Nonstandard Dynamic Programming

Junhua Chen 11/12/2021

Table Of Contents

Abstract:
Most DP questions involve linear states. Today we shall cover DP techniques and ideas that are not particularly well known but are applicable to many problems, or in particular, subtasks of problems.

SECTION I: Range DP

(1) What is it and what's hard about it

(2) The extremal idea for designing recursions

(3) Two problems: Polygon Game, Array Deletion

Section II: Exponential / Bitmask DP

(1) Basic bitmask DP via problem Oddjobs

(2) Basic Frontier DP via problem Wet Towels

(3) Meet in the middle via problem XOR path

Section III: Lexicographic DP

(1) Toy problem: permutation

(2) Turning it into counting problem

(3) Getting $k-th$ element

Range DP

- Common problems you've often seen:
  - *Here's an array of numbers, here's an operation, maximise score*
  - *Here's a bunch of things superimposed over an array, operation, max score*

- Range DP often offers solutions to these problems
- The idea is simple: maintain what the answer is in each subarray
  - i.e $dp[l][r] = best\ result\ using\ A_l \dots A_r$
  - Generally, the state is easy (but people often forget about range dp) but the recursion is hard
  - Often easy to get recursion wrong outright, must be careful

- $N \leq 300$ is often a dead giveaway

The extremal principle

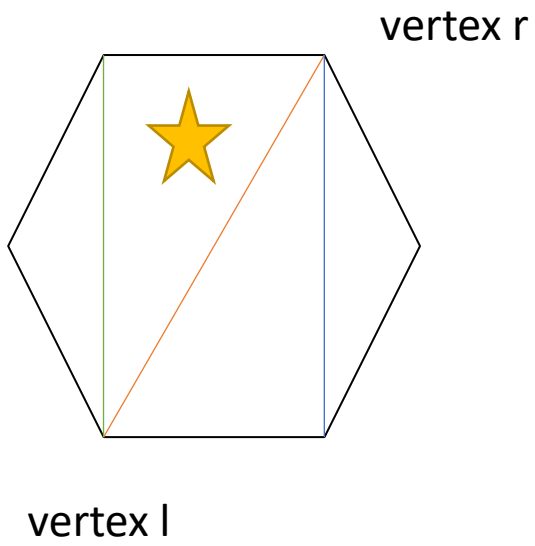
- Common idea used in many problems, but especially range dp
- Basically the general idea is that in a set of things there must be one with some special property
  - E.g 1: In every subarray there is a biggest element
  - E.g 2: If you sequentially erase every element of a subarray there must be one that is deleted last
- How this idea is used is best seen by example
- *Use this to help split your problem into independent subproblems*

Example problem 1: Polygon Game

- TL DR: Given polygon with vertex weights triangulate it to maximise sum of products of edges which are connected
- Aim for $O(N^3)$, lets think for two minutes.
- Any ideas? Why might we be motivated to think about range dp?

Solution

- $dp[l][r] = Best\ Triangulation\ Using\ Subpolygon\ vertices\ l\ \ldots r$
- $dp[i][j] = 0, |i - j| \leq 1$
- Our extremal idea is that in any triangulation the edge connecting vertices l, r must be part of some triangle



Solution (cont'd)

- We got our extremal observation now and we'll use it
- It is easy to enumerate the possibilities for the triangle by enumerating all possible choices for the 3rd vertex i
  - $dp[l][r] = \max_{l<i<r} dp[l][i] + dp[i][r] + (V[l] + V[r]) \times V[i]$
  - That's it! We just considered a very simple extremal property (what is vertex l doing and basically recursion falls out)
  - Lets look at some code!

