Workshop #2 Competitive Programming: Problem Solving Paradigms

Presentation: Imad Dodin
Slides: Muhammad Huzaifa Elahi

>Compete McGill_

Problem Solving Paradigms

- A paradigm
  - A method of approaching and solving a specific problem
  - A guiding methodology to use when approaching a particular type of problem
  - Provide you the right tool for the job
  - Crucial for competitive programming competitions with repeated problem types

Importance of these Paradigms

- Competitive programming is all about **efficiency** and **optimization**
- Simply finding a solution is not sufficient - we **cannot brute force our way to victory!**
- By using the right paradigm, we get the **lowest possible Big O runtime -> key to victory!**

>Compete McGill_

Problem Solving Paradigms

- Types of useful paradigms: we will discuss 4 different types
    a. Complete Search (a.k.a Brute Force),
    b. Divide and Conquer
    c. The Greedy approach
    d. Dynamic Programming

1. Complete Search a.k.a Brute Force (1)

- A method for solving a problem by traversing the entire search space to obtain the solution.
- Can **prune**:
  - Not explore parts of search space if these parts have no possibility of having the solution
- When?
  - 1. there is clearly no other algorithm available
    - e.g. the task of enumerating all permutations of {0, 1, 2, . . .,N − 1} clearly requires O(N!) operations)
  - 2. when better algorithms exist, but are overkill : input size is small
    - e.g. the problem with input N < 100

1. Complete Search a.k.a Brute Force (2)

Example:

- Given $6 < k < 13$ integers, enumerate all possible subsets of size 6 of these integers in sorted order.

Answer:

```
for (int i = 0; i < k; i++)                          // input: k sorted integers
  scanf("%d", &S[i]);
for (int a = 0    ; a < k - 5; a++)                          // six nested loops!
  for (int b = a + 1; b < k - 4; b++)
    for (int c = b + 1; c < k - 3; c++)
      for (int d = c + 1; d < k - 2; d++)
        for (int e = d + 1; e < k - 1; e++)
          for (int f = e + 1; f < k    ; f++)
            printf("%d %d %d %d %d %d\n",S[a],S[b],S[c],S[d],S[e],S[f]);
```

Note: even in largest test case, k = 12, the six nested loops will produce 12C6 = 924 lines: This is **small!**

Tips for Complete Search (1)

- Tip 1: Filtering versus Generating
    - Programs that examine lots of possible solutions and choose the ones that are correct are called **‘filters’**
    - Programs that build up solutions and instantly prune invalid partial solutions are called **‘generators’**
    - **GENERALLY**:
        - ‘generator’ programs are <u>easier to implement</u> when written recursively as it gives us greater flexibility for pruning the search space.
        - filters are easier to code but <u>run slower</u>, given that it is usually far more difficult to prune more of the search space iteratively

>Compete McGill_

Tips for Complete Search (2)

- Tip 2: Prune Infeasible/Inferior Search Space Early
    - Imagine your programming building up a set of potential solutions
    - Finding the entire solution requires a search of these potential solutions
    - As soon as we find that one of these potential solutions cannot be the actual solution
        - PRUNE THE SEARCH!
        - We do not expend computational energy to further check and iterate this potential solution

Tips for Complete Search (3)

- Tip 3: Utilize Symmetries
    - Let us assume for a problem there are 92 solutions but there are only 12 unique solutions as there are rotational and line symmetries in the problem.
        - Take advantage!
        - Generate only 12 unique solutions!
        - If required: generate the whole 92 by rotating and reflecting these 12 unique solutions
- NOTE: Symmetries can sometimes complicate code: only use if there is an obvious benefit (simplicity or runtime speed)

Tips for Complete Search (4)

- Tip 4: Code Optimization
  - Use the faster ArrayList (and StringBuilder) rather than Vector (and StringBuffer).
  - Access a 2D array row by row rather than column by column - arrays are stored in row by row order in memory.
  - Declare most data structures once globally with enough memory to deal with the largest input
    - Avoids passing data structures as arguments
  - Array access in (nested) loops can be slow. If you frequently access the value of A[i] (without changing it) in (nested) loops,
    - Use a local variable temp = A[i] and works with temp instead.

2. Divide and Conquer

- Divide and Conquer (D&C) is a problem-solving paradigm in which:
  - A problem is made simpler by ‘dividing’ it into smaller parts and then conquering each part.
- The steps:

  1. Divide the original problem into sub-problems—usually by half or nearly half,

  2. Find (sub)-solutions for each of these sub-problems—which are now easier,

  3. If needed, combine the sub-solutions to get a complete solution for the main problem.

- Familiar Examples:
  - Quick Sort
  - Merge Sort
  - Heap Sort
  - Binary Search

>Compete McGill_

Binary Search

Example: Binary Search

```
// arr[] is array, l is left index, r is right index, x is desired entry being searched for
int binarySearch(int arr[], int l, int r, int x)
    {
        if (r >= l) {
            int mid = l + (r - l) / 2;
            if (arr[mid] == x)              // if element is at mid point
                return mid;
            if (arr[mid] > x)               // if element is smaller than midpoint, must be in left subarray
                return binarySearch(arr, l, mid - 1, x);
            return binarySearch(arr, mid + 1, r, x);  // else in right subarray
        }
        return -1;    // x is not in the array
    }
```

