zscoder's blog

[Tutorial] Non-trivial DP Tricks and Techniques

By **zscoder**, history, 8 years ago,

Hi everyone! Today I want to share some DP tricks and techniques that I have seen from some problems. I think this will be helpful for those who just started doing DP. Sometimes the tutorials are very brief and assumes the reader already understand the technique so it will be hard for people who are new to the technique to understand it.

Note : You should know how to do basic DP before reading the post

DP + Bitmasks

This is actually a very well-known technique and most people should already know this. This trick is usually used when one of the variables have very small constraints that can allow exponential solutions. The classic example is applying it to solve the Travelling Salesman Problem in $O(n^2 \cdot 2^n)$ time. We let $dp[i][j]$ be the minimum time needed to visit the vertices in the set denoted by i and ending at vertex j. Note that i will iterate through all possible subsets of the vertices and thus the number of states is $O(2^n \cdot n)$. We can go from every state to the next states in $O(n)$ by considering all possible next vertex to go to. Thus, the time complexity is $O(2^n \cdot n^2)$.

Usually, when doing DP + Bitmasks problems, we store the subsets as an integer from 0 to $2^n - 1$. How do we know which elements belong to a subset denoted by i? We write i in its binary representation and for each bit j that is 1, the j-th element is included in the set. For example, the set $35 = 100011_2$ denotes the set $\{0, 4, 5\}$ (the bits are 0-indexed from left to right). Thus, to test if the j-th element is in the subset denoted by j, we can test if `i & (1<<j)` is positive. (Why? Recall that `(1<<j)` is 2^j and how the & operator works.)

Now, we look at an example problem : 453B - Little Pony and Harmony Chest

So, the first step is to establish a maximum bound for the b_i. We prove that $b_i < 2a_i$. Assume otherwise, then we can replace b_i with 1 and get a smaller answer (and clearly it preserves the coprime property). Thus, $b_i < 60$. Note that there are 17 primes less than 60, which prompts us to apply dp + bitmask here. Note that for any pair b_i, b_j with $i \neq j$, their set of prime factors must be disjoint since they're coprime.

Now, we let $dp[i][j]$ be the minimum answer one can get by changing the first i elements such that the set of primes used (i.e. the set of prime factors of the numbers $b_1, b_2, ..., b_i$) is equal to the subset denoted by j. Let $f[x]$ denote the set of prime factors of x. Since $b_i \leq 60$, we iterate through all possible values of b_i, and for a fixed b_i, let $F = f[b_i]$. Then, let x be the complement of the set F, i.e. the set of primes not used by b_i. We iterate through all subsets of x. (see here for how to iterate through all subsets of a subset x) For each s which is a subset of x, we want $dp[i][s|F] = min(dp[i][s|F], dp[i - 1][s] + abs(a[i] - b[i]))$. This completes the dp. We can reconstruct the solution by storing the position where the dp achieves its minimum value for each state as usual. This solution is enough to pass the time limits.

Here are some other problems that uses bitmask dp :

678E - Another Sith Tournament

662C - Binary Table

Do we really need to visit all the states?

Sometimes, the naive dp solution to a problem might take too long and too much memory. However, sometimes it is worth noting that most of the states can be ignored because they will never be reached and this can reduce your time complexity and memory complexity.

Example Problem : 505C - Mr. Kitayuta, the Treasure Hunter

So, the most direct way of doing dp would be let $dp[i][j]$ be the number of gems Mr. Kitayuta can collect after he jumps to island i, while the length of his last jump is equal to j. Then, the dp transitions are quite obvious, because we only need to test all possible jumps and take the one that yields maximum results. If you have trouble with the naive dp, you can read the original editorial.

However, the naive method is too slow, because it would take $O(m^2)$ time and memory. The key observation here is that most of the states will never be visited, more precisiely j can only be in a certain range. These bounds can be obtained by greedily trying to maximize j and minimize j and we can see that their values will always be in the order of $O(\sqrt{m})$ from the initial length of jump. This type of intuition might come in handy to optimize your dp and turn the naive dp into an AC solution.

Change the object to dp

Example Problem : 559C - Gerald and Giant Chess

This is a classic example. If the board was smaller, say 3000×3000, then the normal 2D dp would work. However, the dimensions of the grid is too large here.

