Minor Degree Program

Algorithm Design Techniques

Data Structures and Algorithms

Prakash B Hegade
School of CSE, KLE TECH

Important Note:

The contents of this note are tailored from the data available through Wikipedia, textbooks mentioned in the course and other relevant references. The Objective of this note is to provide the summary of different design techniques studied in the program tenure and a revision note that could also help for placements.

Techniques covered:

1. Brute Force
2. Decrease and Conquer
3. Divide and Conquer
4. Transform and Conquer
5. Dynamic Programming
6. Greedy Technique
7. Space and Time tradeoff
8. Randomized Algorithms
9. Backtracking

Note:

There are many other techniques like branch and bound, liner programming etc that you can look for.

1. Brute Force

Brute force is a straight forward approach to solving a problem, usually, directly based on the problem statement and definitions of the concepts involved.

Basically it's an exhaustive algorithm. Example: 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 reminders.

Basic Algorithm:

```
c ← first(P)
while c != NULL do
  if valid(P, c) then output(P, c)
  c ← next (P, c)
end while
```

Here P is the problem and c is the solution instance. The notion that algorithm captures is that it generate a next solution instance for the given problem and if it is valid, then outputs it.

Brute force is basically a combinatorial explosion.

Examples:

1. Divisor of a number problem: If n has 16 decimal digits, then it takes atleast 10^{15} instructions to achieve the task. When n is 16 bit natural number then it has 19 decimal digits on average and search will take around 10 years
2. Arrangement problem: for 10 letters we have 10! = 3,628,800 candidates, generate and test can happen in less than one second. With 11 letters we have 11! = 39,916,800 which is a 1000% increase. With 20 letters we would need 10 years for the operation execution.

How do we speed up?

1. We can try to **reduce the search space**. For example, for the problem to find all integers between 1 and 1000000 that are evenly divisible by 417, naïve brute force would generate all and test one by one. A better approach is to start with 417 and repeatedly add 417 until the number exceeds 1000000. This would include 1000000/417 steps and no tests
2. We can **re-order the search space**. We could test the most promising candidates first. We could ask the question, what is the probability that the next iteration would be a valid solution?
3. Use **alternatives** to brute force. Domain knowledge can be used to develop different design techniques.

2. Decrease and Conquer

It is based on exploiting the relationship between a solution to a given instance of a problem and solution to a smaller instance of the same problem.

Once the relation is established, it can be either exploited by top-down approach using recursion or bottom-up approach without recursion.
The technique has variations:

- Decrease by a constant
- Decrease by a constant factor
- Variable size decrease

Example: for the problem to compute a^n,
Top down approach is:

$f(n) = f(n-1)*a \quad \text{if } n>1$

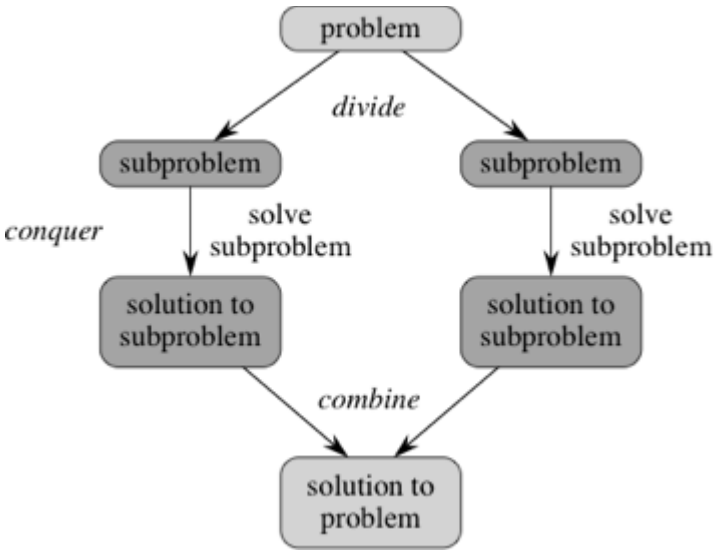
$a \quad \text{if } n = 1$

Bottom-up approach is to multiply a by itself n-1 times.

Though bottom up approach looks like a brute force technique, it is not. We have arrived at the approach using different thought process.

3. Divide and Conquer

The technique is based on based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.



4. Transform and Conquer

A problem instance is transformed to one of the below before the solution is obtained.

- Instance simplification: transform to a simpler or more convenient instance of the same problem
- Representation change: transform to a different representation of the same instance
- Problem reduction: transform to an instance of a different problem for which an algorithm is already available

5. Dynamic Programming

Dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler sub-problems, solving each of those sub-problems just once, and storing their solutions. The next time the same sub-problem occurs, instead of re-computing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a modest expenditure in storage space.

The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem.

6. Greedy Technique

A greedy algorithm is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

In general, greedy algorithms have five components:

- A candidate set, from which a solution is created
- A selection function, which chooses the best candidate to be added to the solution
- A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
- An objective function, which assigns a value to a solution, or a partial solution, and
- A solution function, which will indicate when we have discovered a complete solution

7. Space and time trade-off

A space–time or time–memory trade-off is a case where an algorithm or program trades increased space usage with decreased time. Here, space refers to the data storage consumed in performing a given task (RAM, HDD, etc), and time refers to the time consumed in performing a given task (computation time or response time).

The most common situation is an algorithm involving a lookup table: an implementation can include the entire table, which reduces computing time, but increases the amount of memory needed, or it can compute table entries as needed, increasing computing time, but reducing memory requirements.

8. Randomized Algorithms

A randomized algorithm is an algorithm that employs a degree of randomness as part of its logic. The algorithm typically uses uniformly random bits as an auxiliary input to guide its behaviour, in the hope of achieving good performance in the "average case" over all possible choices of random bits. Formally, the algorithm's performance will be a random variable determined by the random bits; thus either the running time, or the output (or both) are random variables.

Some of the examples to look for are: Bloom Filter, Skip List, Random Binary trees etc.

9. Backtracking

Backtracking is a general algorithm for finding all (or some) solutions to some computational problem, that incrementally builds candidates to the solutions, and abandons each partial candidate ("backtracks") as soon as it determines that candidate cannot possibly be completed to a valid solution.

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

Example: N Queen’s Problem

~*~*~*~*~*~*~*~*~*~