3. Greedy

- A problem solving paradigm wherein an algorithm is greedy if:
  - it makes the locally optimal choice at each step and eventually reaches the globally optimal solution.
- In some cases, greedy works—the solution is short and runs efficiently. For many others, however, it does not.
- A problem must exhibit these two properties in order for a greedy algorithm to work:
  - **1. It has optimal sub-structures.**
    - Optimal solution to the problem contains optimal solutions to the sub-problems.
  - **2. It has the greedy property** (difficult to prove in contest environment!).
    - If we make a choice that seems like the best at the moment and proceed to solve the remaining subproblem, we reach the **optimal solution**.
    - We will never have to reconsider our previous choices.

Dragons of Loowater (1)

- Example:
  - There are n dragon heads and m knights (1 ≤ n,m ≤ 20000).
  - Each dragon head has a diameter and each knight has a height.
  - A dragon head with diameter D can be chopped off by a knight with height H if D ≤ H.
  - A knight can only chop off one dragon head.
  - Given a list of diameters of the dragon heads and a list of heights of the knights, is it possible to chop off all the dragon heads?
  - If yes, what is the minimum total height of the knights used to chop off the dragons' heads?

Dragons of Loowater (2)

- This problem can be solved greedily:
  - Each dragon head should be chopped by a knight with the shortest height that is at least as tall as the diameter of the dragon's head.
  - However, the input is given in an arbitrary order.
  - If we sort both the list of dragon head diameters and knight heights in O(n log n + mlogm), we can use the O(min(n,m)) scan below to determine the answer.
  - NOTE: this is one of many questions where sorting the input can help produce the greedy strategy.

```
gold = d = k = 0;                // array dragon + knight are sorted in non decreasing order
while (d < n && k < m) {         // still have dragon heads or knights
      while (dragon[d] > knight[k] && k < m) k++;        // find the required knight
      if (k == m) break;         // no knight can kill this dragon head, doomed :S
      gold += knight[k];         // the king pay this amount of gold
      d++; k++;                  // next dragon head and knight please
}


if (d == n) printf("%d\n", gold);         // all dragon heads are chopped
else printf("Loowater is doomed!\n");
```

>Compete McGill_

Dynamic Programming - DP (1)

- Dynamic Programming: the most challenging problem-solving technique among the four paradigms
    - Lots of recursion and recurrence relations!
- The key?
    - Determine problem states
    - Determine the relationships/transitions between current problems and their sub-problems.

Dynamic Programming - DP (2)

- When?
    - Primarily : solve optimization problems and counting problems.
    - If you encounter a problem that says :
        - “minimize this”
        - “maximize that”
        - “count the ways to do that”
- Most DP problems in contests ask for the optimal value and not the optimal solution itself
    - Makes problem easier to solve by removing the need to backtrack and produce the solution.
    - **However**, some harder DP problems also require the optimal solution
- Vocab:
    - **state** - a unique subproblem

Top-Down DP

- Top-Down DP:
  - Start from biggest subproblems (top)
  - Recursively go downwards (down) till you reach the smallest subproblems
  - Solve these and chain recursion back up
- Top-Down DP solution:
  - 1. Initialize DP ‘memo’ table with dummy values e.g. ‘-1’.
    - Table dimensions correspond to the problem states
  - 2. At the start of recursive function, check if this state has been computed before.
    - (a) If it has
      - Return the value from the DP memo table, **O(1)**.
    - (b) If it has not been computed
      - Perform the computation once, **O(?) - varies based on task**
      - Store this computed value in table, **O(1)**
      - Future calls to this sub-problem (state) return answer immediately, **O(1)**
      - **NOTE: The process is called Memoization**

>Compete McGill_

Bottom-up DP

- Bottom-up DP:
  - Solve subcases and build solution “upwards” using existing small problem solutions to solve big problems
- Bottom-up DP solution :
  - 1. Determine the required set of parameters that uniquely describe a subproblem (a state).
  - 2. If there are N parameters required to represent the states, prepare an N dimensional DP table, with one entry per state.
    - In bottom-up DP, we only need to initialize some cells of the DP table with known initial values (the base cases).
  - 3. Now, with the base-case cells/states in the DP table already filled, determine the cells/states that can be filled next (the transitions).
  - 4. Repeat this process until the DP table is complete.
    - Usually accomplished through iterations (loops)

DP - Top Down vs Bottom Up

Top-Down	Bottom-Up
Pros: 1. It is a natural transformation from the normal Complete Search recursion 2. Computes the sub-problems only when necessary (sometimes this is faster)	Pros: 1. Faster if many sub-problems are revisited as there is no overhead from recursive calls 2. Can save memory space with the 'space saving trick' technique
Cons: 1. Slower if many sub-problems are revisited due to function call overhead (this is not usually penalized in programming contests) 2. If there are M states, an $O(M)$ table size is required, which can lead to MLE for some harder problems (except if we use the trick in Section 8.3.4)	Cons: 1. For programmers who are inclined to recursion, this style may not be intuitive 2. If there are M states, bottom-up DP visits and fills the value of *all* these M states