Karan2116's blog

Everything About Dynamic Programming

By **Karan2116**, history, 3 years ago,

I decided to gather some good material on the web related to DP and found some good explanation by svg on topcoder forums..Hence wrote this blog.Will format it when i get time.

Prologue to Experiment 3
OH, WHAT TO WEAR? WHAT TO WEAR?
SUCH A VAST WARDROBE! HOW WILL I EVER FIND THE RIGHT COMBINATION?
PERHAPS I CAN HELP!
WHO ARE YOU?
MACE
I'M HERE TO TELL YOU ABOUT "DynaMix" THE NEW "SELECTION OPTIMIZATION SOFTWARE FROM ALGORITHMIC ANOMALY!
DynaMix
HOW DOES IT WORK?
WE USE DYNAMIC PROGRAMMING TO RECURSIVELY CREATE AN OUTFIT THAT YOU'LL LOOK JUST STUNNING IN!
NOW, IF WE WANT TO DO THIS IN O(n²) TIME, WE'LL HAVE TO HURRY!
LET'S BEGIN! THAT SHIRT AND THOSE PANTS YOU'RE HOLDING — HOW GOOD DO THEY LOOK TOGETHER, ON A SCALE FROM 1 TO 10?
UM... SEVEN?
EXCELLENT! I'LL ENTER THAT INTO OUR ARRAY!
THOSE SHOES WITH THAT NECKLACE?
THREE.
THOSE OTHER SHOES WITH THAT SHIRT?
SIX.
THIS NECKLACE WITH THESE PANTS?
EIGHT.
THAT SHIRT WITH THOSE PANTS?
TWO.
THREE HOURS LATER...
THERE! I'VE TOLD YOU EVERYTHING!
AND NOW, IN A FEW SECONDS, THE PARADIGM WILL HAVE WORKED ITS MAGIC!
VOILA! ALL DONE! SOOO...YOU'LL WANT TO WEAR THESE PANTS, THAT SHIRT, THOSE SHOES, AND THIS NECKLACE!
LATER THAT NIGHT...
DARLING, YOU LOOK ABSOLUTELY DYNAMIC!
SOMEHOW, I KNEW HE WAS GOING TO SAY THAT!
BUT HOW DOES IT WORK? TURN THE PAGE TO FIND OUT!

Problem:

About 25% of all SRM problems have the "Dynamic Programming" category tag. The DP problems are popular among problemsetters because each DP problem is original in some sense and you have to think hard to invent the solution for it. Since dynamic programming is so popular, it is perhaps the most important method to master in algorithm competitions.

The easiest way to learn the DP principle is by examples. The current recipe contains a few DP examples, but unexperienced reader is advised to refer to other DP tutorials to make the understanding easier. You can find a lot of DP examples and explanations in an excellent

→ Pay attention

Before contest
Educational Codeforces Round 70 (Rated for Div. 2)
22:53:10

Like You and 77 others like this.

→ amansehrawat218

Rating: **997**
Contribution: **0**

- Settings
- Blog
- Favourites
- Teams
- Submissions
- Groups
- Talks
- Contests

amansehrawat218

→ Top rated

#	User	Rating
1	tourist	3645
2	Radewoosh	3403
3	LHiC	3336
4	wxhtxdy	3329
5	Benq	3320
6	Um_nik	3301
7	V--o_o--V	3275
8	mnbvmar	3193
9	yutaka1999	3190
10	ainta	3180

Countries | Cities | Organizations View all →

→ Top contributors

#	User	Contrib.
1	Errichto	193
2	Radewoosh	185
3	rng_58	164
3	PikMike	164
5	Vovuh	160
6	majk	158
7	300iq	153
8	Um_nik	150
9	Petr	147
10	kostka	143

View all →

→ Find user

Handle:

Find

tutorial Dynamic Programming: From novice to advanced by Dumitru. The purpose of the recipe is to cover general DP aspects.

Solution

Tutorial (coins example)

So what is exactly the dynamic programming, how can we describe it? There's no clear definition for this technique. It can be rather characterized as an algorithmic technique that is usually based on a starting state of the problem, and a recurrent formula or relation between the successive states. A state of the problem usually represents a sub-solution, i.e. a partial solution or a solution based on a subset of the given input. And the states are built one by one, based on the previously built states.

Let's now consider a very simple problem that will help to understand better the details that will be further discussed: Given a list of n coins, their weights W1, W2, ..., Wn; and the total sum S. Find the minimum number of coins the overall weight of which is S (we can use as many coins of each type as we want), or report that it is not possible to select coins in such a way so that they sum up to S. This problem is a special case of the famous unbounded knapsack problem. For this problem a state, let's call it (P) or (P)->k, would represent the solution for a partial sum (P), where P is not greater than S. k is minimal number of coins required to get exact overall weight P. The k value is usually called the result of corresponding state (P).

A dynamic programming solution would thus start with an initial state (0) and then will build the succeeding states based on the previously found ones. In the above problem, a state (Q) that precedes (P) would be the one for which sum Q is lower than P, thus representing a solution for a sum smaller than P. One starts with the trivial state (0), and then builds the state (P1), (P2), (P3), and so on until the final state (S) is built, which actually represents the solution of the problem. One should note that a state can not be processed until all of the preceding states haven't been processed – this is another important characteristic of DP technique.

The last, but not least, detail to discuss is about finding the relation between states that would allow us to build the next states. For simple problems this relation is quite easy to be observed, but for complex problems we may need to do some additional operations or changes to reach such a relation. Let's again consider the sample problem described above. Consider a sum P of coin weights V1, V2, ..., Vj. The state with sum P can only be reached from a smaller sum Qi by adding a coin Ui to it so that Qi + Ui = P. Thus there is a limited amount of states which would lead to the succeeding state (P). The minimum number of coins that can sum up to P is thus equal to the number of coins of one of the states (Qi), plus one coin, the coin Ui.

Implementation-wise, the DP results are usually stored in an array. In our coin example the array "mink[0..S]" contains k values for states. In other words, mink[P] = k means that result of state (P) is equal to k. The array of DP results is calculated in a loop (often nested loops) in some order. The following piece of code contains recurrent equations for the problem, table of results (contents of array mink) and the solution itself.

```
/* Recurrent equations for DP:
  {k[0] = 0;
  {k[P] = min_i (k[P-Wi] + 1);   (for Wi <= P)
*/
/* Consider the input data: S=11, n=3, W = {1,3,5}
   The DP results table is:
  P = 0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11
  ------+--+--+--+--+--+--+--+--+--+--+--
  k = 0 |1 |2 |1 |2 |1 |2 |3 |2 |3 |2 |3
*/
// The implementation:
int n, S;                                        //n &mdash; number of
coin types, S &mdash; desired overall weight
int wgt[MAXN];                                   //array of coin weights
(W); for example: {1, 3, 5};
int mink[MAXW];                                  //array of DP results
(k); look above for the example;

  mink[0] = 0;                                   //base of DP: 0 weight
can be achieved by 0 coins
```

→ Recent actions

ko_osaga → International Olympiad in Informatics (IOI) 2019
Errichto → IOI preparation lectures and contests
PikMike → Educational Codeforces Round 70 [Rated for Div. 2]
nikolozcimakuridze → Is not it cheating?
sridhar15399 → why dont codeforces support big integer in c++
zrmpaul → Wow!Benq AK IOI2019 Day1!
SPatrik → Codeforces Round #577 (Div 2) Editorial
SPatrik → Codeforces Round #577 (Div 2)
Geothermal → AtCoder Beginner Contest 136 English Solutions
cannor147 → Editorial for Codeforces Round #568 (Div. 2)
anjan0791 → Could any one please help understand time complexities for this tutorial.
chokudai → AtCoder Beginner Contest 136 Announcement
surajkvm007 → balanced parenthesis
nikolozcimakuridze → IOI 2019 predictions
Hd7 → Should i try hard tag problems?
majk → Codeforces Global Round 4
rr459595 → Greedy approach proof
cheek_i_breek → Printing containers in C++
Hiasat → [Gym] ACM Arabella 2019
SupaHotFire → How to get better focus for contests
spirited_away_ → How to derive this formula for this atcoder problem.
DrSwad → Introducing MathMash!
JanchoMath → IOI mirror? live scoreboard?
dungenmaster84 → where can I find editorials for SEERC 2018?
BugLeave → The first article

Detailed →

```
  for (int P = 1; P<=S; P++) {                   //iterate through all
the states
    int minres = 1000000000;
    for (int i = 0; i<n; i++) if (wgt[i] <= P) { //suppose that the coin
with weight wgt[i] is the last
      int tres = mink[P &mdash; wgt[i]] + 1;           //the number of
coins with the coin is greater by one
      if (minres > tres) minres = tres;          //choose the minimal
overall number of coins among all cases
    }
    mink[P] = minres;                            //store the result in
mink array
  }
  int answer = mink[S];                          //the answer for the
whole problem is the result for state (S)
```

Tutorial (LCS example)

Consider another problem: given two words, find the length of their longest common subsequence. For example, for two words "quetzalcoatl" and "tezcatlipoca" the longest subsequence has length 6, f.i. "ezaloa".

To solve the problem we introduce the set of subproblems: given a prefix of the first word and a prefix of the second word, find their LCS. Let the prefix of the first word has length i and the prefix of the second word has length j. As we see, the DP state is determined by two integer parameters: i and j. The state domain is therefore (i,j)->L, where i is the length of first word prefix, j is the length of second word prefix and L is the length of the longest common subsequence of these prefixes. The idea of solution is to take the solution for basic subproblem (0,0) and then add letters to the prefixes one-by-one until we reach the final state (n1,n2) which represents the problem for the full words.

Now let's derive the recurrent relations for DP results denoted as L[i,j]. Clearly, if one of the prefixes is empty, then the LCS must be zero. This is a base equation: L[i,0] = L[0,j] = 0. When i and j are positive then we have to treat several cases: 1. The last letter in the first word prefix is not used in the LCS. So it can be erased without changing the subsequence. The corresponding formula is L[i,j] = L[i-1,j]. 2. The last letter in the second word prefix is unused. Similarly, the formula for the case is: L[i,j] = L[i,j-1] 3. Otherwise, last letters 's' and 't' of both prefixes are included in the common subsequence. Clearly, these letters must be equal. In such a case erasing both last letters will reduce LCS by exactly one. The corresponding formula is: L[i,j] = L[i-1,j-1] + 1 (only if 's' = 't'). Among all three cases we should choose the case which gives the maximal length of sequence.

