Problem Solving Paradigms

OLIVIER BOURGEOIS

Introduction

Problem solving paradigms are different tools in a toolbox.

Wouldn’t want to “hammer” a nail with a screwdriver.

Helps identify patterns in similar problem statements.

Naïve solution is not always efficient (and sometimes near-impossible).

So.. What are the most common problem solving patterns?

Paradigms Outline

Complete Search

- Iterative vs Recursive Complete Search
- Example

Divide and Conquer

- Example

Greedy Algorithms

- Example

Dynamic Programming

- Methods
- Example

Complete Search

AKA BRUTE-FORCE

Complete Search (aka Brute-Force)

Consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

Advantages

- Easy to implement
- Will always find a solution

Disadvantages

- Computational cost proportional to the number of possible candidates

Use When

- The input size is limited
- Problem-specific heuristics that can drastically reduce the pool of candidates
- Faster algorithms would take too much time to implement

Complete Search – Iterative

An iterative complete search approach consists at looping over every possible candidate.

Usually used when there’s not more than a few nested loop.

E.g. Outputs every unique pair of numbers in the array [0, 1, 2 … N]

```
for (int i = 0; i < N - 1; i++) {
  for (int j = i + 1; j < N; j++) {
    System.out.println("(" + i + "," + j + ")")
  }
}
```

Complete Search – Recursive

A recursive complete search consists at recursively going through every candidate.

Usually used when there would be a variable amount of nested loops.

E.g. compute all permutations of a string b

```
void permutations(String a, String b) {
  String s = "";

  if (b.length() == 1) s += a + b + " ";
  else
    for (int i = 0; i < b.length(); i++)
      s += permutations(a + b.substring(i, i + 1), b.substring(i + 1, b.length()));

  return s;
}

permutations("", "abcd");
```

Divide and Conquer

WITH EXAMPLES

Divide and Conquer

The divide and conquer paradigm consists of recursively breaking down a problem into two or more sub-problems of the same type until they become simple enough to be solved directly.

Advantage

- Faster than a naïve complete search approach

Disadvantage

- Computational growth can be tricky to calculate
- Requires a good understanding of a problem before being able to classify as DnC

Use When

- It’s clear that the problem can be generalized into smaller subsets

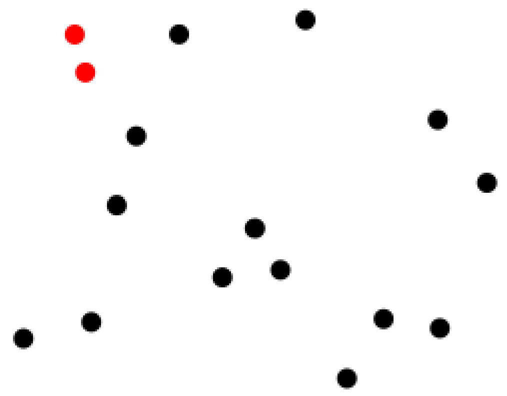
Divide and Conquer – Example

E.g. the closest pair of point problem consists of finding the pair of points in a 2D plane whose length separating them is at a minimum.

Naïve solution would be to compute all pair-wise distances in $O(n^2)$.

Can do $O(n \log n)$ with the following DnC algorithm:

- Sort points according to their x-coordinates
- Split the set of points into two equal-sized subsets by a vertical line
- Solve the problem recursively in the left and right subsets, yielding d_{Lmin} and d_{Rmin}
- Find d_{LRmin} using one point from the left half and one point from the right half
- The final answer is the minimum of d_{Lmin}, d_{Rmin} and d_{LRmin}



Greedy Algorithms

WITH EXAMPLES

Greedy Algorithms

The greedy algorithm paradigm consists at making the locally optimal choice during the entire problem set while hoping to arrive at a globally optimal solution.

