ALGORITHMIC TECHNIQUES

Divide and Conquer

Generalization Step

Solve(P)→Solve(P,Start,End)

Algorithm

```
divide_conquer(input I)
  if (input is small enough) - solve directly
  // THE DIVIDE STEP
  divide I into parts I1, I2,...
  call divide_conquer(I1)
  call divide_conquer(I2)
  ...
  // THE MERGE STEP
  Merge subsolutions
```

Recurrence Relations

Based on how you divide (number of subproblems, and size of each subproblem) and how long it takes to merge the subsolutions, the recurrence relation is established, which then determines the overall running time of your solution. Examples:

- $T(n) = 2\ T(n/2) + cn \Rightarrow O(n \log n)$ [Two subproblems of half the original size, linear time in dividing and merging]
- $T(n) = T(9n/10) + cn \Rightarrow O(n)$
- $T(n) = 8\ T(n/2) + cn \Rightarrow O(n^3)$
- $T(n) = 2\ T(n/2) + cn^2 \Rightarrow O(n^2)$

Greedy Method

Build a complete solution by making a sequence of "best selection steps".

Works well if the problem exhibits "greedy choice property", that is, globally optimal solution can be arrived at by making a locally optimal choice. ***For example***: An optimal minimum spanning tree *can* be built by selecting the smallest weighted "eligible" edge.

Dynamic Programming

Compute solution bottom up by combining subsolutions. Store the results of subsolutions to avoid recomputation of subproblems.

Think of DP when you see (i) optimal substructure and (ii) overlapping subproblems.

Template (Mnemonic: NORA)

1. Develop a **N**otation that can express a subsolution for the given problem: "Say *s(P,i)* represents the solution to problem instance *P*, assuming we start at *i*."
2. Check if the **O**ptimal Substructure (Principle of Optimality) holds.
3. Develop a **R**ecurrence relation that allows building a solution from the subsolutions.
4. Develop the bottom up **A**lgorithm.

For example: Given *n* real (positive/negative) numbers *A(1) ... A(n)*, determine a contiguous subsequence *A(i) ... A(j)* for which the sum of elements in the subsequence is maximized.

Branch and Bound

More general technique, can be used to solve many optimization problems, even those that are NP-complete.

Download again? **https://is.gd/q4uANh**

Basic Template

1. Describe the solution space as a graph
2. Do a Breadth First Search
3. Evaluate nodes for upper and lower bounds
4. Branch into the "best" node
5. Prune nodes via elimination rules
6. Terminate if performance goals met

How to develop bounds

Branch and Bound depends upon being able to "bound" nodes effectively in the search tree.

For a minimization problem: Find a lower bound by making some observation that cost can be no lower than value "x", usually by **relaxing** the problem so that a simpler approach (say greedy) might work, even though the solution may not be a feasible solution to the original problem.
Find an upper bound by using a known solution.

For maximization problem: Lower bound is any feasible solution. Upper bound is theoretical.

Termination criteria: B&B is used for ***hard*** problems, so keep an eye on target performance of the solution. E.g., if it is OK to find a solution that costs at most 5% more than optimal, then use that to find terminating conditions.

Backtracking

A problem like Sudoku, or knapsack can be solved by building partial solutions, discarding the ones that don't work and backtracking in the solution search tree to try a different solution. Depending on the discarding mechanism, this can be very efficient technique!

Problem Complexity and Classes

NP-Completeness

We consider two classes of problems: P and NP. Class P consists of problems that can be solved in polynomial time. Class NP consists of problems, for which you can verify a given solution in polynomial time. An NP-Complete problem is a problem which can be considered a "central" (or complete) problem for all problems in class NP. Class "NPC" is the class of all NP-complete problems. **Essentially, class NPC consists of hard problems for which polynomial time algorithms are not known to exist.** The reason we try to find if a problem is NP-complete is to determine if it is hard. If we prove that a given problem is NP-complete, we can convince others, if required, that the problem is hard, and our efforts are more justified in other ways of tackling NP-Completeness.

A Fascinating Open Problem

If you (or someone else) finds a polynomial time algorithm for an NP-complete problem, that is equivalent of finding a polynomial time algorithm for **ALL** problems in NP, and will also imply that classes P and NP are in fact the same. **THAT WILL BE HUGE!** You will be rich! $$$$. Similarly, if you find that a particular NP-complete problem cannot be solved in polynomial time using our current computation models – that would prove that these classes are in fact distinct. That would also be huge!

How to prove that problem X is NP-complete?

To prove that problem X is NP-complete, we have to start with a known NP-complete problem, and reduce that to our problem X.

Mnemonic: **SoX – ReStoX** (SAT Outside the box – X Inside – Reduce SAT to X).

Tackling NP-Completeness

Strategy 1 – "Context": Look for simplifications in data and context that render the problem solvable in polynomial time.

Strategy 2 – "Find low Exponent": Look for improvements in running time which make the exponential "bearable."

Strategy 3 – "Approximate and Refine": Understand performance bounds that are acceptable practically, and use approximation algorithms.

Beyond class NP

Some problems are provably *undecidable*, irrespective of the time we spend on it.

Applying All This

Tips and Tricks

1. Remember the time when you brought that hamster home. When you named it Pete, your family knew it was your pet, and it was going to stay. Naming is your friend. If you can name it, you can own it. Start with a good notation: "Let S(n,m) represent ...."

Download again? **https://is.gd/q4uANh**

2. Avoid premature optimizations, something that only work on some inputs.
3. Generalization can help you in using divide and conquer and recursive solutions.

6212 Solution Process

Phase 1 – Define Your Problem

1. Draw a box, with inputs and outputs
2. **(Important)** Name the box *(Hint: Generic word like "solver" is not a name.)*

Phase 2 – Solve Your Problem

1. What is the brute force time complexity?
2. Try different algorithmic design techniques and different data structures if needed.
3. If the problem appears very hard, is it NP-complete? Can we prove that, and tackle its NP-completeness?

Learning to Learn

Don't forget the Ebbinghaus forgetting curve. If you review something 4 times, say once shortly after learning, once after a day, once after a week and another time after a month, there is a very good chance you will remember it forever.

Your chance of learning and remembering something increases if you explain that to someone. Volunteering to give a presentation on a topic is an excellent way to learn it even further.

(Last, but not Least!) People Matter

Don't forget that a big part of graduate education is building your network of friends.