Implementation-wise, the DP results are stored in two-dimensional array. The values of this array are calculated in two nested loops. It is important that the states are traversed in such order that parameter values are non-decreasing because the DP result for the state (i,j) depends on the results for states (i-1,j), (i,j-1), (i-1,j-1).

```
/* Recurrent relations for DP:
  {L[i,0] = L[0,j] = 0;
  |            {L[i-1,j],
  {L[i,j] = max|L[i,j-1],
               {L[i-1,j-1]+1   (only if last symbols are equal)
*/
/* Table of DP results:
   S|    t  e  z  c  a  t  l  i  p  o  c  a
T ji| 0  1  2  3  4  5  6  7  8  9 10 11 12
----+---------------------------------------
  0 | 0  0  0  0  0  0  0  0  0  0  0  0  0
q 1 | 0  0  0  0  0  0  0  0  0  0  0  0  0
u 2 | 0  0  0  0  0  0  0  0  0  0  0  0  0
e 3 | 0  0  1  1  1  1  1  1  1  1  1  1  1
t 4 | 0  1  1  1  1  1  2  2  2  2  2  2  2
z 5 | 0  1  1  2  2  2  2  2  2  2  2  2  2
a 6 | 0  1  1  2  2  3  3  3  3  3  3  3  3
l 7 | 0  1  1  2  2  3  3  4  4  4  4  4  4
c 8 | 0  1  1  2  3  3  3  4  4  4  4  5  5
o 9 | 0  1  1  2  3  3  3  4  4  4  5  5  5
```

```
a 10| 0  1  1  2  3  4  4  4  4  4  5  5  6
t 11| 0  1  1  2  3  4  5  5  5  5  5  5  6
l 12| 0  1  1  2  3  4  5  6  6  6  6  6  6
*/
// The implementation:
int n1, n2;                                              //lengths of
words
char str1[1024], str2[1024];                             //input words
int lcs[1024][1024];                                     //DP results
array

  for (int i = 0; i<=n1; i++)                            //iterate
through all states (i,j)
    for (int j = 0; j<=n2; j++) {                        //in
lexicographical order
      if (i == 0 || j == 0)
        lcs[i][j] = 0;                                   //the DP base
case
      else {
        lcs[i][j] = max(lcs[i-1][j], lcs[i][j-1]);       //handle cases 1
and 2
        if (str1[i-1] == str2[j-1])
          lcs[i][j] = max(lcs[i][j], lcs[i-1][j-1] + 1); //handle case 3
      }
    }
  int answer = lcs[n1][n2];
```

Discussion

Comparison with memoization

There is another technique called memoization which is covered in detail by recipe "Optimizing recursive solution". Recursive solution with memoization is very similar to backward-style dynamic programming solution. Both methods solve recurrent equations, which means that they deal with state domain — set of states with some result defined. The results for some states are determined from base of recurrence. The results for other states depend on the results of previous states. The DP solution iterates through the states in some particular order set by coder, while memoization iterates through them in order of depth-first search. DP never calculates the DP result for any state twice, just like the recursive solution with full memoization. The memoization approach does not spend time on unnecessary states — it is a lazy algorithm. Only the states which influence the final answer are processed. Here are the pros and cons of memoization over DP: 1[+]. Sometimes easier to code. 2[+]. Does not require to specify order on states explicitly. 3[+]. Processes only necessary states. 4[-]. Works only in the backward-style DP. 5[-]. Works a bit slower than DP (by constant).

Most of DP problems can be divided into two types: optimization problems and combinatoric problems. The optimization problems require you to choose some feasible solution so that the value of goal function is minimized (or maximized). Combinatoric problems request the number of ways to do something or the probability of some event. Let's have a closer look at these problem types.

Optimization DP problem

Optimization problem asks to choose the best feasible solution according to some goal function. Both coins and LCS examples are optimization-type. The recurrent equation looks like R[s] = min(F1(R[i], R[j], ..., R[k]), F2(R[u], R[v], ..., R[w]), ..., Fl(R[q], R[p], ..., R[z])), where R is the DP results array. Simply speaking, the result is chosen as the best = minimal among the several candidate cases. For each case the result is calculated from the results of previous DP states. For example in coins problem all the possible last coin cases are considered. Each of them yields one case in the recurrent formula. The result for the state is a minimum among all such cases. In LCS example there were three cases: first word last letter unused, second word last letter unused and both words last letter used.

It is often useful to fill the DP results array with neutral values before calculating anything. The neutral value is a result which does not affect the problem answer for sure. In case of minimization problem the neutral value is positive infinity: since it is greater than any number, all the recurrent formulas would prefer a case with finite value to such a neutral element. In other words, the state with neutral value result can be thought of as an impossible state. Note that for maximization problem negative infinity is a neutral element.

The DP states are often called DP subproblems because they represent some problem for input data which is subset of the whole problem input. For example, in LCS case each subproblem involves two arbitrary prefixes of the original two words. The DP method relies on the optimal substructure property: given the optimal solution for the whole problem, its partial solutions must be optimal for the subproblems. In the coins case it means that if the solution for whole problem with overall weight S is optimal and it contains coin with weight w, then the solution without w coin must also be optimal for the subproblem with overall weight (S — w).

Optimal substructure property is very important: if it does not hold and the optimal solution has the subsolution which is not optimal, then it would be discarded somewhere in the middle of DP on taking the minimum. Often the DP solution turns out to be theoretically wrong because it lacks the optimal substructure. For example there is a classical travelling salesman problem. Let the DP state domain be (k,l)->D where D is the minimal length of the simple path going through exactly k cities with 0-th city being the first one and l-th city being the last one. The optimal substructure property in such a DP does not hold: given the shortest tour its subpath with fixed last city and overall number of cities is not always the shortest. Therefore the proposed DP would be wrong anyway.

Combinatoric DP problem

The goal of combinatoric DP problem is to find number of ways to do something or the probability that the event happens. Often the number of ways can be big and only the reminder modulo some small number is required. The recurrent equation looks like R[s] = F1(R[i], R[j], ..., R[k]) + F2(R[u], R[v], ..., R[w]) + ... + Fl(R[q], R[p], ..., R[z]). The only difference from optimization case is the sum instead of minimum — and it changes a lot. The summation means that the different ways from F1, F2, ..., Fl cases altogether comprise the all the ways for state (s).

The example of combinatoric case is a modified coins problem: count the number of ways to choose coins so that their overall weight is equal to S. The state domain is the same: (P)->k where k is number of ways to choose coins so that their overall weight is exactly P. The recurrent equations are only a bit different in combinatoric problem.

```
/* Recurrent equations for DP:
  {k[0] = 1;
  {k[P] = sum_i (k[P-Wi]);   (for Wi <= P)
*/
/* Consider the input data: S=11, n=3, W = {1,3,5}
   The DP results table is:
  P = 0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11
  ------+--+--+--+--+--+--+--+--+--+--+--
  k = 1 |1 |1 |2 |3 |5 |8 |12|19|30|47|74
*/
```

There is also a neutral value for combinatoric problem. Since combinatoric problem uses summation, the neutral element is zero. The DP results in combinatoric case usually represents number of ways to do smth, so if the result is zero than there is no way to do it. The neutral result value means that the case is impossible. It may be useful to fill DP results array with zero values, though it is usually done automatically. In case of combinatorics it is important that each possible way is counted and that no way is counted more than once. The second condition is sometimes difficult to satisfy.

Forward vs backward DP style

All the DPs described above are done in backward style. The schema is: iterate through all the states and for each of them calculate the result by looking backward and using the already known DP results of previous states. This style can also be called recurrent since it uses recurrent equations directly for calculation. The relations for backward-style DP are obtained by examining the best solution for the state and trying to decompose it to lesser states.

There is also forward-style DP. Surprisingly it is often more convenient to use. The paradigm of this style is to iterate through all the DP states and from each state perform some transitions leading forward to other states. Each transition modifies the currently stored result for some unprocessed states. When the state is considered, its result is already determined completely. The forward formulation does not use recurrent equations, so it is more complex to prove the correctness of solution strictly mathematically. The recurrent relations used in forward-style DP are obtained by considering one partial solution for the state and trying to continue it to larger states. To perform forward-style DP it is necessary to fill the DP results with neutral values before starting the calculation.

The first example will be combinatoric coins problem. Suppose that you have a partial solution with P overall weight. Then you can add arbitrary coin with weight Wi and get overall weight P+Wi. So you get a transition from state (P) to state (P+Wi). When this transition is considered, the result for state (P) is added to the result of state (P+Wi) which means that all the ways to get P weight can be continued to the ways to get P+Wi weight by adding i-th coin. Here is the code.

```
/* Recurrent relations (transitions) of DP:
  {k[0] = 1;
  {(P)->k ===> (P+Wi)->nk    add k to nk
*/
  //res array is automatically filled with zeroes
  res[0] = 1;                                 //DP base is the same
  for (int p = 0; p<s; p++)                   //iterate through DP states
    for (int i = 0; i<n; i++) {               //iterate through coin to
add
      int np = p + wgt[i];                    //the new state is (np)
      if (np > s) continue;                   //so the transition is (p)
==> (np)
      res[np] += res[p];                      //add the DP result of (p)
to DP result of (np)
    }
  int answer = res[s];                        //problem answer is the
same
```

The second example is longest common subsequence problem. It is of maximization-type, so we have to fill the results array with negative infinities before calculation. The DP base is state (0,0)->0 which represents the pair of empty prefixes. When we consider partial solution (i,j)->L we try to continue it by three ways: 1. Add the next letter of first word to the prefix, do not change subsequence. 2. Add the next letter of the second word to the prefix, do not change subsequence. 3. Only if the next letters of words are the same: add next letter to both prefixes and include it in the subsequence. For each transition we perform so-called relaxation of the larger DP state result. We look at the currently stored value in that state: if it is worse that the proposed one, then it is replaced with the proposed one, otherwise it is not changed. The implementation code and compact representation of DP relations are given below.

```
/* Recurrent relations (transitions) of DP:
  {L[0,0] = 0;
  |          /> (i+1,j)->relax(L)
  {(i,j)->L ==> (i,j+1)->relax(L)
             \> (i+1,j+1)->relax(L+1)  (only if next symbols are equal)
*/
void relax(int &a, int b) {                   //relaxation routine
  if (a < b) a = b;
}

  memset(lcs, -63, sizeof(lcs));              //fill the DP results array
with negative infinity
  lcs[0][0] = 0;                              //set DP base: (0,0)->0
  for (int i = 0; i<=n1; i++)
    for (int j = 0; j<=n2; j++) {             //iterate through all
states
      int tres = lcs[i][j];
      relax(lcs[i+1][j], tres);               //try transition of type 1
      relax(lcs[i][j+1], tres);               //try transition of type 2
      if (str1[i] == str2[j])                 //and if next symbols are
the same
        relax(lcs[i+1][j+1], tres + 1);       //then try transition of
type 3
    }
  int answer = lcs[n1][n2];
```

Recovering the best solution for optimization problems

The optimization problem asks us to find the feasible solution with the minimal value of goal function, but DP finds only the goal function value itself. It does not produce the best solution along with the numerical answer. In practical usage the answer without a solution is useless, though in topcoder algorithm problems often only the answer is required. Anyway, it is useful to know how to reconstruct the best solution after DP.

In the most common case the each transition is an atomic improvement of some partial solution and recurrent equation is something like: R[s] = min(F1(R[u], u), F2(R[v], v), ..., Fk(R[w], w)). In other words, the result for the state is produced from a single best previous state plus some modification. In such a case the DP solution can be reconstructed trivially from the DP solution path. The DP solution path goes from some base DP state to some final state and consists of the states which represent all the partial solutions of the desired best solution. There are two ways to get this path.

