Notes on

Algorithms and Complexity

Csc. 540

Unit 1.1

Advanced Algorithm Analysis Techniques

Prepared and compiled by

Pradyumna Bhattarai

Scholar

First Semester, Msc. CSIT

CDCSIT

Table of Contents

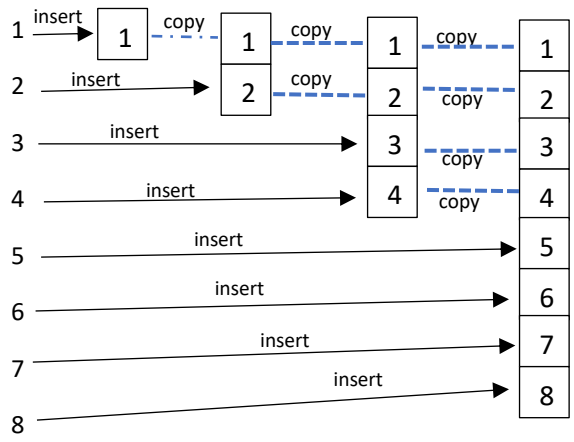
1.1 Advance Algorithm Analysis Techniques:

1.1.1 Amortized Analysis:

In an amortized analysis, we average the time required to perform a sequence of data-structure operations over all the operations performed. With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive. Amortized analysis differs from average-case analysis in that probability is not involved; an **amortized analysis guarantees the average performance of each operation in the worst case.**

It is an average cot analysis. It is required because all operations within a set is not worst. Therefore, if an algorithm average cost is better than it is worthwhile to implement.

Example: Hash Table



Following points are considered:

- Insert and Copy Operation Take One Cost
- Table Doubling cost not added

Size, Insert Cost, copy cost and Actual Cost are now tabulated as below:

i	1	2	3	4	5	6	7	8	9	10
size i	1	2	4	4	8	8	8	8	16	16
Insert Cost	1	1	1	1	1	1	1	1	1	1
Copy Cost	0	1	2	0	4	0	0	0	8	0
Actual Cost (C_i)	1	2	3	1	5	1	1	1	9	1

Aggregate Analysis:

In aggregate analysis, we show that for all n, a sequence of n operations takes worst-case time T (n) in total. In the worst case, the average cost, or amortized cost, per operation is therefore T (n)/ n.

$$Ci = n + \sum_{j=0}^{\lfloor \log_2(n-1) \rfloor} 2^j$$

Generalizing equation: = n + 2n -3

= 3n-3

Which is O(n).

Let, $\widehat{C_\iota}$ be an amortize cost,

$\widehat{C_\iota} = \theta(n) = \frac{O(n)}{n} = \frac{n}{n} = 1$

Amortized cost is constant. i.e., O (1).

Accounting Method:

Considering, the cost is added or reduced from the bank. And Bank >= 0 .

$\widehat{C_\iota} = 3$, Considering every 1 operation cost as 3 , we get the table tabulated as below:

Actual Cost (C_i)	1	2	3	1	5	1	1	1	9	1
$\widehat{C_\iota}$	3	3	3	3	3	3	3	3	3	3
Bank	2	3	3	5	3	5	7	9	3	5

If every 1 operation cost is 3.

The n operation cost 3n.

Therefore, $\theta(n) = \frac{3n}{n} = 3$

Hence, $\theta(n) = 1$ which is constant.

Potential Method:

The potential method works as follows. We will perform n operations, starting with an initial data structure D_0. For each i = 1 ,2 ,3,4, …. n, we let c_i be the actual cost of the i^{th} operation and D_i be the data structure that results after applying the i^{th} operation to data structure D_{i-1}. A potential function (ϕ) maps each data structure D_i to a real number ϕ(D_i), which is the potential associated with data structure D_i . The amortized cost $\widehat{C_\iota}$ of the i^{th} operation with respect to potential function is defined by:

$$\widehat{C_\iota} = c_i + \emptyset(D_i) - \emptyset(D_{i-1})$$

Let's assume, $\emptyset(D_i) = 2i - 2^{\lceil \log i \rceil}$

$$\emptyset(D_{i-1}) = 2(i-1) - 2^{\lceil \log(i-1) \rceil}$$

$$\widehat{C_i} = \begin{cases} i & if\ i-1\ is\ an\ exact\ power\ of\ 2 \\ 1 & otherwise \end{cases} + 2i - 2^{\lceil \log i \rceil} - 2(i-1) - 2^{\lceil \log(i-1) \rceil}$$

Case I: Taking i,

$= i + 2i - \mathbf{2^{\lceil \log i \rceil}} - 2i + 2 - 2^{\lceil \log(i-1) \rceil}$

$= i - 2(\boldsymbol{i} - \mathbf{1}) + 2 + (\boldsymbol{i} - \mathbf{1})$

$= i - 2i + 2 + 2 + i - 1$

= 3

Case II: Taking 1,

$= 1 - \mathbf{2^{\lceil \log i \rceil}} + 2 - 2^{\lceil \log(i-1) \rceil}$

$= 1 - (i-1) + 2 + (i-1)$

= 3

So, from case I and II, we can see the amortized cost is 3 which is constant.