Fun observation because I like maths (optional, read yourself)

Observation: Every triangulation of a polygon contains an "short edge" i.e joining $v_k \leftrightarrow v_{k+2}$ for some k

Proof: ~~Holds for n=5~~ There are $n-2$ triangles in any triangulation and n sides so it must be case that there exist two pairs of sides belonging in same triangle in triangulation by pigeonhole principle. Those two sides must be adjacent. This gives us *two* short edges as clearly we can't have 3 sides in a triangle for $n \neq 3$ Q.E.D

Exercise: Devise a different proof by considering graph created by the triangle's adjacency

Problem 2: Delete

Statement:

You have array $A[1] \ldots A[2N]$ and the operation of deleting two adjacent elements of the array. You would be deleting every element of the array and your score is the sum of the products of the pairs of elements you delete.

e.g For the sequence of deletions 1 2 3 4 −> 1 4 −> ∅ you score 6+4=10

Maximise your score ($N \leq 80$)

Solution

- Once again state is what you'd expect: $dp[l][r] = ans\ for\ A[l\ \ldots r]$ and of course $dp[i][i + even] = -\infty, dp[i][i-1] = 0$
- Extremal observation: In every set of deletions one of the deletions happens last (e.g in previous example 1 4 is deleted last)
  - Idea: Choose elements in range which are deleted last, they split the range into *independent subproblems*
  - Thus we get $dp[l][r] = \max_{l \le i < j \le r} dp[l][i-1] + dp[i+1][j-1] + dp[j+1][r] + A[i]A[j]$ for easy $O(N^4)$
  - Implementation is easy

- Notice the way I enumerate the states in the two outer for loops. At least I feel like its quite a nice way to do range dp

```
for (int len=2; len < N; len++) {
    //assume dp[l][l-1] = 0 for all l
    for (int l=1; l <= N; l+=2) {
        int r = l+len-1;
        dp[l][r] = -INF;
        for (int i=l; i<r; i++) {
            for (int j=i+1; j<=r; j+=2) {
                dp[l][r] = max(dp[l][r], dp[l][i-1]+dp[i+1][j-1]+
                                         dp[j+1][r] + A[i]*A[j]);
            }
        }
    }
}
```

General sanity checks for the recursion

- Will optimal answers always be considered?
- Are the subproblems created independent?
- Are all possibilities considered by DP legal?

- These are all important considerations to thinking why a recursion might be correct or incorrect

SECTION I: Range DP

(1) What is it and what's hard about it

(2) The extremal idea for designing recursions

(3) Two problems: Polygon Game, Array Deletion

Section II: Exponential / Bitmask DP

(1) Basic bitmask DP via problem Oddjobs

(2) Basic Frontier DP via problem Wet Towels

(3) Meet in the middle via problem XOR path

Section III: Lexicographic DP

(1) Toy problem: permutation

(2) Turning it into counting problem

(3) Getting $k-th$ element

Basic bitmask DP

- Maintaining DP where one of the states is subsets of a set can help tackle many standard NP−hard problems
- Great for solving subtasks even if it can’t solve the entire problem
- Straw Poll: Have we all done bitmask DP before?

Oddjobs (basic bitmask DP)

- TL DR Arrange $N \leq 15$ people to N jobs such that each person does a different job and the total cost of work $\sum_{i=1}^{N} Cost(x, job(x))$ is minimised (cost matrix given)
- While the Hungarian method solves this in a blazingly fast $O(N^3)$ we'll stick with a simpler dynamic programming approach
- $dp[S \subset \{1 \ \ldots N\}]$ is the minimum cost to assign persons $1 \ \ldots |S|$ to jobs in S. Recursion is to assign person $|S|$ to 1 of the jobs.
- Subsets are maintained as bitmasks

Implementation

```
int N, A[20][20], dp[35000];
//read costs into A here

for (int i = 1; i < 1<<N; i++) {
    //get number of students
    int upto = 0;
    for (int j = 0; j < N; j++) {
        if (!((1 << j) & i)) upto++;
    }

    //the recursion
    dp[i] = 1e9;
    for (int j = 0; j < N; j++) {
        if ((1 << j) & i) {
            dp[i] = min(dp[i], dp[i - (1 << j)] + A[upto][j]);
        }
    }
}
```

Bit hacking Cheatsheet for your convinience

http://orac.amt.edu.au/notes/Binary.pdf

Here are also some fancier operations which are really only useful if you want absolute speed except for the first one.