The first way is to recalculate the DP from the end to the start. First we choose the final state (f) we want to trace the path from. Then we process the (f) state just like we did it in the DP: iterate through all the variants to get it. Each variant originates in a previous state (p). If the variant produces the result equal to DP result of state (f), then the variant if possible. There is always at least one possible variant to produce the DP result for the state, though there can be many of them. If the variant originating from (p) state is possible, then there is at least one best solution path going through state (p). Therefore we can move to state (p) and search the path from starting state to state (p) now. We can end path tracing when we reach the starting state.

Another way is to store back-links along with the DP result. For each state (s) we save the parameters of the previous state (u) that was continued. When we perform a transition (u) ==> (s) which produces better result than the currently stored in (s) then we set the back-link to (s) to the state (u). To trace the DP solution path we need simply to repeatedly move to back-linked state until the starting state is met. Note that you can store any additional info about the way the DP result was obtained to simplify solution reconstruction.

The first approach has a lot of drawbacks. It is usually slower, it leads to DP code being copy/pasted, it requires backward-style DP for tracing the path. It is good only in the rare case when there is not enough memory to store the back-links required in the second way. The second way is preferred since it is simple to use and supports both backward and forward style DP solutions. If the result for each DP state originates from more than one previous DP state, then you can store the links to all the previous states. The path reconstruction is easiest to implement in the recursive way in such a case.

For example of recovering the solution coins problem is again considered. Note that the DP code is almost the same except that the back-link and item info is set on the relaxation. The later part of tracing the path back is rather simple.

```
/* Consider the input data: S=11, n=3, W = {1,3,5}
   The DP results + back-links table is:
  P  = 0 |1 |2 |3 |4 |5 |6 |7 |8 |9 |10|11
  -------+--+--+--+--+--+--+--+--+--+--+--
mink = 0 |1 |2 |1 |2 |1 |2 |3 |2 |3 |2 |3
prev = ? |S0|S1|S0|S1|S0|S1|S2|S3|S4|S5|S6
item = ? |I0|I0|I1|I1|I2|I2|I2|I2|I2|I2|I2
*/

int mink[MAXW];                      //the DP result array
int prev[MAXW], item[MAXW];          //prev &mdash; array for back-links
to previous state
int k;                               //item &mdash; stores the last item
index
int sol[MAXW];                       //sol[0,1,2,...,k-1] would be the
desired solution

  memset(mink, 63, sizeof(mink));    //fill the DP results with positive
infinity
  mink[0] = 0;                       //set DP base (0)->0
  for (int p = 0; p<s; p++)          //iterate through all states
    for (int i = 0; i<n; i++) {      //try to add one item
      int np = p + wgt[i];           //from (P)->k we get
      int nres = mink[p] + 1;        //to (P+Wi)->k+1
      if (mink[np] > nres) {         //DP results relaxation
```

```
        mink[np] = nres;              //in case of success
        prev[np] = p;                 //save the previous state
        item[np] = i;                 //and the used last item
      }
    }

  int answer = mink[s];
  int cp = s;                         //start from current state S
  while (cp != 0) {                   //until current state is zero
    int pp = prev[cp];                //get the previous state from back-
link
    sol[k++] = item[cp];              //add the known item to solution
array
    cp = pp;                          //move to the previous state
  }
```

The coins tutorial was taken from Dumitru's DP recipe.

Problem

The correct dynamic programming solution for the problem is already invented. And perhaps already coded. But the time/space complexity is unsatisfactory. The solution exceeds time or memory limit or seems to exceed it if it is not implemented yet. The purpose of recipe is do describe various tricks used to optimize DP solutions.

Solution

Most of DP problems are combinatorial or optimization type. In first case we are asked to calculate number of ways to do something, so to get result for a state we have to sum up several formulas that depend on previous results. In second case we need to find solution with minimal value of goal function, and the result for a state is calculated by taking minimum over several formulas which again depend on the previous results. Depending on the type of problem, the main operation will be summation or taking minimum — it will be important in some optimizations.

Consolidate equivalent states

Actually, it is not DP optimization but the main principle of DP. Any recursive solution which has lots of equivalent states can benefit from using DP. If there are no equivalent states in recursive solution, turning it into dynamic programming is useless.

Let's consider some state domain (s)->R which contains two particular states x and y. Intuitively these states are equivalent if the problem answer depends on them in the same way. In other words, there is no difference between these states with respect to the final problem answer. If state domain of DP contains lots of groups of equivalent states, then DP parameters are likely to be redundant. All the equivalence classes can be merged into single states. If the problem is combinatorial, then the result for consolidated state must be sum of all results over the merged group of states. If the problem is minimization, the result must be minimum of results of states merged. This way the new state domain (less in size) can be defined. After it is defined, you need to write the transition relations between new states.

Consider TSP problem as an example. The bruteforce recursive solution searches over all simple paths from city 0 recursively. State of this solution is any simple path which starts from city 0. In this way a state domain is defined and the recursive relations for them are obvious. But then we can notice that states (0,A,B,L) and (0,B,A,L) are equivalent. It does not matter in what order the internal cities were visited — only the set of visited cities and the last city matter. It means that our state domain is redundant, so let's merge the groups of equivalent states. We will get state domain (S,L)->R where S is set of visited states, L is the last city in the path and R is the minimal possible length of such path. The recursive solution with O((N-1)!) states is turned into DP over subsets with O(2^N*N) states.

Prune impossible states

The state is impossible if its result is always equal to zero(combinatorial) / infinity(minimization). Deleting such a state is a good idea since it does not change problem answer for sure. The impossible states can come from several sources:

1. Explicit parameter dependence. The state domain is (A,B) and we know that for all possible states B = f(A) where f is some function (usually analytic and simple). In such a case

B parameter means some unnecessary information and can be deleted from state description. The state domain will be just (A). If we ever need parameter B to perform a transition, we can calculate it as f(A). As the result the size of state domain is decreased dramatically.

2. Implicit parameter dependence. This case is worse. We have state domain (A,B) where A represents some parameters and we know that for any possible state f(A,B) = 0. In other words, for each possible state some property holds. The best idea is of course to express one of parameters as explicitly dependent on the others. Then we can go to case 1 and be happy=) Also if we know that B is either f1(A) or f2(A) or f3(A) or ... or fk(A) then we can change state domain from (A,B) to (A,i) where i=1..k is a number of equation B variant. If nothing helps, we can always use approach 4.

3. Inequalities on parameters. The common way to exploit inequalities for parameters is to set tight loop bounds. For example, if state domain is (i,j) and i<j then we can write for(i=0;i<N;i++) for(j=i+1;j<N;j++) or for(j=0;j<N;j++) for(i=0;i<j;i++). In this case we avoid processing impossible states and average speedup is x(2). If there are k parameters which are non-decreasing, speedup will raise to x(k!).

4. No-thinking ways. Even if it is difficult to determine which states are impossible, the fact of their existence itself can be exploited. There are several ways:

A. Discard impossible states and do not process them. Just add something like "if (res[i][j]==0) continue;" inside loop which iterates over DP states and you are done. This optimization should be always used because overhead is tiny but speedup can be substantial. It does not decrease size of state domain but saves time from state processing.

B. Use recursive search with memoization. It will behave very similar to DP but will process only possible states. The decision to use it must be made before coding starts because the code differs from DP code a lot.

C. Store state results in a map. This way impossible states do not eat memory and time at all. Unfortunately, you'll have to pay a lot of time complexity for this technique: O(log(N)) with ordered map and slow O(1) with unordered map. And a big part of code must be rewritten to implement it.>

Store results only for two layers of DP state domain

It is helpful when the DP is layered. It means that state domain has form (i,A) where i is layer index and A is additional parameters. Moreover, transition rules depend only on two adjacent layers. The usual case is when result for state (i,A) is dependent only on results for states (i-1,*). In such a DP only two neighbouring layers can be stored in memory. Results for each layer are discarded after the next one is calculated. The memory is then reused for the next layer and so on.

Memory contains two layers only. One of them is current layer and another one is previous layer (in case of forward DP one layer is current and another is next). After current layer is fully processed, layers are swapped. There is no need to swap their contents, just swap their pointers/references. Perhaps the easiest approach is to always store even-numbered layers in one memory buffer(index 0) and odd-numbered layers in another buffer(index 1). To rewrite complete and working DP solution to use this optimization you need to do only: 1. In DP results array definition, change the first size of array to two. 2. Change indexation [i] to [i&1] in all accesses to this array. (i&1 is equal to i modulo 2). 3. In forward DP clear the next layer contents immediately after loop over layer index i. Here is the code example of layered forward-style minimization DP: ~~~~~ int res[2][MAXK]; //note that first dimension is only 2 for (int i = 0; i<N; i++) { memset(res[(i+1)&1], 63, sizeof(res[0])); //clear the contents of next layer (set to infinity) for (int j = 0; j<=K; j++) { //iterate through all states as usual int ni = i + 1; //layer index of transition destination is fixed int nj, nres = DPTransition(res[i&1][j], ???); //get additional parameters and results somehow if (res[ni&1][nj] > nres) //relax destination result res[ni&1][nj] = nres; //note &1 in first index } } ~~~~~

This technique reduces memory requirement in O(N) times which is often necessary to achieve desired space complexity. If sizeof(res) reduces to no more than several mebibytes, then speed performance can increase due to cache-friendly memory accesses.

Sometimes you have to store more than two layers. If DP transition relations use not more that k subsequent layers, then you can store only k layers in memory. Use modulo k operation to get the array index for certain layer just like &1 is used in the example.

There is one problem though. In optimization problem there is no simple way to recover the path(solution) of DP. You will get the goal function of best solution, but you won't get the solution itself. To recover solution in usual way you have to store all the intermediate results.

There is a general trick which allows to recover path without storing all the DP results. The path can be recovered using divide and conquer method. Divide layers into approximately two halves and choose the middle layer with index m in between. Now expand the result of DP from (i,A)->R to (i,A)->R,mA where mA is the "middle state". It is value of additional parameter A of the state that lies both on the path and in the middle layer. Now let's see the following: 1. After DP is over, problem answer is determined as minimal result in final layer (with certain properties maybe). Let this result be R,mA. Then (m,mA) is the state in the middle of the path we want to recover. 2. The results mA can be calculated via DP. Now we know the final and the middle states of the desired path. Divide layers into two halves and launch the same DP for each part recursively. Choose final state as answer for the right half and middle state as answer for the left half. Retrieve two states in the middle of these halves and continue recursively. This technique requires additional O(log(N)) time complexity because result for each layer is recalculated approximately log(N) times. If some additional DP parameter is monotonous (for each transition (i,A) — (i+1,B) inequality A<=B holds) then domain of this parameter can also be divided into two halves by the middle point. In such a case asymptotic time complexity does not increase at all.

Precalculate

Often DP solution can benefit from precalculating something. Very often the precalculation is simple DP itself.

A lot of combinatorial problems require precalculation of binomial coefficients. You can precalculate prefix sums of an array so that you can calculate sum of elements in segment in O(1) time. Sometimes it is beneficial to precalculate first k powers of a number.

