Amortized Analysis

Amortized analysis is generally used for certain algorithms where a sequence of similar operations are performed.

Amortized analysis provides a bound on the actual cost of the entire sequence, instead of bounding the cost of sequence of operations separately.

Amortized analysis is similar to average-case analysis in that it is concerned with the cost averaged over a sequence of operations.

Average case analysis relies on probabilistic assumptions about the data structures and operations in order to compute an expected running time of an algorithm.
Its applicability is therefore dependent on certain assumptions about the probability distribution of algorithm inputs.

Amortized analysis differs from average-case analysis. Probability is not involved in amortized analysis. Amortized analysis guarantees the average performance of each operation in the worst case.

E.g., Dynamic array insertion.
One algorithm is to allocate a fixed size array, and as new elements are inserted, allocate a fixed size array of double the old length when necessary.
In the worst case an insertion of an element can require time proportional to the length of the entire list, so in the worst case insertion is an O(n) operation.
However such a worst case is infrequent, so insertion is an O(1) operation using amortized analysis.
Amortized analysis holds no matter what the input is.

Another example,
Discuss:: Searching for a minimum value in an array.

It is not just a tool for analysis, it's a way of thinking about the design, since designing and analysis are closely related.

Aggregate Method

The aggregate method gives a global view of a problem.

In this method, if n operations takes worst-case time T(n) in total.

Then the amortized cost of each operation is T(n)/n, though different operations may take different time, in this method varying cost is neglected.

Accounting Method

Different charges are assigned to different operations according to their actual cost.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by T(n), where n is the input size.

Different types of asymptotic notations are used to represent the complexity of an algorithm.

Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- O − Big Oh
- Ω − Big omega
- θ − Big theta
- o − Little Oh
- ω − Little omega

Apriori and Apostiari Analysis

Apriori analysis means, analysis is performed prior to running it on a specific system.

In this type of analysis the function is defined using some theoretical model. Therefore, the time and space complexity of an algorithm is assessed by just looking at the algorithm rather than running it on a particular system with a different memory, processor, and compiler.

Apostiari analysis of an algorithm means we perform analysis of an algorithm only after running it on a system. It directly depends on the system and changes from system to system.

In an industry, we cannot perform Apostiari analysis as the software is generally made for an anonymous user, which runs it on a system different from those present in the industry.

In Apriori, it is the reason that we use asymptotic notations to determine time and space complexity as they change from computer to computer; however, asymptotically they are the same.

Different Algorithms and their Strategies

Divide and Conquer approach

A problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Generally, divide-and-conquer algorithms have three parts −

- Divide the problem into a number of sub-problems that are smaller instances of the same problem.
- Conquer the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- Combine the solutions to the sub-problems into the solution for the original

problem.

Pros and cons of Divide and Conquer Approach

Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

In this approach, most of the algorithms are designed using recursion. Therefore, memory requirement is relatively high as for recursive function execution stack is used, where function's execution state needs to be stored.

Application of Divide and Conquer Approach

Following are some problems, which are solved using divide and conquer approach.

- Finding the maximum and minimum of a sequence of numbers
- Merge sort
- Binary search

Example - Analysis of finding max and min of set of numbers using different algorithms.

Naïve Method

Naïve method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```
Algorithm: Max-Min-Element (numbers[])
max := numbers[1]
min := numbers[1]

for i = 2 to n do
   if numbers[i] > max then
      max := numbers[i]
   if numbers[i] < min then
      min := numbers[i]
return (max, min)
```

Analysis

The number of comparison in Naive method is 2n - 2.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach Analysis

In this approach,

- The array is divided into two halves,
- Use recursive approach to find max and min for each half,
- Return the maximum of two max of each half and the min of two minima of each half.

```
Algorithm: Max - Min(x, y)
if y == x then
   return number[x] as min and number[x] as max.
if y – x == 1 then
   return (max(numbers[x], numbers[y]), min((numbers[x], numbers[y]))
else
   (max1, min1):= maxmin(x, ((x + y)/2))
   (max2, min2):= maxmin(((x + y)/2) + 1),y)
return (max(max1, max2), min(min1, min2))
```


Analysis

Let T(n) be the number of comparisons made by , where the number of elements n = y - x + 1

If T(n) represents the numbers, then the recurrence relation can be represented as