Exercise: Implement ctz from the other operations

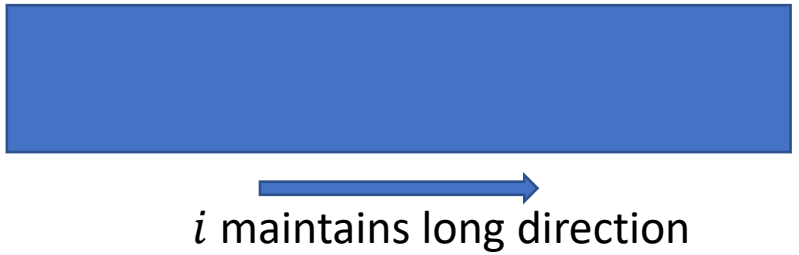
Fancy Operation	What it does	Example
x&(−x) (from fenwick)	Least significant bit value	12&(−12)=4
__builtin_popcount __builtin_popcountll	Counts number of 1 bits in integer (for 32,64 bit)	__builtin_popcount(12)=2
__builtin_ctz __builtin_ctzll	Counts number of trailing zeros in integer	__builtin_ctz(16)=4

Basic frontier DP

- Problems where the dp state is the “complete” information about the current part of solution being constructed
- We illustrate this through problem Wet Towels (https://orac2.info/problem/seln07towels/) (TL DR tile a weighted $7 \times N$ grid with dominos such that the sum of the domino’s weight differences are minimised)
- The difficulty is often in implementation: The recursion (we call them state transitions here) are often very complex and it is one of the programmer’s main tasks to characterize them efficiently

Solution and implementation tricks

- Clearly we should progressively tile the grid along the long axis, keeping the complete profile of the length 7 axis. So lets define our state as:
  - $dp[state \subset \{1 \ ... 7\}][i] =$ best cost to tile subgrid (state, i) where:
    - Columns $i+1 \ ... N$ is completely untiled
    - Columns $1 \ ... i-1$ is completely tiled
    - Square (x, i) is tiled iff $x \in state$



Aside: We can solve this question by Hungarian algorithm too by reducing to min weight matching

The state transition

- Lets say there's 7 rows and N columns. At the state we can characterise transitions as either

(1) Placing a vertical domino in column i

(2) Placing a horizontal domino in $(x, i), (x + 1, i)$ for all $i \in state$

```cpp
#include <iostream>
#define LL long long
#define MAXN 10005

using namespace std;
int N,val[MAXN][7];
LL dp[MAXN][130]; //mask representing state s (1 means placed), i
bool done[MAXN][130];

LL f(int n, int m) {
    if (n==N and m==0) {return(0);}
    else if (n==N) {return(1LL<<60);}
    else if (m==127) {return(f(n+1,0));}
    else if (!done[n][m]) {
        done[n][m]=true;
        dp[n][m]=1LL<<60;
        LL s=0; //the cost to put horizontal dominos everywhere
        for (int i=0; i<6; i++) {
            if (!((m&(1<<i)) or (m&(2<<i)))) {
                //can place vertical domino at i,i+1 & try placing it
                dp[n][m]=min(dp[n][m],f(n,m+3*(1<<i))+abs(val[n][i]-val[n][i+1]));
            }
        }
        for (int i=0; i<7; i++) {
            s+=(m&(1<<i)) ? 0 : abs(val[n+1][i]-val[n][i]);
        }

        dp[n][m]=min(dp[n][m],f(n+1,127-m)+s); //try placing all horizontal dominos
    }
    return(dp[n][m]);
}
```

Meet in the middle

- It is a technique for optimising brute force approaches
- The main idea is to split an brute force of size N into two brute forces of size $\frac{N}{2}$ which are combined by a semi−bruteforce merge
- The complexity would thus go from say $O(2^N)$ to $O(2^{\frac{N}{2}} f(n))$ where merging one element takes $f(n)$
- You can think of it as a halfway house between D&C and sqrt decomp, in that your splitting in half but the recursion is only one layer deep
- Thus if you can merge answers quickly it can be viable approach. It is especially good for counting

Example problem: XOR paths (codeforces)

- https://codeforces.com/contest/1006/problem/F
- One minute to think

Useful combinatorial Arguments & estimates

(1) The number of down−right paths from (0,0) to (N,M) is $\binom{N+M}{M}$

Proof: The paths contain N down operations and M right operations. Thus for each binary string of length N+M with M "0"s there is a unique path. So there are $\binom{N+M}{M}$ paths

