2}$ for the containment of x. If the element is not found, then the search is recursively continued either in the lower half ($x < A[\frac{n}{2}]$) or in the upper half ($x > A[\frac{n}{2}]$). Similar to linear search, we shall analyze by considering the cost of successful and unsuccessful search separately. Note that if the search is an unsuccessful search, then starting from location $\frac{n}{2}$, it recursively looks for element x and finally the search terminates at the gap between the elements. There are $(n+1)$ gaps and as part of search discovery the last comparison will be to identify one of the $(n+1)$ gaps. For example, for the sorted array $\{1, 2, 3, 5, 7\}$, the gaps are denoted using G_i. If we search for 4, then the search will terminate at G_4.

_	1	_	2	_	3	_	5	_	7	_
G_1		G_2		G_3		G_4		G_5		G_6

Further, it takes $\lfloor \log n \rfloor + 1$ comparisons to identify one of the $(n+1)$-gaps. Therefore, we need exactly $\lfloor \log n \rfloor + 1$ comparisons to declare the result of an unsuccessful search.
For a successful search, the search for x terminates at one of the n elements in A and the position of x determines the cost of search. Similar to linear search, we focus only on the position of x and do not focus on the input (neither A nor x). If x is at $A[\frac{n}{2}]$, then we incur one comparison. If x is at $A[\frac{n}{4}]$ or $A[\frac{3n}{4}]$, then we incur two comparisons. Similarly, there are four possible locations for x that incur three comparisons.

Successful Search: Possible scenarios:

Cost	Number of positions for x	Probable positions for x
1	1	$\frac{1}{2}$
2	2	$\frac{1}{4}, \frac{3}{4}$
3	4	$\frac{1}{8}, \frac{3}{8}, \frac{5}{8}, \frac{7}{8}$
4	8	.....
5	16	.....
i	2^{i-1}	.....

Note that in case of linear search, for each cost (i.e., cost=1,2,3,...), there is exactly one position for x. i.e., if cost=3, then x is at $A[3]$. Whereas in binary search, many positions can compete for the same cost and all such positions must be taken into account for averaging argument. Towards this attempt, we divide the inputs into equivalence classes, based on the number of comparisons needed. There will be one equivalence class for all those inputs for which $x = A[\frac{n}{2}]$. There will be two equivalence classes for inputs such that x is at $A[\frac{n}{4}]$ or $A[\frac{3n}{4}]$, respectively. In general, we observe the following

Number of Comparisons	Number of equivalence classes
1	1
2	2
3	4
t	2^{t-1}

Since each position of A is equally likely for x, probability that $A[i] = x$ is $\frac{1}{n}$. Further, the cost of comparison to locate x lies in the range $[1..(\log n + 1)]$ and different x (locations of x are different) can take the same cost, and the number of different such x is precisely the number of equivalence classes which is given by the above table. Therefore,

Cost of successful search $= \sum\limits_{t=1}^{\log n+1} \frac{1}{n} \cdot t \cdot 2^{t-1} = \frac{1}{n} \sum\limits_{t=1}^{\log n+1} \frac{d}{d2}(2^t)$

$= \frac{1}{n} \frac{d}{d2} \sum\limits_{t=1}^{\log n+1} (2^t) = \frac{1}{n} \frac{d}{d2} 2^{\log n+2} - 1 = \frac{1}{n} 4\log n \cdot 2^{\log n-1} = \frac{1}{2n} 4\log n \cdot n^{\log 2} = \theta(\log n)$

Average cost $= q(\log n) + (1-q)(\log n + 1) = \theta(\log n)$ where q is the probability of successful search.