```
T(n) = 0 if n = 1
     = 1 if n = 2
     = T(n/2) + T(n/2) + 2 if n > 2
```

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less.

However, using the asymptotic notation both of the approaches end up in O(n).

Greedy Algorithm

Amongst all the algorithmic approaches, the simplest and straightforward approach is the Greedy method.

In this approach, the decision is taken on the basis of current available information without worrying about the effect of the current decision in future.

Greedy algorithms build a solution

  part by part,
  choosing the next part in such a way, that it gives an immediate benefit.

This approach never reconsiders the choices taken previously.
This approach is mainly used to solve optimization problems.

Greedy method is easy to implement and quite efficient in most of the cases.
One can say that Greedy algorithm is an algorithmic paradigm based on heuristic that follows local optimal choice at each step with the hope of finding global optimal solution.

In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

Components of Greedy Algorithm

Greedy algorithms have the following five components −

    A candidate set − A solution is created from this set.

A selection function − Used to choose the best candidate to be added to the solution.

A feasibility function − Used to determine whether a candidate can be used to contribute to the solution.

An objective function − Used to assign a value to a solution or a partial solution.

A solution function − Used to indicate whether a complete solution has been reached.

Areas of Application

Greedy approach is used to solve many problems, such as

Finding the shortest path between two vertices using Dijkstra’s algorithm.

Finding the minimal spanning tree in a graph using Prim’s /Kruskal’s algorithm, etc.

Where Greedy Approach Fails

In many problems, Greedy algorithm fails to find an optimal solution, moreover it may produce a worst solution.

Explain Travelling Salesman problem.

Problems like Travelling Salesman can't be solved by this strategy. E.g., if salesman needs to travel n cities, there are (n-1)! possibilities.

What about Heuristic approach?

Heuristics approaches use a set of guiding rules for selection of the next node. As this result in approximations, it will not always give the optimal solution. High quality admissible heuristics can find a useful solution in a fraction of the time required for a full brute force of the problem.

Backtracking algorithm

Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.[1][2]

The classic textbook example of the use of backtracking is the eight queens puzzle, that asks for all arrangements of eight chess queens on a standard chessboard so that no queen attacks any other.

In the common backtracking approach, the partial candidates are arrangements of k queens in the first k rows of the board, all in different rows and columns. Any partial solution that contains two mutually attacking queens can be abandoned.

Backtracking can be applied only for problems which admit the concept of a "partial candidate solution" and a relatively quick test of whether it can possibly be completed to a valid solution.

It is useless, for example, for locating a given value in an unordered table. When it is applicable, however, backtracking is often much faster than brute force enumeration of all complete candidates, since it can eliminate a large number of candidates with a single test.

Backtracking is an important tool for solving constraint satisfaction problems, such as crosswords, verbal arithmetic, Sudoku, and many other puzzles.

Brute Force Algorithm
==========

In computer science, brute-force search or exhaustive search, also known as generate and test, is a very general problem-solving technique and algorithmic paradigm that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

E.g.,
1. A brute-force algorithm to find the divisors of a natural number n would enumerate all integers from 1 to n, and check whether each of them divides n without remainder.
2. A brute-force approach for the eight queens puzzle would examine all possible arrangements of 8 pieces on the 64-square chessboard, and, for each arrangement, check whether each (queen) piece can attack any other.

While a brute-force search is simple to implement, and will always find a solution if it exists, its cost is proportional to the number of candidate solutions – which in many practical problems tends to grow very quickly as the size of the problem increases (combinatorial explosion).
Therefore, brute-force search is typically used when the problem size is limited, or when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size. The method is also used when the simplicity of implementation is more important than speed.

Dynamic Programming
==========

Dynamic Programming is used in optimization problems.

solves problems by combining the solutions of subproblems.
solves each sub-problem just once and saves its answer in a table

Two main properties of a problem suggest that the given problem can be solved using Dynamic Programming. These properties are overlapping sub-problems and optimal substructure.

Overlapping Sub-Problems

Similar to Divide-and-Conquer approach, Dynamic Programming also combines solutions to sub-problems. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don’t have to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.
In mathematics, the Fibonacci numbers, commonly denoted Fn form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,[1]

Explain Fibonacci sequence.

Optimal Sub-Structure

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its sub-problems.

For example, the Shortest Path problem has the following optimal substructure property −

If a node x lies in the shortest path from a source node u to destination node v, then the shortest path from u to v is the combination of the shortest path from u to x, and the shortest path from x to v.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps −

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem

LCS Problem Statement: Given two sequences, find the length of longest subsequence present in both of them.

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.

For example, “abc”, “abg”, “bdf”, “aeg”, ‘”acefg”, .. etc are subsequences of “abcdefg”. So a string of length n has 2^n different possible subsequences.

Branch and Bound Algorithm

The goal of a branch-and-bound algorithm is to find a value x that maximizes or minimizes the value of a real-valued function f(x), called an objective function, among some set S of admissible, or candidate solutions. The set S is called the search space, or feasible region. The rest of this section assumes that minimization of f(x) is desired; this assumption comes without loss of generality, since one can find the maximum value of f(x) by finding the minimum of g(x) = −f(x). A B&B algorithm operates according to two principles:

It recursively splits the search space into smaller spaces, then minimizing f(x) on these smaller spaces; the splitting is called branching.

Branching alone would amount to brute-force enumeration of candidate solutions and testing them all. To improve on the performance of brute-force search, a B&B algorithm keeps track of bounds on the minimum that it is trying to find, and uses these bounds to "prune" the search space, eliminating candidate solutions that it can prove will not contain an optimal solution.

Turning these principles into a concrete algorithm for a specific optimization problem requires some kind of data structure that represents sets of candidate solutions. Such a representation is called an instance of the problem.

E.g., 0/1 Knapsack problem to understand Branch and Bound.

Given two integer arrays val[0..n-1] and wt[0..n-1] that represent values and weights associated with n items respectively.

Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W.

Stochastic Methods

Stochastic optimization (SO) methods are optimization methods that generate and use random variables.

For stochastic problems, the random variables appear in the formulation of the optimization problem itself, which involves random objective functions or random constraints.

Stochastic optimization methods also include methods with random iterates. Some stochastic optimization methods use random iterates to solve stochastic problems, combining both meanings of stochastic optimization.

Stochastic optimization methods generalize deterministic methods for deterministic problems.

E.g.,

{Xn:n= 0,1,2, . . .},where the state space of Xn is {0,1,2,3,4} representing which of four types of transactions a person submits to an on-line data-base service, and time n corresponds to the number of transactions submitted.

{Xn:n= 0,1,2, . . .},where the state space of Xn is {1,2} representing whether an electronic component is acceptable or defective, and time n corresponds to the number of components produced.