Note that the number of blocked cells is not too large though, so we can try to dp on them. Let S be the set of blocked cells. We add the ending cell to S for convenience. We sort S in increasing order of x-coordinate, and break ties by increasing order of y-coordinate. As a result, the ending cell will always be the last element of S.

Now, let $dp[i]$ be the number of ways to reach the i-th blocked cell (assuming it is not blocked). Our goal is to find $dp[s]$, where $s = |S|$.

Note that since we have sort S by increasing order, the j-th blocked cell will not affect the number of ways to reach the i-th blocked cell if $i < j$. (There is no path that visits the j-th blocked cell first before visiting the i-th blocked cell)

The number of ways from square (x_1, y_1) to (x_2, y_2) without any blocked cells is $\frac{(x_2-x_1+y_2-y_1-2)!}{(x_2-x_1+1)!(y_2-y_1+1)!}$. (if $x_2 > x_1, y_2 > y_1$. The case when some two are equal can be handled trivially). Let $f(P, Q)$ denote the number of ways to reach Q from P. We can calculate $f(P, Q)$ in $O(1)$ by precomputing factorials and its inverse like above.

The base case, $dp[1]$ can be calculated as the number of ways to reach S_1 from the starting square. Similarly, we initialize all $dp[i]$ as the number of ways to reach S_i from the starting square.

Now, we have to subtract the number of paths that reach some of the blocked cells. Assume we already fixed the values of $dp[1], dp[2], ..., dp[i - 1]$. For a fix blocked cell S_i, we'll do so by dividing the paths into groups according to the **first** blocked cell it encounters. The number of ways for each possible first blocked cell j is equal to $dp[j] \cdot f(S_j, S_i)$, so we can subtract this from $dp[i]$. Thus, this dp works in $O(n^2)$.

Another problem using this idea : 722E - Research Rover

Open and Close Interval Trick

Example Problem : 626F - Group Projects

First, note that the order doesn't matter so we can sort the a_i in non-decreasing order. Now, note that every interval's imbalance can be calculated with its largest and smallest value. We start adding the elements to sets from smallest to largest in order. Suppose we're adding the i-th element. Some of the current sets are **open**, i.e. has a minimum value but is not complete yet (does not have a maximum). Suppose there are j open sets. When we add a_i, the sum $a_i - a_{i-1}$ will contribute to each of the j open sets, so we increase the current imbalance by $j(a_i - a_{i-1})$.

Let $dp[i][j][k]$ be the number of ways such that when we inserted the first i elements, there are j **open sets** and the total imbalance till now is k. Now, we see how to do the state transitions. Let $v = dp[i - 1][j][k]$. We analyze which states involves v.

Firstly, the imbalance of the new state must be $val = k + j(a_i - a_{i-1})$, as noted above. Now, there are a few cases :

1. We place the current number a_i in its own group : Then, $dp[i][j][val] += v$.
2. We place the current number a_i in one of the open groups, but **not close it** : Then, $dp[i][j][val] += j \cdot v$ (we choose one of the open groups to add a_i.
3. Open a new group with minimum = a_i : Then, $dp[i][j + 1][val] += v$.
4. Close an open group by inserting a_i in one of them and **close it** : Then, $dp[i][j - 1][val] += j \cdot v$.

The answer can be found as $dp[n][0][0] + dp[n][0][1] + ... + dp[n][0][k]$.

Related Problems :

466D - Increase Sequence

367E - Sereja and Intervals

"Connected Component" DP

Example Problem : JOI 2016 Open Contest — Skyscrapers

Previously, I've made a blog post here asking for a more detailed solution. With some hints from **Reyna**, I finally figured it out and I've seen this trick appeared some number of times.

Abridged Statement : Given $a_1, a_2, ..., a_n$, find the number of permutations of these numbers such that $|a_1 - a_2| + |a_2 - a_3| + ... + |a_{n-1} - a_n| \le L$ where L is a given integer.

Constraints : $n \le 100, L \le 1000, a_i \le 1000$

Now, we sort the values a_i and add them into the permutation one by one. At each point, we will have some connected components of values (for example it will be something like $2, ?, 1, 5, ?, ?, 3, ?, 4$)

Now, suppose we already added a_{i-1}. We treat the $?$ as a_i and calculate the cost. When we add a new number we increase the values of the $?$ and update the cost accordingly.