Advantage

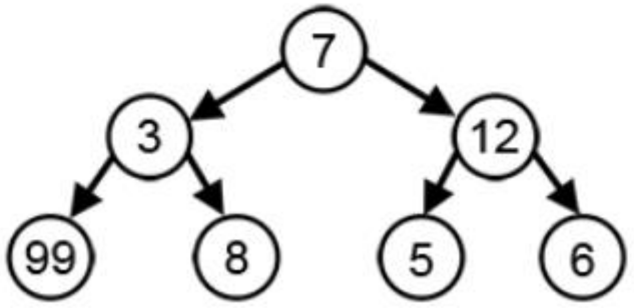
- Usually easy to implement

Disadvantage

- Depending on the problem, can arrive at a local maximum
- Need a good choice of heuristics to arrive at an optimum solution

Use When

- The problem is typically known as requiring a greedy solution



e.g. largest sum

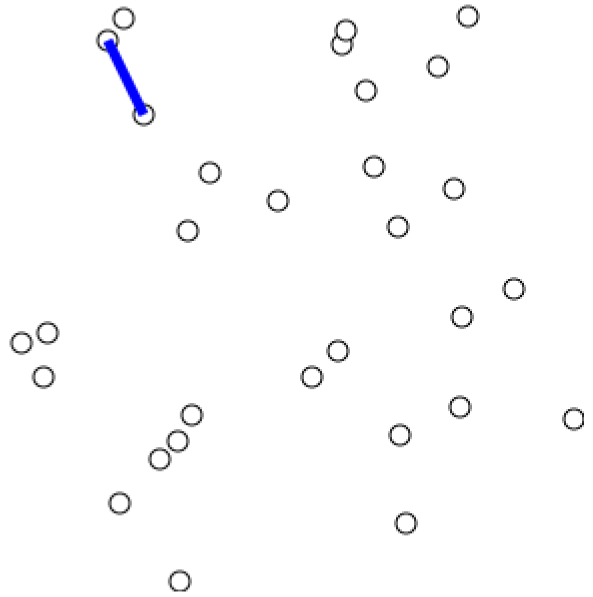
Greedy Algorithms – Example

E.g. Prim's algorithm for generating minimum spanning trees.

Most algorithms involving MST's are greedy.

Followed is Prim's implementation:

- Initialize a tree with a single vertex, chosen arbitrarily
- Of the edges that connects the tree to vertices not yet in the tree, find the minimum-weight edge; Add it to the tree
- Repeat step 2 until all vertices are in the tree



Dynamic Programming

WITH EXAMPLES

Dynamic Programming

Dynamic programming consists of breaking down a problem into many sub-problems and making sure that each sub-problem is solved a maximum of one time.

Advantage

- Very fast

Disadvantage

- Can be tricky to figure out how to break the problem in an elegant way

Use When

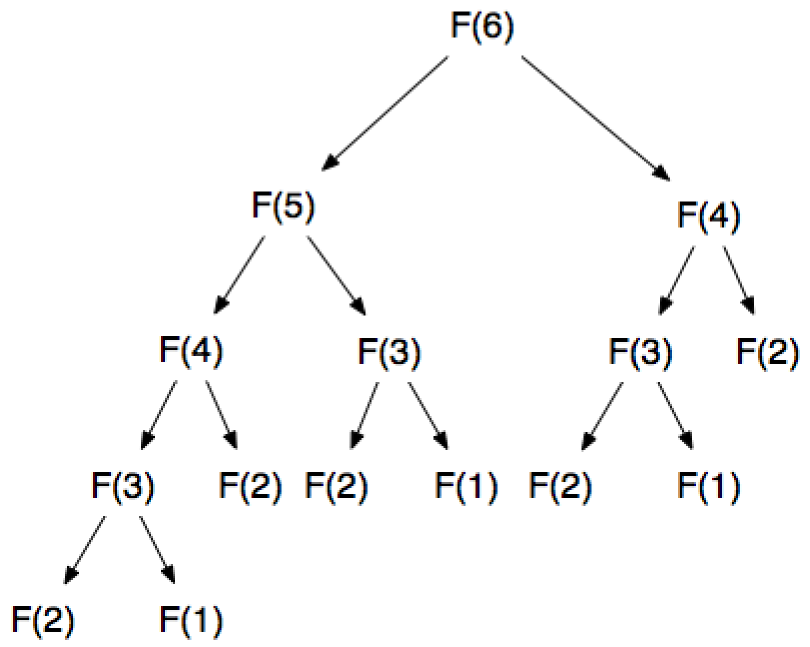
- The problem requires you to solve the same sub-problem(s) over and over

Dynamic Programming – Why do it?