Another Approach: One can also view the cost incurred by binary search using the computational tree; which is a binary tree where each node represents a comparison between x, the element to be searched and an array element. The root of the binary tree represents the comparison 'whether x is same as $A[\frac{n}{2}]$'. In the next level, the left node represents the comparison whether 'x is same as $A[\frac{n}{4}]$' and the right node represents the comparison whether 'x is same as $A[\frac{3n}{4}]$' and so on. The total cost to search x is
$1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + \ldots + k \cdot 2^{k-1}$
The average cost $= \dfrac{\text{total cost}}{\text{number of nodes}} = \dfrac{1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + \ldots + k \cdot 2^{k-1}}{n}$

Note that k is the depth of the tree and $k = \lfloor \log n \rfloor$. This expression is similar as the one discussed in the previous section and hence, the cost of successful search is $\theta(\log n)$.

- **Insertion Sort**
  Approach 1: Insertion sort is an incremental sort in which at the end of iteration i, the first i elements of the array A are sorted. During i^{th} iteration, the position for $A[i]$ is linearly searched in $A[1..i]$ and necessary swapping is done once the right position is found out. Note that the element may be placed at $A[i]$ itself. This implies that the average time complexity of insertion sort equals the sum of the average time required to search the position for i^{th} element for every $i \geq 2$. Note that a linear search on i elements on an average takes $\frac{i+1}{2}$. Therefore, the average cost of insertion sort is precisely $\sum\limits_{i=2}^{n} \frac{i+1}{2}$, which is $\theta(n^2)$.

  Approach 2: The location where $A[i]$ would be placed is equally likely in $[1..i]$. The probability p_j that $A[i]$ goes into one of $A[1..i]$ is $\frac{1}{i}$. The comparison costs for

- placing $A[i]$ at position i is 1
- placing $A[i]$ at position $i-1$ is 2
- ...
- placing $A[i]$ at position 2 is $i-1$
- placing $A[i]$ at position 1 is also $i-1$

Therefore, the average cost for placing $A[i]$ into the right position in $[1..i]$ =
$\sum\limits_{j=1}^{i} j \cdot p_j = \frac{1}{i}[1+2+\cdots+(i-1)+(i-1)] = \frac{1}{i}(\frac{i(i-1)}{2}+i-1) = \frac{i-1}{2}+\frac{i-1}{i}$
Thus, the average cost of insertion sort =
$\sum\limits_{i=2}^{n} \frac{i-1}{2}+\frac{i-1}{i} = \theta(n^2)$

Approach 3: An inversion in an array A refers to a pair (a_i, a_j) having the property $i < j$ and $a_i > a_j$. The goal of any sorting algorithm is to remove all inversions. A sorted array, for example $(1, 2, 3, 4, 5)$ has no inversions and the array $(5, 4, 3, 2, 1)$ has 10 inversions. In general, the array sorted in reverse has $\binom{n}{2}$ inversions. Note that the insertion sort gradually removes all inversions present in the input array and the number of inversions is a random variable. Further, the number of inversions is same as the number of swaps performed by the insertion sort algorithm. As part of average case analysis, we wish to find the expected number of inversions which in turn captures the expected number of swaps. Note that in this analysis, our primitive operation is the number of swaps instead of the number of comparisons. We assume for an input of size n each of the $n!$ permutations of the input is equally likely.
We define the indicator random variable X as follows;
$X_{ij} = 1$ if there is an inversion between a_i and a_j
$= 0$ otherwise
Note that the probability (inversion between a_i and a_j) = $\frac{1}{2}$ and also probability (no inversion between a_i and a_j) = $\frac{1}{2}$.
$E(X_{ij}) = \frac{1}{2}\cdot 1 + \frac{1}{2}\cdot 0 = \frac{1}{2}$
Expected number of inversions = $E(X) = \sum_{i,j} E(X_{ij}) = \binom{n}{2} \cdot \frac{1}{2} = \theta(n^2)$.
Therefore, on an average, the insertion sort performs $\theta(n^2)$ swaps to output a sorting sequence.

From the perspective of equivalence class theory, the set of all inputs is partitioned into $\binom{n}{2}$ equivalence class where E_i denotes the set of inputs with inversion i. Any input is equally likely to come from any of the $m = \binom{n}{2}$ equivalence classes. The cost of removing i inversions is i. Therefore, the expected cost is $\frac{1}{m}\sum_{i=1}^{m} i$ which is $\theta(n^2)$.