Although the term precalculation refers to the calculations which are going before the DP, a very similar thing can be done in the DP process. For example, if you have state domain (a,b)->R you may find it useful to create another domain (a,k)->S where S is sum of all R(a,b) with b<k. It is not precisely precalculation since it expands the DP state domain, but it serves the same goal: spend some additional time for the ability to perform a particular operation quickly.>

Rotate the optimization problem

There is a DP solution with state domain (W,A)->R for maximization problem, where W is weight of partial solution, A is some additional parameter and R is maximal value of partial solution that can be achieved. The simple problem unbounded knapsack problem will serve as an example for DP rotation.

Let's place additional requirement on the DP: if we increase weight W of partial solution without changing other parameters including result the solution worsens. Worsens means that the solution with increased weight can be discarded if the initial solution is present because the initial solution leads to better problem answer than the modified one. Notice that the similar statement must true for result R in any DP: if we increase the result R of partial solution the solution improves. In case of knapsack problem the requirement is true: we have some partial solution; if another solution has more weight and less value, then it is surely worse than the current one and it is not necessary to process it any further. The requirement may be different in sense of sign (minimum/maximum. worsens/improves).

This property allows us to state another "rotated" DP: (R,A)->W where R is the value of partial solution, A is the same additional parameter, and W is the minimal possible weight for such a partial solution. In case of knapsack we try to take items of exactly R overall value with the least overall weight possible. The transition for rotated DP is performed the same way. The answer for the problem is obtained as usual: iterate through all states (R,A)->W with desired property and choose solution with maximal value.

To understand the name of the trick better imagine a grid on the plane with coordinates (W,R) where W is row index and R is column index. As we see, the DP stores only the result for rightmost(max-index) cell in each row. The rotated DP will store only the uppermost(min-index) cell in each column. Note the DP rotation will be incorrect if the requirement stated above does not hold.

The rotation is useful only if the range of possible values R is much less than the range of possible weights W. The state domain will take O(RA) memory instead of O(WA) which can help sometimes. For example consider the 0-1 knapsack problem with arbitrary positive real weights and values. DP is not directly applicable in this case. But rotated DP can be used to create fully polynomial approximation scheme which can approximate the correct answer with relative error not more than arbitrary threshold. The idea is to divide all values by small eps and round to the nearest integer. Then solve DP with state domain (k,R)->W where k is number of already processed items, R is overall integer value of items and W is minimal possible overall weight. Note that you cannot round weights in knapsack problem because the optimal solution you obtain this way can violate the knapsack size constraint.

Calculate matrix power by squaring

This technique deals with layered combinatorial DP solution with transition independent of layer index. Two-layered DP has state domain (i,A)->R and recurrent rules in form R(i+1,A) = sum(R(i,B)*C(B)) over all B parameter values. It is important that recurrent rules does not depend on the layer index.

Let's create a vector V(i) = (R(i,A1), R(i,A2), ..., R(i,Ak)) where Aj iterates through all possible values of A parameter. This vector contains all the results on i-th layer. Then the transition rule can be formulated as matrix multiplication: V(i+1) = M * V(i) where M is transition matrix. The answer for the problem is usually determined by the results of last layer, so we need to calculate V(N) = M^N * V(0).

The DP solution is to get V(0) and then multiply it by M matrix N times. It requires O(N * A^2) time complexity, or more precisely it requires O(N * Z) time where Z is number of non-zero elements of matrix M. Instead of one-by-one matrix multiplication, exponentiation by squaring can be used. It calculates M^N using O(log(N)) matrix multiplications. After the matrix power is available, we multiply vector V(0) by it and instantly get the results for last layer. The overall time complexity is O(A^3 * log(N)). This trick is necessary when A is rather small (say 200) and N is very large (say 10^9).

Use complex data structures and algorithms

Sometimes tough DP solutions can be accelerated by using complex acceleration structures or algorithms. Binary search, segment trees (range minimum/sum query), binary search tree (map) are good at accelerating particular operations. If you are desperate at inventing DP solution of Div1 1000 problem with proper time complexity, it may be a good idea to recall these things.

For example, longest increasing subsequence problem DP solution can be accelerated to O(N log(N)) with dynamic range minimum query data structure or with binary search depending on the chosen state domain.

Examples

ConnectTheCities

According to the problem statement, each transmitter can be moved to any location without constraints. But it is useless to change order of transmitters. It can be proven as follows: if you have solution where two transmitters change order, swap their final destinations and number of moves won't increase for sure but the connectivity will remain the same. So we can assume that in optimal solution transmitters are placed in the same order as they stay initially.

In DP solution we place transmitters one-by-one from left to right. Besides number of transmitters already placed, we need to store the position of foremost transmitter. This information is necessary to check connectivity of our chain. Also not to exceed move limit we have to include number of moves made so far into the state of DP. The state domain is (k,f,m)->r where k is number of already placed leftmost transmitters, f is position of front transmitter, m is number of moves made and r is minimal transmittion range necessary to maintain connectivity. Transition is performed by setting the k-th transmitter to some p coordinate which is greater or equal to f. This transmitter becomes foremost one, transmission range can be increased to (p — f) if necessary and number of moves increases by |p — X[k]|. The problem answer is produced by considering all possible placements of all n transmitters and trying to connect the last one to the city B. This DP solution is O(N * D^2 * M) where N is number of transmitters, D is distance between cities and M is maximum number of moves allowed.

It is easy to see that if the number of moves m done so far increases then the partial solution worsens because the remaining limit on moves decreases and the set of possible continuations narrows. So we can rotate the DP problem with parameters m and r. The new state domain is (k,f,r)->m where r is exact transmitter range required for connectivity and m is minimal number of used moves possible with such parameters. The transition as an operation on (k,f,r,m)-tuple remains the exactly same. When we calculate the problem answer we do the same things but also we check that the number of moves does not exceed the limit. The rotated DP is O(N * D^3) in time and can be done in O(D^2) space complexity if "store two layers" optimization is used.

```
int n, d;                                                   //number of
transmitters and distance between cities
int res[2][MAXD][MAXD];                                     //state domain
results: (k,f,r)->m
...
    sort(xarr.begin(), xarr.end());                         //do not forget to
sort the transmitter positions!
    memset(res, 63, sizeof(res));
    res[0][0][0] = 0;                                       //DP base: all zeros
possible, others impossible
    for (int k = 0; k<n; k++) {                             //iterate k &mdash;
number of transmitters places so far
      memset(res[(k+1)&1], 63, sizeof(res[0]));
      for (int f = 0; f<=d; f++)                            //iterate f &mdash;
position of foremost(and last) transmitter
        for (int r = 0; r<=d; r++) {                        //iterate r &mdash;
required transmission range
          int m = res[k&1][f][r];                           //get minimal number
of moves required
          if (m > maxmoves) continue;                       //discard state if it
is impossible or number of moves exceeds the limit
          for (int p = f; p<=d; p++)                        //iterate p &mdash;
possible position of k-th transmitter
            relax(res[(k+1)&1][p][max(r,p-f)], m + abs(p-xarr[k])); //try
transition
        }
    }
    int answer = 1000000000;                                //getting answer as
minimal possible r
    for (int f = 0; f<=d; f++)                              //over all states
with k = n
      for (int r = 0; r<=d; r++) {
        int m = res[n&1][f][r];
        if (m > maxmoves) continue;                         //with number of
moves under limit
        int rans = max(r, d-f);                             //do not forget the
last segment for transmission
        relax(answer, rans);
      }
    return answer;
```

TheSequencesLevelThree

The key idea for this problem is to sort all elements and then construct possible sequences by adding elements one-by-one. Since we add elements in increasing order, each new element can be pushed to the left or to the right. So each partial solution has x left-most elements and y right-most elements already set, the remaining middle part of the sequence is not determined yet. When we add a new element either x or y is increased by one. Since there is additional constraint on the difference between neighbouring elements, we have to memorize the border elements. The values of these elements may be very large, so it is better to store their indices: L is the index of last pushed element on the left (x-th element from the left) and R is the index of last pushed element to the right (y-th element from the right). Look code example for the picture.

Ok, now we have defined state domain (k,x,y,L,R)->C where k is number of used elements, x is number of elements to the left, y is number of elements to the right, L is index of the left border element, R is index of the right border element, C is number of such partial solutions. Note that L and R are 1-indexed for the reason described further in the text. There are two conditional transitions: to (k+1,x+1,y,k+1,R) and to (k+1,x,y+1,L,k+1). We see that DP is layered by parameter k, so we can iterate through all DP states in order of increasing k. To get the problem answer we have to consider all states with one undefined element (i.e. k = N-1) and try to put maximal element to the middle. Also the statement says that there must be at least one element to the left and one to the right of the "top" number, so only states with x>=1 and y>=1 are considered for the answer.

It is obvious that this state domain contains a lot of impossible states because any valid state has x + y = k. From this equation we express y = k — x explicitly. Now we can throw away all states with other y values. The new state domain is (k,x,L,R)->C. It is great that the DP is still layered by parameter k, so the order of state domain traversal remains simple. Note that if we exploited k = x + y instead and used state domain (x,y,L,R)->C we would have to iterate through all states in order of increasing sum x + y.

Ok, but do we really need the x and y parameters?! How does the problem answer depend on them? The only thing that depends on x and y is getting the problem answer. Not exact x and y values are required but only whether they are positive or zero. The states with x=2 and x=5 are thus equivalent, though x=0 and x=1 are not. The information about x and y parameters is almost unnecessary. To deal with x>=1 and y>=1 conditions we introduce "null" element in sequence. If parameter L is equal to zero then there is no element on the left side (as if x = 0). If L is positive then it is index of sequence element which is on the left border (and of course x>0). Right side is treated the same way. Now information about x parameter is not necessary at all. Let's merge equivalent states by deleting parameter x from the state domain. The state domain is now (k,L,R)->C.