Sometimes a problem is not possible via greedy algorithms because of local maxima, and using complete search leads to an explosive growth of sub-problems to solve.

For instance, let’s look at a naïve implementation of a Fibonacci sequence.

```
static long fib(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fib(n-1) + fib(n-2);
}
```



Dynamic Programming – Methods

Bottom-up (tabulation) vs Top-down (memoization)

There are two main methods of solving dynamic programming problems.

The first method is iterative generation which consists at generating every sub-problem starting from the base case and moving towards the end-problem.

- Usually very fast
- May be computing unused sub-problems

The second method is to recursively solve sub-problems starting from the end-problem and going down, caching results as the recursion happens.

- Solves only the sub-problems that are needed
- Depth of recursion could be huge; could lead to stack overflow or a big overhead

Dynamic Programming – Fibonacci

Here is the Fibonacci sequence using an iterative (tabulation) strategy.

```
long fibonacci[] = new long[N];

fibonacci[0] = 0;
fibonacci[1] = 1;

for (int i = 2; i < N; i++) {
  fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
}
```

Dynamic Programming – Fibonacci

Here is the Fibonacci sequence using a recursive (memoization) strategy.

```
long fibonacci[] = new long[N];

static long fib(int n) {
  if (fibonacci[n] != 0) return fibonacci[n];
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibonacci[n] = fibonacci[n-1] + fibonacci[n-2];
}
```

Dynamic Programming – Card Magic

Johanna knows mind reading magic, or so she says. Her new trick consists of lining up N decks of cards, each deck having K cards numbered from 1 to K. She asks you to think of a number T between 1 and N·K and to focus your thoughts on it. Then, she carefully picks one card from each of the N decks. Magically, the sum of the numbers on the N picked cards is exactly the number that you were thinking of! You suspect it might be a trick she pulls on many people, and that she just picks the cards at random and happened to get it right with you just by chance. You start wondering just how large that chance was. Compute the number of ways to pick one card from each deck while having the correct sum.

Input is N [1 - 100], K [1 - 50], T [1 - N·K]

Reference: https://open.kattis.com/problems/cardmagic

Dynamic Programming – Card Magic

Naïve implementation would be to try solving this using combinatorics.

Could be hard and/or long to compute the final result.

Instead, we can use dynamic programming coupled with an iterative approach.

- Base case: for one deck, there is exactly 1 solution for each of T between 1 and K.
- Given some number of deck n, we can fill in the row using results from number of deck n-1:
  - Initialize table[n][t] to zero.
  - Iterate over each possible card for the last card. Let’s call this iterator i.
  - For each of these, look up in the table the odds of arriving at a sum of k-i and add it to table[n][t].
- Repeat for as many rows (number of decks) as needed.

Dynamic Programming – Card Magic

```
int[][] table = new int[N+1][N*K+1];
for (int i = 1; i < K+1; i++) {
  table[1][i] = 1;
}

for (int n = 2; n < N+1; n++) {
  for (int t = 2; t < n*K+1; t++) {
    int sum = 0;
    for (int k = 1; k < K+1; k++) {
      if (t-k > 0) sum += table[n-1][t-k];
      if (sum > 500_000_000) sum %= 1_000_000_009;
    }
    table[n][t] = sum;
  }
}

System.out.println(table[N][T]);
```

References

Wikimedia Commons

Open Kattis

Competitive Programming 3, by Steven and Felix Halim