Let $dp[i][j][k][l]$ be the number of ways to insert the first i elements such that :

- There are j connected components
- The total cost is k (assuming the $?$ are a_{i+1})
- l of the ends of the permutations has been filled. (So, $0 \le l \le 2$)

I will not describe the entire state transitions here as it will be very long. If you want the complete transitions you can view the code below, where I commented what each transition means.

Some key points to note :

- Each time you add a new element, you have to update the total cost by $a_{i+1} - a_i$ times the number of filled spaces adjacent to an empty space.
- When you add a new element, it can either combine 2 connected components, create a new connected components, or be appended to the front or end of one of the connected components.

Code with comments

A problem that uses this idea can be seen here : 704B - Ant Man

$\times 2, +1$ trick

This might not be a very common trick, and indeed I've only seen it once and applied it myself once. This is a special case of the "Do we really need to visit all the states" example.

Example 1 : Perfect Permutations, Subtask 4

My solution only works up to Subtask 4. The official solution uses a different method but the point here is to demonstrate this trick.

Abridged Statement : Find the number of permutations of length N with exactly K inversions. ($K \le N, N \le 10^9, K \le 1000$ (for subtask 4))

You might be wondering : How can we apply dp when N is as huge as 10^9? We'll show how to apply it below. The trick is to skip the unused states.

First, we look at how to solve this when N, K are small.

Let $dp[i][j]$ be the number of permutations of length i with j inversions. Then,
$dp[i][j] = dp[i-1][j] + dp[i-1][j-1] + ... + dp[i-1][j-(i-1)]$. Why? Again we consider the permutation by adding the numbers from 1

to i in this order. When we add the element i, adding it before k of the current elements will increase the number of inversions by k. So, we sum over all possibilities for all $0 \le k \le i - 1$. We can calculate this in $O(N^2)$ by sliding window/computing prefix sums.

How do we get read of the N factor and replace it with K instead? We will use the following trick :

Suppose we calculated $dp[i][j]$ for all $0 \le j \le K$. We have already figured out how to calculate $dp[i + 1][j]$ for all $0 \le j \le K$ in $O(K)$. The trick here is we can calculate $dp[2i][j]$ from $dp[i][j]$ for all j in $O(K^2)$.

How? We will find the number of permutations using $1, 2, ..., n$ and $n + 1, n + 2, ..., 2n$ and combine them together. Suppose the first permutation has x inversions and the second permutation has y inversions. How will the total number of inversions when we merge them? Clearly, there'll be at least $x + y$ inversions.

Now, we call the numbers from 1 to n small and $n + 1$ to $2n$ large. Suppose we already fixed the permutation of the small and large numbers. Thus, we can replace the small numbers with the letter 'S' and large numbers with the letter 'L'. For each L, it increases the number of inversions by the number of Ss at the right of it. Thus, if we want to find the number of ways that this can increase the number of inversions by k, we just have to find the number of unordered tuples of nonnegative integers $(a_1, a_2, ..., a_n)$ such that they sum up to k (we can view a_i as the number of Ss at the back of the i-th L)

How do we count this value? We'll count the number of such tuples where each element is positive and at most k and the elements sum up to k instead, regardless of its length. This value will be precisely what we want for large enough n because there can be at most k positive elements and thus the length will not exceed n when $n > k$. We can handle the values for small n with the naive $O(n^2)$ dp manually so there's no need to worry about it.

Thus, it remains to count the number of such tuples where each element is positive and at most k and sums up to $S = k$. Denote this value by $f(S, k)$. We want to find $S(k, k)$. We can derive the recurrence $f(S, k) = f(S, k - 1) + f(S - k, k)$, denoting whether we use k or not in the sum. Thus, we can precompute these values in $O(K^2)$.

Now, let $g_0, g_1, g_2, ..., g_K$ be the number of permutations of length n with number of inversions equal to $0, 1, 2, ..., K$.

To complete this step, we can multiply the polynomial $g_0 + g_1x + ... + g_Kx^K$ by itself (in $O(K^2)$ or $O(K \log K)$ with FFT, but that doesn't really change the complexity since the precomputation already takes $O(K^2)$), to obtain the number of pairs of permutations of $\{1, 2, ..., n\}$ and $\{n + 1, n + 2, ..., 2n\}$ with total number of inversions i for all $0 \le i \le K$.