But there is still room for improvement. Notice that for any valid state max(L,R) = k. That's because k-th element is the last added element: it must be the left border element or the right border element. So states with max(L,R) != k are all impossible (give zero results). We can exploit this equation. Eliminating parameter k from state domain is not convenient because then we would have to iterate through states (L,R) in order of increasing max(L,R). So we would replace parameters L,R to parameters d,m. These two parameters do not have much sense — they are used only to encode valid L,R pairs: m==false: L = d, R = k; m==true : L = k, R = d; The final state domain is (k,d,m), where k is number of set elements, d is index of element other than k, m means which border element is k. Since DP is layered, we can use "store two layers" space optimization. The final time complexity is O(N^2) and space complexity is O(N). Note that such a deep optimization is overkill because N<=50 in the problem statement. You could stop at O(N^4) =)

```
//
// [a(1), a(2),  ..., a(x-1), arr[L], ?, ?, ..., ?, arr[R], b(y-1), ...,
b(2), b(1)]
//  \____________known_____________/     unknown
\____________known____________/
//             x elements                                       y elements
//                           already set elements: k = x + y

int n;                                                     //k &mdash; number
of elements already set
int64 res[2][MAXN][2];                                     //d: arr[d] is last
element on one of borders
...                                                        //m=0 means arr[d]
is last on the left, m=1 &mdash; on the right
    n = arr.size();                                        //note that actual
elements are enumerated from 1
    arr.push_back(-1);                                     //index d=0 means
there is no element on the border yet
    sort(arr.begin(), arr.end());
    res[0][0][0] = 1;                                      //DP base: no
elements, no borders = 1 variant
    int64 answer = 0;
    if (n < 3) return 0;                                   //(better to handle
this case explicitly)
    for (int k = 0; k<n; k++) {                            //iterate through
all states
      memset(res[(k+1)&1], 0, sizeof(res[0]));             //(do not forget to
clear the next layer)
      for (int d = 0; d<=k; d++)                           //(d cannot be
greater than k)
        for (int m = 0; m<2; m++) {
          int64 tres = res[k&1][d][m];
          if (tres == 0) continue;                         //discard null
states
          int L = (m==0 ? d : k);                          //restore L,R
parameters from d,m
```

```
          int R = (m==0 ? k : d);
          int nelem = arr[k+1];                       //we'll try to add
this element

          if (L==0 || abs(arr[L]-nelem)<=maxd)        //trying to add
element to the left border
            add(res[(k+1)&1][R][0], tres);

          if (R==0 || abs(arr[R]-nelem)<=maxd)        //trying to add
element to the right border
            add(res[(k+1)&1][L][1], tres);

          if (k == n-1)                               //trying to add
highest(last) element to the middle
            if (L>0 && abs(arr[L]-nelem)<=maxd)       //ensure that there
is a good element to the left
              if (R>0 && abs(arr[R]-nelem)<=maxd)     //ensure that there
is a good element to the right
                add(answer, tres);                    //adding nelem to
the middle produces solutions
        }
    }
    return int(answer);
  }
```

GameWithGraphAndTree

The solution of this problem was discussed in detail in recipe "Commonly used DP state domains". Let size(v) be size of v-subtree. The following constraints hold for any useful transition: 1,2) size(p) = |s|; p in s; 3,4) size(q) = |t|; q in t; 5) t and s have no common elements; All the other iterations inside loops are useless because either ires[k][p][s] or gres[son][q][t] is zero so there is not impact of addition.

The 5-th constraint gives a way to reduce O(4^N) to O(3^N) by applying the technique described in recipe "Iterating Over All Subsets of a Set". Since t and s do not intersect, t is subset of complement to s and loop over t can be taken from that recipe. The time complexity is reduced to O(3^N * N^3).

The easiest way to exploit constraints 1 and 2 is to check ires[k][p][s] to be positive immediately inside loops over s. The bad cases for which constraints are not satisfied are pruned and the lengthy calculations inside do not happen for impossible states. From this point is it hard to give precise time complexity. We see that number of possible sets s is equal to C(n, size(p)) where C is binomial coefficient, and this binomial coefficient is equal to O(2^n / sqrt(n)) in worst case. So the time complexity is not worse than O(3^N * N^2.5).

The other optimizations are not important. The cases when q belongs to set s are pruned. Also there is a check that gres[son][q][t] is positive. The check is faster than modulo multiplication inside loop over t so let it be. The loop over t remains the most time-consuming place in code. Ideally this loop should iterate only over subsets with satisfied constraint 3 — it should accelerate DP a lot but requires a lot of work, so it's better to omit it. Here is the optimized piece of code:

```
    for (int p = 0; p<n; p++) {
      for (int s = 0; s<(1<<n); s++) if (ires[k][p][s])
//check that result is not zero &mdash; prune impossible states
        for (int q = 0; q<n; q++) if (graph[p][q]) {
          if (s & (1<<q)) continue;
//prune the case when q belongs to set s
          int oth = ((1<<n)-1) ^ s;
//get complement to set s
          for (int t = oth; t>0; t = (t-1)&oth) if (gres[son][q][t])
//iterate over non-empty subsets of oth
            add(ires[k+1][p][s^t], mult(ires[k][p][s], gres[son][q][t]));
//do calculation only for possible gres states
        }
    }
```

TourCounting

In this problem we are asked to find number of cycles in graph. I'll present a long way to solve this problem by DP to show how DP is accelerated by fast matrix exponentiation. All the cycles have a starting vertex. Let's handle only cycles that start from vertex 0, the case of all vertices can be solved by running the solution for each starting vertex separately. For the sake of simplicity we will count empty cycles too. Since there are exactly n such cycles, subtract n from the final answer to get rid of empty cycles.

The DP state domain is (i,v)->C where i is length of tour, v is last vertex and C is number of ways to get from vertex 0 to vertex v in exactly i moves. The recurrent equation is the following: C(i+1,v) = sum_u(C(i,u) * A[u,v]) where A is adjacency matrix. The DP base is: C(0,*) = 0 except C(0,0) = 1. The answer is clearly sum_i(C(i,0)) for all i=0..k-1. This is DP solution with time complexity O(k*n). It is too much because we have to run this solution for each vertex separately.

Let's try to use binary matrix exponentiation to speed this DP up. The DP is layered, the matrix is just adjacency matrix, but answer depends not only on the last layer, but on all layers. Such a difficulty appears from time to time in problems with matrix exponentiation. The workaround is to add some variables to our vector. Here it is enough to add one variable which is answer of problem. Let R(i) = sum_j(C(j,0)) for all j=0..i-1. The problem answer is then R(k), so it no longer depends on intermediate layers results. Now we have to include it into DP recurrent equations: R(i+1) = R(i) + C(i,0).

To exploit vector-matrix operations we need to define vector of layer results and transition matrix. Vector is V(i) = [C(i,0), C(i,1), C(i,2), ..., C(n-1,0); R(i)]. Matrix TM is divided into four parts: upperleft (n x n) part is precisely the adjacency matrix A, upperright n-column is zero, bottomright element is equal to one, bottomleft n-row is filled with zeros except for first element which is equal to one. It is transition matrix because V(i+1) = V(i) * TM. Better check on the piece of paper that the result of vector-matrix multiplication precisely matches the recurrent equations of DP. The answer is last (n-th) element of vector V(k) = V(0) * TM^k. DP base vector V(0) is (1, 0, 0, ..., 0; 0). If power of matrix is calculated via fast exponentiation, the time complexity is O(n^3 * log(k)). Even if you run this DP solution for each vertex separately the solution will be fast enough to be accepted.

But there are redundant operations in such a solution. Notice that the core part of transition matrix is adjacency matrix. It remains the same for each of DP run. To eliminate this redundancy all the DP runs should be merged into one run. The slowest part of DP run is getting power of transition matrix. Let's merge all transition matrices. The merged matrix is (2*n x 2*n) in size, upperleft (n x n) block is adjacency matrix, upperright block is filled with zeros, bottomright and bottomleft blocks are identity matrices. This matrix contains all previously used transition matrices as submatrices. Therefore the k-th power of this matrix also contains k-th powers of all used transition matrices TM. Now we can get answer of each DP by multiplying the vector corresponding to DP base and getting correct element of result. The time complexity for the whole problem is O(n^3 * log(k)).

```
struct Matrix {                                 //class of (2n x 2n)
matrix
  int arr[MAXN*2][MAXN*2];
};
Matrix Multiply(const Matrix &a, const Matrix &b) { //get product of two
matrices
  Matrix res;
  for (int i = 0; i<s; i++)
    for (int j = 0; j<s; j++) {
      int tres = 0;
      for (int u = 0; u<s; u++)
        add(tres, mult(a.arr[i][u], b.arr[u][j]));
      res.arr[i][j] = tres;
    }
  return res;
}
...
    Matrix matr, res;                           //matr is the
constructed matrix
    n = g.size();                               //res will be its
power
    s = 2*n;                                    //s is size of matrix
  for (int i = 0; i<n; i++)
   for (int j = 0; j<n; j++) {                         //fill the matrix
elements:
    matr.arr[i][j] = (g[i][j] == 'Y');            //upperleft =
adjacency matrix
    matr.arr[i+n][j+n] = (i==j);                  //bottomright =
identity matrix
    matr.arr[i+n][j] = (i==j);                    //borromleft =
identity matrix
   }

  for (int i = 0; i<s; i++)                          //set res matrix to
identity
   for (int j = 0; j<s; j++)
    res.arr[i][j] = (i==j);
  for (int p = k; p>0; p>>=1) {                      //perform binary
exponentiation
   if (p & 1) res = Multiply(res, matr);            //is current bit of
power is set, multiply result by matr
   matr = Multiply(matr, matr);                     //matr represents
(2^b)-th power of original matrix
  }

  int answer = ((-n)%m + m) % m;                     //subtract n empty
cycles from problem answer
  for (int i = 0; i<n; i++)                          //sum results of
individual DP runs
   add(answer, res.arr[n+i][i]);                    //get (n+i)-th
element of V(0) * (matr ^ k)
  return answer;                                     //where V(0) has only
i-th non-zero element set to 1
```

RoadOrFlightHard

In this problem we are asked to find minimal distance between two points in a special graph (with line structure). Just like in Dijkstra algorithm, state domain should include v parameter — city number and the result of state is minimal distance from starting city (0) to current city (v). Also there is constraint on number of takeoffs, so we have to memorize that number as parameter t. After these obvious thoughts we have the state domain (v,t)->D, where v=0..n and t=0..k. Unlike previous examples, this DP solution follows backwards(recurrent) style, so each result is determined using the previous results. The transition rules are rather simple. If king comes from previous city by ground, then D(v,t) = D(v-1,t) + R[v-1] where R is array of road travelling times. If king arrives to current city by plane from city u (u < V), then D(v,t) = D(u,t-1) + (F[u] + F[u+1] + F[u+2] + ... + F[v-2] + F[v-1]). Since any of such conditions may happen, the result of minimum of results for road and flight cases for all possible departure cities. The problem answer is min(D(n,t)) for t=0..k.

Unfortunately, this DP has O(N^3 * K) time complexity. For each possible flight the sum of array elements in a segment is calculated and this calculation results in the innermost loop. To eliminate this loop we have to precalculate prefix sums for flight times. Let S(v) = F(0) + F(1) + F(2) + ... + F(v-1) for any v=0..n. This array is called prefix sums array of F and it can be computed in linear time by obvious DP which originates from these recurrent equations: S(0) = 0; S(v+1) = S(v) + F(v). Having prefix sums array, sum of elements of any segment can be calculated in O(1) time because F(L) + ... + F(R-1) = S(R) — S(L). So the recurrent relations for flight case are rewritten as: D(v,t) = D(u,t-1) + (S(v) — S(u)). The time complexity is now O(N^2 * K).