(2) The number of down−right paths from (0,0) to $x + y = a$ is 2^a

Proof: Argued in same way as (1)

(3) $\binom{2N}{N} \approx \frac{4^N}{\sqrt{\pi N}}$

Proof: by Stirling's approximation $n! = (1 + O\left(\frac{1}{n}\right))\left(\frac{n}{e}\right)^n \sqrt{2\pi n}$ where as $n \to \infty$

(4) The number of ways to write n as the ordered sum of k nonnegative integers is $\binom{n+k-1}{k}$

Proof: Choose k separators among n+k−1 things (we won't use it here but useful to know)

(5) The number of balanced bracket sequences of length $2n$ is $\frac{1}{n+1}\binom{2n}{n}$ (called Catalan number)

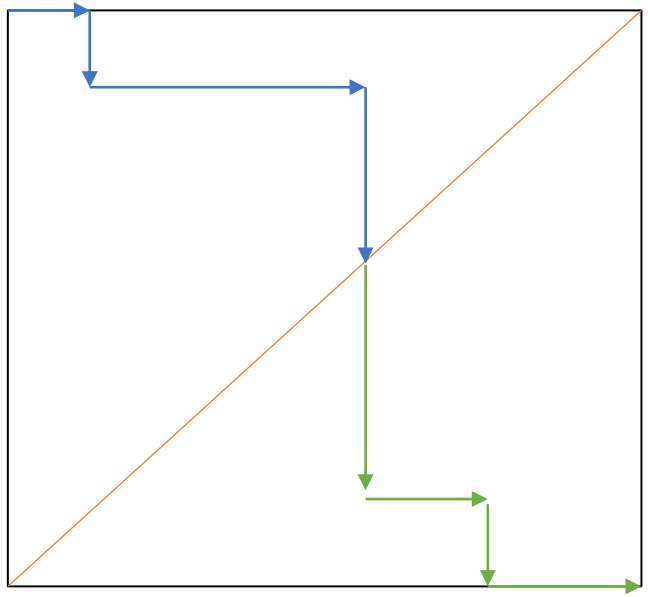
Proof: Setup recursion of form: either split the sequence into two balanced sequences or take out first and last brackets to form another balanced bracket sequences and apply induction (we won't use it here but useful to know)

The brute force and how to improve it

- Lets just take $N = M = 20$ here because why not
- We can of course check all $\binom{40}{20} \approx 1.4 \times 10^{11}$ paths but I don't think C++ runs that fast. So lets use meet in middle
- Generating all paths up to the long diagonal, however, is feasible as there's 2^{20} of them.

Split paths into upper and lower half

- See: meet in middle!



Animation:
https://www.youtube.com/watch?v=xvFZjo5PgG0

Solution outline

GENERATE ALL TOP PATHS //$O(2^N)$

FOR EACH BOTTOM PATH //2^N to loop through

COUNT HOW MANY TOP PATHS ENDING WHERE IT STARTS THAT XOR WITH IT TO K //$O(log(2^N)) = O(N)$ *by map. This is our merge step*

So complexity is $O(2^N N)$ as opposed to bruteforce time of $O(\frac{4^N}{\sqrt{N}})$.

Implementation: A few things to watch

- https://codeforces.com/contest/1006/submission/138309869
- How I only store the upper paths but not the lower one: cuts memory in half (often meet in the middle has tight memory and time limits)
- Tr1::unordered_map is very fast and shaves a fair bit of time off
- Edge cases must be covered

Reading Material for Problem Superheros

Try to solve the problem yourself. However, if you get stuck for more than an hour, read the editorial (Q3 only):

https://drive.google.com/file/d/1pMdKOgGVARDXy3rCW6VT_kL-tHO1lGFV/view?usp=sharing

For implementation, we introduce the following trick for enumerating nonempty subsets of a bitmask n:

```
for (int i=n; i>0; i=(i-1)&n) {
    // i is a subset
}
```

SECTION I: Range DP

(1) What is it and what's hard about it

(2) The extremal idea for designing recursions

(3) Two problems: Polygon Game, Array Deletion

Section II: Exponential / Bitmask DP

(1) Basic bitmask DP via problem Oddjobs

(2) Basic Frontier DP via problem Wet Towels

(3) Meet in the middle via problem XOR path

Section III: Lexicographic DP

(1) Toy problem: permutation

(2) Turning it into counting problem

(3) Getting k−th element

What Lexicographic DP problems look like

- Type 1: Given a sequence with certain properties compute its lexicographic order among all sequences with such property
- Type 2: Find the kth lexicographically least such sequence