Next, we just have to multiply this with $f(0, 0) + f(1, 1)x + ... + f(K, K)x^K$ and we get the desired answer for permutations of length $2n$, as noted above.

Thus, we have found a way to obtain $dp[2i][\cdot]$ from $dp[i][\cdot]$ in $O(K^2)$.

To complete the solution, we first write N in its binary representation and compute the dp values for the number formed from the first 10 bits (until the number is greater than K). Then, we can update the dp values when N is multiplied by 2 or increased by 1 in $O(K^2)$ time, so we can find the value $dp[N][K]$ in $O(K^2 \log N)$, which fits in the time limit for this subtask.

My code.

Example 2 : Problem Statement in Mandarin

This solution originated from the comment from **YuukaKazami** here

Problem Statement : A sequence $a_1, a_2, ..., a_n$ is valid if all its elements are pairwise distinct and $a_i \in [1, A]$ for all i. We define $value(S)$ of a valid sequence S as the product of its elements. Find the sum of $value(S)$ for all possible valid sequences S, modulo p where p is a prime.

Constraints : $A, p \le 10^9, n \le 500, p > A > n + 1$

Firstly, we can ignore the order of the sequence and multiply the answer by $n!$ in the end because the numbers are distinct.

First, we look at the naive solution :

Now, let $dp[i][j]$ be the sum of values of all valid sequences of length j where values from 1 to i inclusive are used.

The recurrence is $dp[i][j] = dp[i - 1][j] + i \cdot dp[i - 1][j - 1]$, depending on whether i is used.

This will give us a complexity of $O(An)$, which is clearly insufficient.

Now, we'll use the idea from the last example. We already know how to calculate $dp[i + 1][\cdot]$ from $dp[i][\cdot]$ in $O(n)$ time. Now, we just have to calculate $dp[2i][\cdot]$ from $dp[i][\cdot]$ fast.

Suppose we want to calculate $dp[2A][n]$. Then, we consider for all possible a the sum of the values of all sequences where a of the elements are selected from $1, 2, ..., A$ and the remaining $n - a$ are from $i + 1, i + 2, ..., 2A$.

Firstly, note that $\prod(a_i + A) = \sum_{s \subset \{0,1,...,n-1\}} A^{n-|s|} \cdot \prod_{i \in s} a_i$.

Now, let a_i denote the sum of all values of sequences of length i where elements are chosen from $\{1, 2, ..., A\}$, i.e. $dp[A][i]$.

Let b_i denote the same value, but the elements are chosen from $\{A + 1, A + 2, ..., 2A\}$.

Now, we claim that $b_i = \sum_{j=0}^{i} A^{i-j} \cdot a_j \cdot \binom{A-j}{i-j}$. Indeed, this is just a result of the formula above, where we iterate through all possible subset sizes. Note that the term $\binom{A-j}{i-j}$ is the number of sets of size i which contains a given subset of size j and all elements are chosen from $1, 2, ..., A$. (take a moment to convince yourself about this formula)

Now, computing the value of $\binom{A-j}{i-j}$ isn't hard (you can write out the binomial coefficient and multiply its term one by one with some precomputation, see the formula in the original pdf if you're stuck), and once you have that, you can calculate the values of b_i in $O(n^2)$.

Finally, with the values of b_i, we can calculate $dp[2A][\cdot]$ the same way as the last example, as $dp[2A][n]$ is just $\sum_{i=0} na_i \cdot b_{n-i}$ and we can calculate this by multiplying the two polynomials formed by $[a_i]$ and $[b_i]$. Thus, the entire step can be done in $O(n^2)$.

Thus, we can calculate $dp[2i][\cdot]$ and $dp[i + 1][\cdot]$ in $O(n^2)$ and $O(n)$ respectively from $dp[i][\cdot]$. Thus, we can write A in binary as in the last example and compute the answers step by step, using at most $\log A$ steps. Thus, the total time complexity is $O(n^2 \log A)$, which can pass.

This is the end of this post. I hope you benefited from it and please share your own dp tricks in the comments with us.

dp, tutorial, tricks