1.1.2 Probabilistic Analysis

Probabilistic analysis is the use of probability in the analysis of problems. Most commonly, we use probabilistic analysis to analyze the running time of an algorithm. Sometimes we use it to analyze other quantities, such as the hiring cost in procedure HIRE-ASSISTANT. In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs. Then we analyze our algorithm, computing an average-case running time, where we take the average over the distribution of the possible inputs. Thus, we are, in effect, averaging the running time over all possible inputs. When reporting such a running time, we will refer to it as the **average-case running time.**

We must be very careful in deciding on the distribution of inputs. For some problems, we may reasonably assume something about the set of all possible inputs, and then we can use probabilistic analysis as a technique for designing an efficient algorithm and as a means for gaining insight into a problem. For other problems, we cannot describe a reasonable input distribution, and in these cases we cannot use probabilistic analysis.

Hiring Problem:

Suppose that you need to hire a new office assistant. Your previous attempts at hiring have been unsuccessful, and you decide to use an employment agency. The employment agency sends you one candidate each day. You interview that person and then decide either to hire that person or not. You must pay the employment agency a small fee to interview an applicant. To actually hire an applicant is more costly, however, since you must fire your current office assistant and pay a substantial hiring fee to the employment agency. You are committed to having, at all times, the best possible person for the job. Therefore, you decide that, after interviewing each applicant, if that applicant is better qualified than the current office assistant, you will fire the current office assistant and hire the new applicant. You are willing to pay the resulting price of this strategy, but you wish to estimate what that price will be.

The procedure HIRE-ASSISTANT, given below, expresses this strategy for hiring in pseudocode. It assumes that the candidates for the office assistant job are numbered 1 through n. The procedure assumes that you are able to, after interviewing candidate i, determine whether candidate i is the best candidate you have seen so far. To initialize, the procedure creates a dummy candidate, numbered 0, who is less qualified than each of the other candidates.

```
HIRE-ASSISTANT(n)
best D = 0 // candidate 0 is a least-qualified dummy candidate
for i =1 to n
      interview candidate i
      if candidate i is better than candidate best
            best =i
            hire candidate i
```

Interviewing has a low cost, say c_i, whereas **hiring** is expensive, costing c_h. Letting m be the number of people hired, the total cost associated with this algorithm is **$O(c_in + c_hm)$.** No matter how many people we hire, we always interview n candidates and thus always incur the cost c_in associated with

interviewing. We therefore concentrate on analyzing $c_h m$, the hiring cost. This quantity varies with each run of the algorithm.

Worst-case analysis

In the worst case, we actually hire every candidate that we interview. This situation occurs if the candidates come in strictly increasing order of quality, in which case we hire n times, for a total hiring cost of **$O(c_h n)$.**

Of course, the candidates do not always come in increasing order of quality. In fact, we have no idea about the order in which they arrive, nor do we have any control over this order. Therefore, it is natural to ask what we expect to happen in a typical or average case.

1.1.3 Monte Carlo and Las Vegas

Randomized Algorithms:

A randomized algorithm is one that makes use of a randomizer (such as a random number generator). Some of the decisions made in the algorithm depend on the output of the randomizer. Since the output of any randomizer might differ in an unpredictable way from run to run, the output of a randomized algorithm could also differ from run to run for the same input. The execution time of a randomized algorithm could also vary from run to run for the same input.

Randomized algorithms can be categorized into two classes: The first is algorithms that always produce the same(correct)output for the same input. These are called **Las Vegas algorithms**. The execution time of a Las Vegas algorithm depends on the output of the randomizer. If we are lucky, the algorithm might terminate fast, and if not, it might run for a longer period of time. In general, the execution time of a Las Vegas algorithm is characterized as a random variable.

The second is algorithms whose outputs might differ from run to run (for the same input). These are called **Monte Carlo algorithms**. Consider any problem for which there are only two possible answers, say, yes and no. If a Monte Carlo algorithm is employed to solve such a problem, then the algorithm might give incorrect answers depending on the output of the randomizer. We require that the probability of an incorrect answer from a Monte Carlo algorithm be low.

Typically, for a fixed input, a Monte Carlo algorithm does not display much variation in execution time between runs, whereas in the case of a Las Vegas algorithm this variation is significant.

Problem Statement:

Given an array with n elements A[1…n]. Half of the array contains 0s, the other half contains 1s.

Goal:

Finding an index that contains a 1

Las Vegas

```
repeat:
      k = RandInt(n)
      if A[k] = 1, return k
```

Las Vegas: Output is always correct, Ex: Randomized Quicksort

Monte Carlo

```
repeat 300 times:
      k = RandInt(n)
      if A[k] = 1 , return k
return "Failed"
```

Monte Carlo: Output may be incorrect, Randomized Median

Another Example:

General Search for Repeating number:

```
AlgoSearchRepeat(A)
{
for i = 0 to n-1
      for j=i+1 to n
            if A[i] = = A[j]
            return true
}
```

General Search:

```
AlgoSearch(A,a)
{
for i=0 to n-1
      if A[i] == a
      return True
}
```

Las Vegas Search:

```
AlgoLasVegasSearchRepeat(A)
{
while( true ) do
      i = random() mod n+1
      j= random() mod n+1
      if ( i != j and A[i] == A[j])
            return True
}
```

Monte Carlo Search:

```
AlgoMonteCarloSearch (A,a, x)
{
for i = 0 to n-1
      j = random() mod n+1
      if ( A[j] ==a )
      return True
}
```

Exa**mple 2 (Primality Testing):**

Any integer greater than one is said to be a prime if its only divisors are 1 and the integer itself. By convention, we take 1 to be a non-prime. Then 2,3,5,7,11, and 13arethe first six primes. Given an integer n, the problem of deciding whether n is a prime is known as primality testing.

LasVegas()

```
{
        while (true) do
        {
        i=Random() mod 2;
        if (i >= 1)then return;
}
}
```