The next optimization is not that easy. The best result in case king arrives to city v by some plane is D(v,t) = min_u(D(u,t-1) + S(v) — S(u)) where u takes values from 0 to v-1. Transform the equation: D(v,t) = min_u(D(u,t-1) — S(u) + S(v)) = min_u(D(u,t-1) — S(u)) + S(v). Notice that the expression inside the minimum does not depend on v anymore. Let's denote the whole minimum as A(v-1,t-1). The A array is added to state domain and its contents can be calculated during the DP. It is interesting to discuss the meaning of A(v,t). I would say it is the best virtual flight distance from city 0 to city v. It is virtual because the flight can start at any city. Here are the full and final transitions: D(v,t) = min(D(v-1,t) + R[v-1], A(v-1,t-1) + S(v)); A(v,t) = min(A(v-1,t), D(v,t) — S(v));

Now the solution works in O(N * K) time. But the size of results arrays exceed memory limit. The workaround is very simple. Notice that the final DP is layered by v parameter because to get results (v,t) only (v-1,*) and (v,*) states are required. So we can store results only for two adjacent layers at any time. After this optimization space complexity becomes linear O(N + K).

```
int64 sum[MAXN];                                              //prefix
sums array S
int64 vfd[2][MAXK];                                           //A(v,t)
&mdash; virtual flight distance
int64 res[2][MAXK];                                           //D(v,t)
&mdash; minimal distance to city v with t takeoffs
...
  sum[0] = 0;
  for (int v = 0; v<n; v++) sum[v+1] = sum[v] + flight[v];    //prefix
sums calculation

  memset(res, 63, sizeof(res));                               //DP base
for city 0:
  memset(vfd, 63, sizeof(vfd));                               //only zero
takeoffs with zero distance
  res[0][0] = 0;                                              //all other
entries are infinities
  vfd[0][0] = 0;
  for (int v = 1; v<=n; v++)                                  //iterate
through all states with v>0
    for (int t = 0; t<=k; t++) {
      res[v&1][t] = res[(v-1)&1][t] + road[v-1];              //try road
way to here
      if (t > 0) res[v&1][t] = min(res[v&1][t], vfd[(v-1)&1][t-1] +
sum[v]); //try flight arrival here
      vfd[v&1][t] = min(vfd[(v-1)&1][t], res[v&1][t] &mdash; sum[v]);
//try flight departure here
    }
  int64 answer = INF;                                         //find
minimal distance to city n
  for (int t = 0; t<=k; t++) answer = min(answer, res[n&1][t]);
  return answer;
```

Problem: The most creative part of inventing dynamic programming solution is defining recurrent relations. The recurrent relations consist of two parts: state domain and transitions. State domain is a set of states (subproblems) in dynamic programming. For each state the subresult will be calculated eventually. Transitions are the relations between different states which help calculate the subresults.

This recipe covers frequently used state domain types. The general approaches of dealing with them and real SRM examples are given. Also few optimizations specific to particular domains are mentioned here.

Solution Code of DP solution usually contains an array representing subresults on the state domain. For example, classic knapsack problem solution will be like:

```
  int maxcost[items+1][space+1];
  memset(maxcost, -63, sizeof(maxcost));   //fill with negative infinity
  maxcost[0][0] = 0;                       //base of DP
  for (int i = 0; i<items; i++)            //iterations over states in
proper order
    for (int j = 0; j<=space; j++) {
      int mc = maxcost[i][j];              //we handle two types forward
transitions
      int ni, nj, nmc;                     //from state (i,j)->mc to
state (ni,nj)->nmc

      ni = i + 1;                          //forward transition: do not
add i-th item
      nj = j;
      nmc = mc;
      if (maxcost[ni][nj] < nmc)           //relaxing result for new
state
        maxcost[ni][nj] = nmc;

      ni = i + 1;                          //forward transition: add i-th
item
      nj = j + size[i];
      nmc = mc + cost[i];
      if (nj <= space && maxcost[ni][nj] < nmc)
        maxcost[ni][nj] = nmc;
    }
  int answer = -1000000000;                //getting answer from state
results
  for (j = 0; j<=space; j++)
    if (maxcost[items][j] > answer)
      answer = maxcost[items][j];
  return answer;
```

Here (i,j) is state of DP with result equal to maxcost[i][j]. The result here means the maximal cost of items we can get by taking some of first i items with overall size of exactly j. So the set of (i,j) pairs and concept of maxcost[i][j] here comprise a state domain. The forward transition is adding or not adding the i-th item to the set of items we have already chosen.

The order of iterations through all DP states is important. The code above iterates through states with pairs (i,j) sorted lexicographically. It is correct since any transition goes from set (i,*) to set (i+1,*), so we see that i is increasing by one. Speaking in backward (recurrent) style, the result for each state (i,j) directly depends only on the results for the states (i-1,*).

To determine order or iteration through states we have to define order on state domain. We say that state (i1,j1) is greater than state (i2,j2) if (i1,j1) directly or indirectly (i.e. through several other states) depends on (i2,j2). This is definition of order on the state domain used. In DP solution any state must be considered after all the lesser states. Else the solution would give incorrect result.

Multidimensional array The knapsack DP solution described above is an example of multidimensional array state domain (with 2 dimensions). A lot of other problems have similar state domains. Generally speaking, in this category states are represented by k parameters: (i1, i2, i3, ..., ik). So in the code we define a multidimensional array for state results like: int Result[N1][N2][N3]...[Nk]. Of course there are some transition rules (recurrent relations). These rules themselves can be complex, but the order of states is usually simple.

In most cases the states can be iterated through in lexicographical order. To do this you have to ensure that if I = (i1, i2, i3, ..., ik) directly depends on J = (j1, j2, j3, ..., jk) then I is lexicographically greater that J. This can be achieved by permuting parameters (like using (j,i) instead of (i,j)) or reversing them. But it is usually easier to change the order and direction of nested loops. Here is general code of lexicographical traversion:

```
  for (int i1 = 0; i1<N1; i1++)
    for (int i2 = 0; i2<N1; i2++)
      ...
        for (int ik = 0; ik<Nk; ik++) {
          //get some states (j1, j2, j3, ..., jk) -> jres by performing
transitions
          //and handle them
        }
```

Note: changing order of DP parameters in array and order of nested loops can noticably affect performance on modern computers due to CPU cache behavior.

This type of state domain is the easiest to understand and implement, that's why most DP tutorials show problems of this type. But it is not the most frequently used type of state domain in SRMs. DP over subsets is much more popular.

Subsets of a given set

The problems of this type has some set X. The number of elements in this set is small: less than 20. The idea of DP solution is to consider all subsets of X as state domain. Often there are additional parameters. So generally we have state domain in form (s,a) where s is a subset of X and "a" represents additional parameters.

Consider TSP problem as an example. The set of cities X={0, 1, 2, ..., N-1} is used here. State domain will have two parameters: s and a. The state (s,a)->R means that R is the shortest path from city 0 to city "a" which goes through all the vertices from subset s exactly once. The transition is simply adding one city v to the end of path: (s,a)->R turns into (s+{v},v)->R + M[a,v]. Here M[i,j] is distance between i-th and j-th city. Any hamiltonian cycle is a path which goes through each vertex exactly once plus the edge which closes the cycle, so the answer for TSP problem can be computed as min(R[X,a]+M[a,0]) among all vertices "a".

It is very convenient to encode subsets with binary numbers. Look recipe "Representing sets with bitfields" for detailed explanation.

The state domain of DP over subsets is usually ordered by set inclusion. Each forward transition adds some elements to the current subset, but does not subtract any. So result for each state (s,a) depends only on the results of states (t,b) where t is subset of s. If state domain is ordered like this, then we can iterate through subsets in lexicographical order of binary masks. Since subsets are usually represented with binary integers, we can iterate through all subsets by iterating through all integers from 0 to 2^N — 1. For example in TSP problem solution looks like:

```
  int res[1<<N][N];
  memset(res, 63, sizeof(res));       //filling results with positive
infinity
  res[1<<0][0] = 0;                   //DP base

  for (int s = 0; s < (1<<N); s++)    //iterating through all subsets in
lexicographical order
    for (int a = 0; a < N; a++) {
      int r = res[s][a];
      for (int v = 0; v < N; v++) {   //looking through all transitions
(cities to visit next)
        if (s & (1<<v)) continue;     //we cannot visit cities that are
already visited
        int ns = s | (1<<v);          //perform transition
        int na = v;
        int nr = r + matr[a][v];      //by adding edge (a &mdash; v)
distance
        if (res[ns][na] > nr)         //relax result for state (ns,na)
with nr
          res[ns][na] = nr;
      }
    }
  int answer = 1000000000;            //get TSP answer
  for (int a = 0; a < N; a++)
    answer = min(answer, res[(1<<N)-1][a] + matr[a][0]);
```

Often in DP over subsets you have to iterate through all subsets or supersets of a given set s. The bruteforce implementation will require O(4^N) time for the whole DP, but it can be easily optimized to take O(3^N). Please read recipe "Iterating Over All Subsets of a Set".

Substrings of a given string

There is a fixed string or a fixed segment. According to the problem definition, it can be broken into two pieces, then each of pieces can be again divided into two pieces and so forth until we get unit-length strings. And by doing this we need to achieve some goal.

Classical example of DP over substrings is context-free grammar parsing algorithm. Problems which involve putting parentheses to arithmetic expression and problems that ask to optimize the overall cost of recursive breaking are often solved by DP over substrings. In this case there are two special parameters L and R which represent indices of left and right borders of a given substring. There can be some additional parameters, we denote them as "a". So each state is defined by (L,R,a). To calculate answer for each state, all the ways to divide substring into two pieces are considered. Because of it, states must be iterated through in order or non-decreasing length. Here is the scheme of DP over substrings (without additional parameters):

```
  res[N+1][N+1];                              //first: L, second: R
  for (int s = 0; s<=N; s++)                  //iterate size(length) of
substring
    for (int L = 0; L+s<=N; L++) {            //iterate left border index
      int R = L + s;                          //right border index is clear
      if (s <= 1) {
        res[L][R] = DPBase(L, R);             //base of DP &mdash; no
division
        continue;
      }
      tres = ???;
      for (int M = L+1; M<=R-1; M++)          //iterate through all divisions
        tres = DPInduction(tres, res[L][M], res[M][R]);
      res[L][R] = tres;
    }
  answer = DPAnswer(res[0][N]);
```

Subtrees(vertices) of a given rooted tree

The problem involves a rooted tree. Sometimes a graph is given and its DFS search tree is used. Some sort of result can be calculated for each subtree. Since each subtree is uniquely identified by its root, we can treat DP over subtrees as DP over vertices. The result for each non-leaf vertex is determined by the results of its immediate children.

The DP over subtree has a state domain in form (v,a) where v is a root of subtree and "a" may be some additional parameters. states are ordered naturally be tree order on vertices. Therefore the easiest way to iterate through states in correct order is to launch DFS from the root of tree. When DFS exits from a vertex, its result must be finally computed and stored in global memory. The code generally looks like:

```
  bool vis[N];                                      //visited mark for DFS
  res[N];                                           //DP result array

  void DFS(int v) {                                 //visit v-rooted subtree
recursively
    vis[v] = true;                                  //mark vertex as visited
    res[v] = ???;                                   //initial result, which
is final result in case v is leaf
    for (int i = 0; i<nbr[v].size(); i++) {         //iterate through all
sons s
      int s = nbr[v][i];
      if (!vis[s]) {                                //if vertex is not
visited yet, then it's a son in DFS tree
        DFS(s);                                     //visit it recursively
        res[v] = DPInduction(res[v], res[s]);       //recalculate result for
current vertex
      }
    }
  }
  ...
  memset(vis, false, sizeof(vis));                  //mark all vertices as
not visited
  DFS(0);                                           //run DFS from the root =
vertex 0
  answer = DPAnswer(res[0]);                        //get problem answer from
result of root
```

Sometimes the graph of problem is not connected (e.g. a forest). In this case run a series of DFS over the whole graph. The results for roots of individual trees are then combined in some way. Usually simple summation/maximum or a simple formula is enough but in tough cases this "merging problem" can turn out to require another DP solution.