- **Quick Sort**
  Quick sort is a divide and conquer approach to sort the given array. The algorithm first identifies a special element, namely *pivot* element. Then it partitions the whole array into two, one having values less than the pivot and the other having values greater than the pivot. Usually, the first element or the last element of the array is taken as the pivot element. Then it recursively calls the quick sort algorithm in both sub arrays. Therefore the recurrence for the quick sort for an arbitrary input is, $T(n) = T(q) + T(n-q-1) + \theta(n)$. If every time during the recursion, we get a balanced partition, then, $T(n) = T(\frac{n}{2}) + T(\frac{n}{2}-1) + O(n)$ which

is $O(n \log n)$.
If every time during the recursion, we obtain a skewed partition, then $T(n) = T(0) + T(n-2) + O(n) = O(n^2)$.
However, for an arbitrary input, we may not be able to guess the structure of the partition. As part of average case analysis, we assume that each of $n!$ permutations of the input is equally likely and calculate the average number of comparisons done by the partition routine over all calls to that routine. We do not focus on the structure of the partition or the input, instead, we focus on when exactly two elements of A are compared.

Average case analysis of Quick sort
As part of quick sort, comparison between elements takes place at the time of partition (divide phase) and there is no comparison cost as part of conquer phase (combine phase). We shall focus our attention on partition routine and analyze the average time complexity. Note that two elements a_i, a_j of the input array A are compared if both a_i and a_j occur in the same partition A', and either a_i is a pivot or a_j is a pivot. We make use of an indicator random variable X_{ij} which is defined as follows;

$X_{ij} = 1$ if a_i is compared with a_j
$= 0$, otherwise
Since, we do not know how many X_{ij}'s will be '1', it is natural to look at the expectation of X_{ij}'s.
The expected value of X equals $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

$E(X) = E[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}]$

$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$

$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 \cdot Pr[X_{ij} = 1] + 0 \cdot Pr[X_{ij} = 0]$
Note that $Pr[X_{ij} = 1] = Pr[a_i$ is a pivot or a_j is a pivot $]$ and
$Pr[a_i$ is a pivot $] = Pr[a_j$ is a pivot $] = \dfrac{1}{j-i+1}$
Therefore, $Pr[X_{ij} = 1] = \dfrac{1}{j-i+1} + \dfrac{1}{j-i+1} = \dfrac{2}{j-i+1}$

$E(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \dfrac{2}{j-i+1}$

Let $j - i = k$, then $E(X) = \sum_{i=1}^{n-1} \sum_{k=1}^{n} \dfrac{2}{k+1}$

$E(X) \leq \sum_{i=1}^{n-1} \sum_{k=1}^{\infty} \dfrac{2}{k+1}$

$= \sum_{i=1}^{n-1} 2 \cdot \log n$
$= O(n \log n)$
Thus, quick sort on an average performs $O(n \log n)$ comparisons.

Another Approach: Since the size of the partition is a random variable, all sizes of the subarray from 0 to $n-1$ are equally likely. We assume that all $n!$ permutations of the input are equally likely and also, the pivot can be any one of n elements of the input array. Let $A(n)$ denote the average time to sort an array of size n, then

$$A(n) = \frac{1}{n}[\sum_{r=1}^{n} A(r-1) + A(n-r)] + O(n)$$

where $A(r-1)$ is the average time to sort a subarray of size $(r-1)$. On simplification, we get,

$$A(n) = \frac{1}{n}\sum_{r=1}^{n} 2A(r-1) + O(n)$$

. Further,

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + O(n)$$

. On further simplification, we get,

$$\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{O(n)}{n(n+1)}$$

.
Let $a_n = \frac{A(n)}{n+1}$, then the above recurrence becomes,

$$a_n = a_{n-1} + \frac{O(n)}{n(n+1)}$$

which on simplification, we get $a_n = O(\log n)$. Thus, $A(n) = O(n \log n)$.