In interests of time we will just examine one of the simplest such questions, where sequences is “Permutations”

Problem: Cryptography

- Given a permutation $P_{1\ldots N}$ find its lexicographical order mod $10^9 + 7$
- Definition of lexicographic order: $P \leq_{lex} Q$ iff exist index k that P and Q are same before index k and $Q_k > P_k$

- Wishful thinking: What if we had an oracle that can tell us how many permutations start with certain prefix. Call it $f(P_1, \ldots P_k)$
- Then the number of permutations lexicographically less than P is $\mathrm{n}(P) = \sum_{k=1}^{n} \sum_{j=1}^{P_k - 1} f(P_1, \ldots P_{k-1}, j) = \sum_{k=1}^{n} \#seqs\ diverging\ at\ k$
- So lexicographic order is $n(P) + 1$

Now what?

- The method we explained is very general and is usable on all problems of this class
- The trick to most lexicographic problems is constructing the function f, usually by encoding a prefix as a ‘local’ condition in a dp
- In our case $f(x_1 \ \ldots x_k) = (n-k)!$ (for $x_{1\ldots k}$ distinct and 0 else) which is a godsend, and thus gives us a way of getting $n(P)$ in $O(N^2)$ time
- This is optimised down to $O(NlogN)$ by your favourite range sum point add data structure to get $\sum_{j=1}^{P_k-1} f(P_1, \ldots P_{k-1}, j)$ quickly

Implementation: Quite Easy

- https://oj.uz/submission/492562
- Care must be taken to ensure $O(NlogN)$ running time!

Exercise (1 minute)

- The same problem but for balanced bracket sequences
- How do we get $f(x_1 \ \ldots x_k)$?
- Counting problem!

Answer

- $dp[i][j] = \#length\ i + j\ sequences\ beginning\ with\ j\ '('s$

Second half of problem: Getting kth element

- I like to call the technique ‘walking the dp’
- Suppose a cat came around and gave us $X_1 \ \ldots X_k$. Lets try find X_{k+1}
- Suppose that like before we had $f(x_1, \ \ldots, x_k)$ like before

The algorithm pseudocode: We find One element at time

```
X := ∅
K := target_order - 1 //# sequences ≤_lex X
For i = 1 … N
     // Find X_i
     For j = 1 … |Σ| //candidates for X_k in order
          if K ≥ f(X_1 … X_{i-1}, j):
               K := K − f(X_1 … X_{i-1}, j)
          else
               X_i := j
               break
```

Implementation: kth permutation

- Disclaimer: This code has been tested on n = 1 … 5 only

```
#include <iostream>
#include <algorithm>
#define MAXN 22
#define LL long long
using namespace std;

void getkthPerm(int n, LL k) {
    k--;
    LL fac[MAXN]={1};
    for (LL i=1; i<=n; i++) {
        fac[i] = i*fac[i-1];
    }
    bool used[MAXN] = {};
    for (int i=1; i<=n; i++) {
        for (int j=1; j<=n; j++) {
            if (used[j]) {continue;}
            if (fac[n-i] <= k) {k-=fac[n-i];}
            else {
                used[j] = true;
                printf("%d ",j);
                break;
            }
        }
    }
    printf("\n");
}
```

Problemset

Here * Denotes increased difficulty and italics denote essential problems

Section I	Section II	Section III
Polygon game (template provided)	*Oddjobs (template provided)*	*Cryptography (template provided) [oj.uz NOI20 Crypt]*
Outer Space invaders	*Wet Towels (template provided)*	Linear garden [oj.uz]
USACO 2019 December Platnium Q1* [USACO]	Superheros	Calvinball championship [oj.uz]
Mountain (IOI17 prac)** [oj.uz]	Ice Hockey World championships [oj.uz]	twofive **
	Bus tour *	Magical Stones ***
	Fun park (CEOI 2019)** [oj.uz]	

Thank you for your attention!!!