The DPInduction is very simple in case when there are no additional parameters. But very often state domain includes the additional parameters and becomes complicated. DPInduction turns out to be another(internal) DP in this case. Its state domain is (k,a) where k is number of sons of vertex considered so far and "a" is additional info. Be careful about the storage of results of this internal DP. If you are solving optimization problem and you are required to recover the solution (not only answer) then you have to save results of this DP for solution recovering. In this case you'll have an array globalres[v,a] and an array internalres[v,k,a]. Topcoder problems rarely require solution, so storage of internal DP results is not necessary. It is easier not to store them globally. In the code below internal results for a vertex are initialized after all the sons are traversed recursively and are discarded after DFS exits a vertex. This case is represented in the code below:

```
  bool vis[N];
  gres[N][A];
  intres[N+1][A];

  void DFS(int v) {
    vis[v] = true;

    vector<int> sons;
    for (int i = 0; i<nbr[v].size(); i++) {    //first pass: visit all
sons and store their indices
      int s = nbr[v][i];
      if (!vis[s]) {
        DFS(s);
        sons.push_back(s);
      }
    }

    int SK = sons.size();                      //clear the internal
results array
    for (int k = 0; k<=SK; k++)
      memset(intres[k], ?, sizeof(intres[k]));

    for (int a = 0; a<A; a++)                  //second pass: run
internal DP over array of sons
      intres[0][a] = InternalDPBase(v, a);
    for (int k = 0; k<SK; k++)                 //k = number of sons
considered so far
      for (int a = 0; a<A; a++)                //a = additional parameter
for them
        for (int b = 0; b<A; b++) {            //b = additional parameter
for the son being added
          int na = DPTransition(v, a, b);
          int nres = DPInduction(intres[k][a], gres[sons[k]][b]);
          intres[k+1][na] = DPMerge(intres[k+1][na], nres);
        }
    for (int a = 0; a<A; a++)                  //copy answer of internal
DP to result for vertex
      gres[v][a] = intres[SK][a];
  }
  ...
  memset(vis, false, sizeof(vis));              //series of DFS
  for (int v = 0; v<N; v++) if (!vis[v]) {
    DFS(v);
    ???                                         //handle results for
connected component
  }
  ???                                           //get the answer in some
way
```

It is very important to understand how time/space complexity is calculated for DP over subtrees. For example, the code just above requires O(N*A^2) time. Though dumb analysis says it is O(N^2*A^2): {N vertices} x {SK<=N sons for each} x A x A. Let Ki denote number of sons of vertex i. Though each Ki may be as large as N-1, their sum is always equal to N-1 in a rooted tree. This fact is the key to further analysis. Suppose that DFS code for i-th vertex runs in not more than Ki*t time. Since DFS is applied only once to each vertex, the overall time will be TC(N) = sum(Ki*t) <= N*t. Consider t=A^2 for the case above and you'll get O(N*A^2) time complexity. To benefit from this acceleration, be sure not to iterate through all vertices of graph in DFS. For example above, running memset for the whole intres array in DFS will raise the time complexity. Time of individual DFS run will become O(N*A + Ki*A^2) instead of O(Ki*A^2). The overall time complexity will become O(N^2*A + N*A^2) which is great regress in case if A is much smaller that N. Using the same approach you may achieve O(N*A) space complexity in case you are asked to recover solution. We have already said that to recover solution you have to store globally the array internalres[v,k,a]. If you allocate memory for this array dynamically, then you can ignore completely states with k>Ki. Since the sum of all Ki is N, you will get O(N*A) space.

Layer count + layer profile

This is the toughest type of DP state domain. It is usually used in tiling or covering problems on special graphs. The classic examples are: calculate number of ways to tile the rectangular board with dominoes (certain cells cannot be used); or put as many chess figures on the chessboard as you can so that they do not hit each other (again, some cells may be restricted).

Generally speaking, all these problems can be solved with DP over subsets (use set of all cells of board). DP with profiles is an optimization which exploits special structure in this set. The board we have to cover/tile is represented as an array of layers. We try to consider layers one by one and store partial solutions after each layer. In simple rectangular board case layer is one row of the board. The profile is a subset of cells in current row which are already tiled.

The state domain has form (k,p) where k is number of fully processed layers and p is so-called profile of solution. Profile is the necessary information about solution in layers that are not fully processed yet. The transitions go from (k,p) to (k+1,q) where q is some new profile. The number of transitions for each state is usually large, so they all are iterated through by recursive search, sometimes with pruning. The search has to find all the ways to increase the partial solution up to the next layer.

The example code below calculates the number of way to fully cover empty cells on the given rectangular board with dominoes.

```
int res[M+1][1<<N];
                                        //k = number of fully tiled rows
int k, p, q;                            //p = profile of k-th row =
subset of tiled cells
bool get(int i) {                       //q = profile of the next row (in
search)
  return matr[k][i] == '#'
      || (p & (1<<i));                  //check whether i-th cell in
current row is not free
}
void Search(int i) {                    //i = number of processed cells
in current row
  if (i == N) {
    add(res[k+1][q], res[k][p]);        //the current row processed, make
transition
    return;
  }

  if (get(i)) {                         //if current cell is not free,
skip it
    Search(i+1);
    return;
  }

  if (i+1<N && !get(i+1))               //try putting (k,i)-(k,i+1)
domino
    Search(i+2);

  if (k+1<M && matr[k+1][i] != '#') {   //try putting (k,i)-(k+1,i)
domino
    q ^= (1<<i);                        //note that the profile of next
row is changed
    Search(i+1);
    q ^= (1<<i);
  }
}
...
res[0][0] = 1;                       //base of DP
for (k = 0; k<M; k++)                //iterate over number of
processed layers
  for (p = 0; p<(1<<N); p++) {       //iterate over profiles
    q = 0;                           //initialize the new profile
    Search(0);                       //start the search for all
transitions
  }
int answer = res[M][0];              //all rows covered with empty
profile = answer
```

The asymptotic time complexity is not easy to calculate exactly. Since search for i performs one call to i+1 and one call to i+2, the complexity of individual search is not more than N-th Fibonacci number = fib(N). Moreover, if profile p has only F free cells it will require O(fib(F)) time due to pruning. If we sum C(N,F)fib(F) for all F we'll get something like (1+phi)^N, where phi is golden ratio. The overall time complexity is O(M * (1+phi)^N). Empirically it is even lower.

The code is not optimal. Almost all DP over profiles should use "storing two layers" space optimization. Look "Optimizing DP solution" recipe. Moreover DP over broken profiles can be used. In this DP state domain (k,p,i) is used, where i is number of processed cells in a row. No recursive search is launched since it is converted to the part of DP. The time complexity is even lower with this solution.

The hard DP over profiles examples can include extensions like: 1. Profile consists of more than one layer. For example to cover the grid with three-length tiles you need to store two layers in the profile. 2. Profile has complex structure. For example to find optimal in some sense hamiltonian cycle on the rectangular board you have to use matched parentheses strings as profiles. 3. Distinct profile structure. Set of profiles may be different for each layer. You can store profiles in map in this case.

Examples:

InformFriends

We are asked to find assignment man->fact which maximizes the number of facts with constraint that everyone must know all the facts. In the optimal assignment people can be divided into fact-groups. Each fact-group consists of people who are told the same fact. And each fact-group must be able to tell everybody else about the fact. Let's precalculate for each subset of people whether they can become a fact-group. The subset can be a fact-group if set of all thier friends united with them is the whole set. After possible fact-groups are calculated, we have to determine maximal number of non-intersecting fact-groups in the set of people. We can define state domain (s)->R where s is subset of people and R is problem answer for them. To determine the answer for state s we have to subtract one of its fact-groups. It is a subset of s and forms a fact-group. So we can iterate through all subsets of s and try them as fact-groups.

```
  n = matr.size();
  int i, j, u;
  for (i = 0; i<(1<<n); i++) {                  //for all subsets i
    int mask = 0;
    for (j = 0; j<n; j++) if (i&(1<<j)) {       //iterate through
people in it
      mask |= (1<<j);
      for (u = 0; u<n; u++) if (matr[j][u]=='Y')   //accumulate the total
set of informed people
        mask |= (1<<u);
    }
    cover[i] = (mask == (1<<n)-1);              //if everyone is
informed, the subset if fact-group
  }

  int ans = 0;
  for (i = 0; i<(1<<n); i++) {                  //go through states
    res[i] = 0;
    for (j = i; j>0; j = (j-1)&i) if (cover[j])    //iterate through all
fact-group subsets
      if (res[i] < res[i^j] + 1)
        res[i] = res[i^j] + 1;                     //relax the result for
i
    if (ans < res[i]) ans = res[i];                //relax the global
answer
  }
  return ans;
```

Breaking strings (ZOJ 2860)

This problem is solved with DP over substrings. Let's enumerate all required breaks and two ends of string with numbers 0, 1, 2, ..., k. Then res[L,R] will be result for the substring which starts in L-th point and ends in R-th point. To get this result we should look through all possible middle points M and consider res[L][M] + res[M][R] + (x[R]-x[L]) as a result. By doing this we get a clear O(k^3) solution (which is TL). What makes this problem exceptional is the application of Knuth's optimization. This trick works only for optimization DP over substrings for which optimal middle point depends monotonously on the end points. Let mid[L,R] be the first middle point for (L,R) substring which gives optimal result. It can be proven that mid[L,R-1] <= mid[L,R] <= mid[L+1,R] — this means monotonicity of mid by L and R. If you are interested in a proof, read about optimal binary search trees in Knuth's "The Art of Computer Programming" volume 3 binary search tree section. Applying this optimization reduces time complexity from O(k^3) to O(k^2) because with fixed s (substring length) we have m_right(L) = mid[L+1][R] = m_left(L+1). That's why nested L and M loops require not more than 2k iterations overall.

```
  for (int s = 0; s<=k; s++)                    //s &mdash; length(size)
of substring
    for (int L = 0; L+s<=k; L++) {              //L &mdash; left point
      int R = L + s;                            //R &mdash; right point
      if (s < 2) {
        res[L][R] = 0;                          //DP base &mdash; nothing
to break
        mid[L][R] = l;                          //mid is equal to left
border
        continue;
      }
      int mleft = mid[L][R-1];                  //Knuth's trick: getting
bounds on M
      int mright = mid[L+1][R];
      res[L][R] = 1000000000000000000LL;
      for (int M = mleft; M<=mright; M++) {     //iterating for M in the
bounds only
        int64 tres = res[L][M] + res[M][R] + (x[R]-x[L]);
        if (res[L][R] > tres) {                 //relax current solution
          res[L][R] = tres;
          mid[L][R] = M;
        }
      }
    }
  int64 answer = res[0][k];
```