- **Merge Sort**
  `Approach:1` Since the best and worst case running time of the merge sort is $n \log n$, the average case is also $n \log n$. However, we shall present an explicit analysis by focusing on the *MERGE()* routine of merge sort to show that the average running time of the merge sort is $O(n \log n)$.
  `Approach:2` Given an array of size n, we subdivide the array into two sub arrays, and do this recursively till the size becomes one. Merge the sorted array recursively in a bottom-up fashion till we get the sorted array of size n. Note that given two sorted arrays of size m, n, we need $m+n-1$ comparisons in the worst case and $min(m,n)$ in the best case to merge two sorted arrays to obtain a sorted array of size $m+n$. In any case, it requires at most $m+n-1$ comparisons.
  When the recursion bottoms out as part of divide phase, the MERGE() routine is invoked which starts merging the sorted arrays. It is easy to see that at the last but one level of the recursion tree there are $\frac{n}{2}$ merges of 1 comparison each on average. In the next level (last but two) there are $\frac{n}{4}$ merges of 2 comparisons each on average, which is $\frac{n}{2}$. In general, note that there are $\frac{n}{2}$ comparisons in each level. Therefore the number of comparisons on average case equals $\log n \cdot \frac{n}{2} = \theta(n \log n)$.

- **Heap Sort**
  For discussion, we shall work with a max-heap and the objective is to sort an array A in

increasing order. We already know that a max-heap can be constructed in $O(n)$ time. As part of heap sort, we iteratively perform EXTRACT-MAX() and swap it with $A[n]$, $A[n-1]$, and so on. i.e., max will be placed at $A[n]$ during iteration-1, second max will be placed at $A[n-1]$. While swapping, max-heap property may be violated which further invokes MAX-HEAPIFY() routine and it takes $O(h) = O(\log n)$ to set right the max-heap property. On an average, during iteration-1, MAX-HEAPIFY() incurs $\frac{\log n}{2}$ comparisons and during iteration-2, it incurs $\frac{\log(n-1)}{2}$ comparisons.

Therefore, the total number of comparisons on the average equals $\sum_{i=n}^{1} \frac{\log i}{2} = \frac{1}{2}\log n! = \theta(n \log n)$.

In the below table, we shall summarize our discussion on average case analysis of sorting algorithms by comparing it with best and worst case analysis.

	Best case	Worst case	Average case
Bubble sort	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$
Selection sort	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$\Omega(n)$	$O(n^2)$	$\theta(n^2)$
Merge sort	$\Omega(n \log n)$	$O(n \log n)$	$\theta(n \log n)$
Quick sort	$\Omega(n \log n)$	$O(n^2)$	$\theta(n \log n)$
Heap sort	$\Omega(n \log n)$	$O(n \log n)$	$\theta(n \log n)$

Acknowledgements: Lecture contents presented in this module and subsequent modules are based on the following text books and most importantly, author has greatly learnt from lectures by algorithm exponents affiliated to IIT Madras/IMSc; Prof C. Pandu Rangan, Prof N.S.Narayanaswamy, Prof Venkatesh Raman, and Prof Anurag Mittal. Author sincerely acknowledges all of them. Special thanks to Teaching Assistants Mr.Renjith.P and Ms.Dhanalakshmi.S for their sincere and dedicated effort and making this scribe possible. Author has benefited a lot by teaching this course to senior undergraduate students and junior undergraduate students who have also contributed to this scribe in many ways. Author sincerely thank all of them.

References:

1. E.Horowitz, S.Sahni, S.Rajasekaran, Fundamentals of Computer Algorithms, Galgotia Publications.
2. T.H. Cormen, C.E. Leiserson, R.L.Rivest, C.Stein, Introduction to Algorithms, PHI.
3. Sara Baase, A.V.Gelder, Computer Algorithms, Pearson.