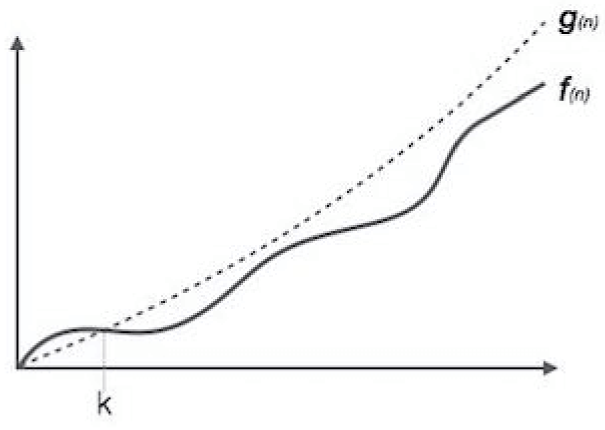
DESIGN & ANALYSIS OF ALGORITHMS (1)

techworldthink • March 08, 2022

1. Define Big Oh notation.

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound. So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



Asymptotic Analysis

For example:

If f(n) and g(n) are the two functions defined for positive integers,

then f(n) = O(g(n)) as f(n) is big oh of g(n) or f(n) is on the order of g(n)) if there exists constants c and no such that:

f(n)≤c.g(n) for all n≥no

This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n). In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

Let's understand through examples

Example 1: f(n)=2n+3 , g(n)=n

Now, we have to find Is f(n)=O(g(n))?

To check f(n)=O(g(n)), it must satisfy the given condition:

f(n)<=c.g(n)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

Let's assume c=5, n=1 then , 2*1+3<=5*1 , 5<=5

For n=1, the above condition is true.

If n=2 , 2*2+3<=5*2 , 7<=10

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n. We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants no, it will always satisfy 2n+3<=c.n. As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n).

2. Write the control abstraction for a typical Divide and Conquer algorithm.

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort theelements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide–and–conquer is a very powerful use of recursion.

```
DANDC (P) {
        if SMALL (P) then return S (p);
        else {
                divide p into smaller instances p1, p2, …. Pk, k >= 1;
                apply DANDC to each of these sub problems;
                 return (COMBINE (DANDC (p1) , DANDC (p2),…., DANDC (pk));
        }
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function ‘S’ is invoked otherwise, the problem ‘p’ into smaller sub problems. These sub problems p1, p2, . . . , pk are solved by recursive application of DANDC.

If the sizes of the two sub problems are approximately equal then the computing time of DANDC is:

$Tn = g(n)$, for small n

$Tn = (a.T(n/b)) + f(n)$, otherwise

Where, T (n) is the time for DANDC on ‘n’ inputs *g (n) is the time to complete the answer directly for small inputs*

f (n) is the time for Divide and Combine

a is no. of subproblems

T(n/b) is size of each problems

eg : merge sort recurence

$$\boxed{T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)}$$

Recurrence Relation

On solving this recurrence relation, we get T(n) = Θ(nlogn)

3. Explain a Greedy strategy which can give the optimal solution for the Knapsack problem.

The **knapsack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

Given weights and values of n items, we need to put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

In the **0-1 Knapsack problem**, we are not allowed to break items. We either take the whole item or don't take it.

A **brute-force solution** would be to try all possible subset with all different fraction but that will be too much time taking.

An **efficient solution** is to use Greedy approach. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.

In Fractional Knapsack Problem,

- As the name suggests, items are divisible here.
- We can even put the fraction of any item into the knapsack if taking the complete item is not possible.
- It is solved using Greedy Method.

Fractional knapsack problem is solved using greedy method in the following steps-

- For each item, compute its value / weight ratio.
- Arrange all the items in decreasing order of their value / weight ratio.
- Start putting the items into the knapsack beginning from the item with the highest ratio.
- Put as many items as you can into the knapsack.

Time Complexity-

- The main time taking step is the sorting of all items in decreasing order of their value / weight ratio.
- If the items are already arranged in the required order, then while loop takes O(n) time.
- The average time complexity of Quick Sort is O(nlogn).

- Therefore, total time taken including the sort is O(nlogn).

4. Write a dynamic programming algorithm to compute the factorial of a number.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

There are two approaches to dynamic programming:

- Top-down approach (use recursion)
- Bottom-up approach (use iteration)
-

Dynamic programming algorithm to compute the factorial of a number

```
fact(n){
    ans = 1;
    for (int i = 2; i <= n; ++i) {
        ans *= i ;
    }
    return ans;
}
```

5. How does Backtracking differ from Branch and Bound?

Backtracking

Backtracking is a problem-solving technique so it solves the decision problem.

When we find the solution using backtracking then some bad choices can be made.

Backtracking uses a Depth first search.

The state space tree is searched until the solution of the problem is obtained.

In backtracking, all the possible solutions are tried. If the solution does not satisfy the constraint, then we backtrack and look for another solution.

Applications of backtracking are n-Queens problem, Sum of subset.

Backtracking is more efficient than the Branch and bound.

It contains the feasibility function.

Backtracking solves the given problem by first finding the solution of the subproblem and then recursively solves the other problems based on the solution of the first subproblem.

Branch and bound

Branch n bound is a problem-solving technique so it solves the optimization problem.

When we find the solution using Branch n bound then it provides a better solution so there are no chances of making a bad choice.

It is not necessary that branch n bound uses Depth first search. It can even use a Breadth-first search and best-first search.

The state space tree needs to be searched completely as the optimum solution can be present anywhere in the state space tree.

In branch and bound, based on search; bounding values are calculated. According to the bounding values, we either stop there or extend.

Applications of branch and bound are knapsack problem, travelling salesman problem, etc.

Branch n bound is less efficient.

It contains the bounding function.

Branch and bound solves the given problem by dividing the problem into two atleast subproblems.