BlockEnemy

Since tree is given in the problem statement, we should try DP on subtrees first=) Given any correct solution for the whole tree, any subtree of it is also correct. So if all pairs of occupied towns are separated, then in each subtree they are also separated. So we can try to use state domain (v)->R where v represents subtree and R represents minimal effort to solve the problem for this subtree. But we'll discover soon that connecting subtrees correctly is impossible because we need to know whether there is an occupied town connected with outside part of subtree. We call such a solution for subtree dangerous. Now we add the boolean parameter (solution is safe/dangerous) in the state domain. Let res[v][t] be the minimal effort needed to get a correct solution for v-subtree which is dangerous(if d=1)/safe(if d=0). Initially we consider edge which is going out of subtree indestructible. We will handle the destruction of outgoing edge later. In this case, the solution for leaves is easy to obtain. If v is non-occupied leaf, then it forms a safe solution, and dangerous solution is impossible to get. If it is occupied, then the solution is dangerous with no effort, and impossible to make safe. Then if the vertex v is not leaf, we add its sons one by one. When adding a son s to v-subtree, we have the following merging rules: 1. v=safe + s=safe -> v=safe 2. v=dangerous + s=safe -> v=dangerous 3. v=safe + s=dangerous -> v=dangerous 4. v=dangerous + s=dangerous -> incorrect solution After merging by these rules, we receive a solution for v-subtree in case of indestructible outgoing edge. Now what changes if the outgoing edge is destructible? We can destruct it by paying additional effort. In this case a dangerous solution turns into safe one. The root of tree has no outgoing edge. The problem answer is minimal effort of safe and dangerous solutions for the root ot DFS tree. The time complexity is O(N^2) because adjacency matrix is used, but can be easily reduced to O(N) with neighbors lists.

```
int res[MAXN][2];

void DFS(int v, int f) {                            //traverse v-subtree,
f is father of v
  vis[v] = true;                                    //mark v as visited
  res[v][0] = (occ[v] ? 1000000000 : 0);            //result in case of
leaf
  res[v][1] = (occ[v] ? 0 : 1000000000);
  for (int s = 0; s<n; s++) if (matr[v][s] < 1000000000) {
    if (vis[s]) continue;                           //iterate over all
sons
    DFS(s, v);                                      //run DFS recursively

    int nres[2];
    nres[0] = res[v][0] + res[s][0];                //safe case requires
safe s and safe v
    nres[1] = min(res[v][0] + res[s][1], res[v][1] + res[s][0]);
    res[v][0] = nres[0];                            //dangerous case
requires dangerous + safe
    res[v][1] = nres[1];
  }
  if (f >= 0 && res[v][0] > res[v][1] + matr[v][f]) //we can destroy
upgoing edge (v-f)
    res[v][0] = res[v][1] + matr[v][f];
}
...
  DFS(0, -1);                                       //run DFS from 0
(with no father)
  return min(res[0][0], res[0][1]);                 //whether root is
dangerous does not matter
}
```

ConstructionFromMatches

We need to construct grid from matches obeying certain rules which have local effect. And the grid is 2 x n which is ideal for DP with profiles. Of course, it is better to slice the grid to 2-cell layers instead of N-cell ones. The profile consists of thicknesses of the last two vertical sticks. They are required to check sum when filling the next layer. To perform the transition, we have to iterate through all possible variants of next five sticks thicknesses and choose only variants that satisfy two equations on cell sums. If we perform it bruteforce, we will get O(N*K^7) algorithm, that's slow. It can be easily optimized if we calculate thicknesses of two sticks explicitly by using cell sum equations. This way the DP will be O(N*K^5) in time and O(N*K^2) in space. That is more than enough for the problem. If you want a better solution, you can notice that: 1. You can use "storing two layers" optimization and reduce space complexity to O(K^2). 2. You can transform the solution to DP with broken profiles. To do it you should add intermediate "half" states (i,p,b,v). hres[i][p][b][v] means minimal cost of full construction of (2*i+1) cells — i layers and a top cell in the next layer. The DP with broken profiles will require O(N*K^4) time and O(N*K^3) space, which can be reduced with technique 1 to O(K^3). The code for simple solution is given below. Since width of profile is well-known and very small, it is easier to write transition code as nested loops instead of recursive search.

```
//...?? aa      //schema of profile and transition
//...  u  p     //u and v comprise profile
//...  u  p     //a, b, c, p, q are five added sticks
//...?? bb      //system of equations is:
//...  v  q     //{u + a + p + b = top[i]
//...  v  q     //{v + b + q + c = bottom[i]
//...?? cc      //the depicted transition leads to (i+1,p,q) state
```

```
  memset(res, 63, sizeof(res));                        //DP base
  for (int u = 0; u<k; u++)                            //choose any two
sticks
    for (int v = 0; v<k; v++)                          //put them to leftmost
vertical line
      res[0][u][v] = cost[u] + cost[v];                //their cost is clear

  for (int i = 0; i<n; i++)
    for (int u = 0; u<k; u++)
      for (int v = 0; v<k; v++) {                      //iterate through
states
        int cres = res[i][u][v];
        for (int a = 0; a<k; a++)                      //choose a and p in
all possible ways
          for (int p = 0; p<k; p++) {
            int b = top[i]-4 &mdash; (u + a + p);         //b is uniquely
determined by top equation
            if (b < 0 || b >= k) continue;             //though it can be
bad...
            for (int q = 0; q<k; q++) {                //choose all possible
q variants
              int c = bottom[i]-4 &mdash; (v + b + q);   //c is uniquely
determined by bottom equation
              if (c < 0 || c >= k) continue;

              int nres = cres + cost[p] + cost[q]      //the new solution
cost
                  + cost[a] + cost[b] + cost[c];
              if (res[i+1][p][q] > nres)               //relaxing the
destination
                res[i+1][p][q] = nres;
            }
          }
      }

  int answer = 1000000000;                             //the last two sticks
do not matter
  for (int u = 0; u<k; u++)                            //choose the best
variant among them
    for (int v = 0; v<k; v++)
      if (answer > res[n][u][v])
        answer = res[n][u][v];
  if (answer >= 1000000000) answer = -1;               //do not forget
"impossible" case
```

GameWithGraphAndTree

This problem is solved by very tricky DP on the tree and subsets. We are required to find number of mappings of the tree on the graph. First of all, we choose root of the tree because it is easier to handle rooted tree. Clearly, we should consider all possible submapping of all subtrees on all vertex-subsets of the graph. The number of these submapping is huge, so we have to determine which properties of these submappings are important for extending the mapping. It turns out that these properties are: 1. Subtree denoted by its root vertex v. Necessary to check the outgoing edge mapping later. 2. Vertex of graph p which is the image of v. Again: necessary to check the mapping of added edge. 3. The full image of v-subtree in graph — set s of already mapped vertices in graph. Necessary to maintain bijectivity of mapping. Therefore we define state domain (v,p,s)->GR. GR is number of submappings with the properties we are interested in. To combine results of sons in tree we need to run another "internal" DP. Remember that internal DP is local for each vertex v of the tree. The first parameter will be number of sons already merged — this is quite standard. Also we'll use additional parameters p and s inside. The state domain is (k,p,s)->IR where IR is the number of submappings of partial v-subtree on graph with properties: 1. The vertex v and subtrees corresponding to its first k sons are being mapped (called domain). 2. Image of v is vertex p in graph. 3. The full image of mapping considered is s — subset of already used vertices. The transition of this internal DP is defined by adding one subtree corresponding to k-th son to the domain of mapping. For example, if w is the k-th son, then we add global state GR[w,q,t] to internal state IR[k,p,s] and get internal state IR[k+1,p,s+t]. Here we must check that there is an edge p-q in the graph and that sets s and t have no common elements. The combinations considered in GR[w,q,t] and IR[k,p,s] are independent, so we can just add their product to the destination state. The answer of internal DP is IR[sk,p,s] which is stored as a result GR[k,p,s] of global DP. This is correct solution of this problem. Unfortunately, it runs in O(4^N * N^3) if implemented like it is in the code below. Of course it gets TL. You need to optimize the solution even further to achieve the required performance. The recipe "Optimizing DP solution" describes how to get this solution accepted.

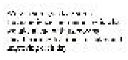
```
int gres[MAXN][MAXN][SIZE];                                 //global DP on
subtrees: (v,p,s)->GR
int ires[MAXN][MAXN][SIZE];                                 //internal DP:
(k,p,s)->IR

void DFS(int v) {                                           //solve DP for
v subtree
  vis[v] = true;
  vector<int> sons;
  for (int u = 0; u<n; u++) if (tree[v][u] && !vis[u]) {   //visit all
sons in tree
    DFS(u);                                                 //calculate
gres[u,...] recursively
    sons.push_back(u);                                      //ans save
list of sons
  }
  int sk = sons.size();
  memset(ires[0], 0, sizeof(ires[0]));                      //base of
internal DP
  for (int p = 0; p<n; p++) ires[0][p][1<<p] = 1;           //one-vertex
mappings v -> p
  for (int k = 0; k<sk; k++) {                              //iterate
through k &mdash; number of sons considered
    int w = sons[k];
    memset(ires[k+1], 0, sizeof(ires[k+1]));                //remember to
clear next layer
    for (int p = 0; p<n; p++) {                             //iterate
through p &mdash; image of v
      for (int s = 0; s<(1<<n); s++)                        //iterate
through s &mdash; full image of current domain
        for (int q = 0; q<n; q++) if (graph[p][q])          //consider
adding mapping which maps:
          for (int t = 0; t<(1<<n); t++) {                  //w -> q;  w-
subtree -> t subset;
            if (s & t) continue;                            //do not break
bijectivity
            add(ires[k+1][p][s^t], mult(ires[k][p][s], gres[w][q][t]));
          }                                                 //add product
of numbers to solution
    }
  }
  memcpy(gres[v], ires[sk], sizeof(ires[sk]));              //since
partial v-subtree with k=sk is full v-subtree
}                                                           //we have
GR[v,p,s] = IR[sk,p,s]
...
    DFS(0);                                                 //launch DFS
from root = 0-th vertex
    int answer = 0;                                         //consider all
variants for i &mdash; image of 0
    for (int i = 0; i<n; i++) add(answer, gres[0][i][(1<<n)-1]);
    return answer;                                          //sum this
variants up and return as an answer
  }
};
```

dp, cookbook

+90 Karan2116 3 years ago 7

Comments (7)

Write comment?

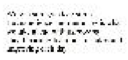


Karan2116

3 years ago, # | +3

*Auto comment: topic has been updated by **Karan2116** (previous revision, new revision, compare).*

→ Reply

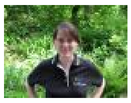


Karan2116

3 years ago, # | +3

*Auto comment: topic has been updated by **Karan2116** (previous revision, new revision, compare).*

→ Reply



Nickolas

3 years ago, # | +17

The original threads in TopCoder Cookbook project forum:

- Introducing Dynamic Programming
- Optimizing DP solution
- Commonly used DP state domains

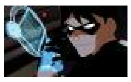
→ Reply

binary_eagle

3 years ago, # | 0

If this is not pleasure, I don't know what is :)

→ Reply



robin_0

3 years ago, # | 0

Wonderful work :)

Thank you :)

→ Reply

kulukamal

17 months ago, # | +6

where to find problem statement of above discussed examples.

→ Reply

Expert
Radewoosh
Mateusz Radecki, Radom, Poland
From University of Warsaw
Contest rating: 3209
Contribution: +124
Friend of: 666 users

for

17 months ago, # | +5

You forgot this: http://codeforces.com/blog/entry/49691

→ Reply