Forums

Select a Forum

Search | Watch Thread | My Post History | My Watches | User Settings
View: Flat (newest first) | Threaded | Tree
Previous Thread | Next Thread

Forums ▸ TopCoder Cookbook ▸ Algorithm Competitions - New Recipes ▸ 2.4.6 Optimizing DP solution

2.4.6 Optimizing DP solution | Feedback: (+17/-0) | [+] [-] | Reply — Wed, Jan 12, 2011 at 9:12 PM IST

syg96
182 posts

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.>

Re: 2.4.6 Optimizing DP solution (response to post by **syg96**) | Feedback: (+7/-0) | [+] [-] | Reply — Wed, Jan 12, 2011 at 9:12 PM IST

syg96
182 posts

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

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.>

Re: 2.4.6 Optimizing DP solution (response to post by **syg96**) | Feedback: (+7/-0) | [+] [-] | Reply — Wed, Jan 12, 2011 at 9:13 PM IST

syg96
182 posts

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.

Re: 2.4.6 Optimizing DP solution (response to post by **syg96**) | Feedback: (+5/-0) | [+] [-] | Reply — Wed, Jan 12, 2011 at 9:13 PM IST

syg96
182 posts

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 - number of transmitters places so far
      memset(res[(k+1)&1], 63, sizeof(res[0]));
      for (int f = 0; f<=d; f++)                   //iterate f - position of foremost(and last) transmitter
        for (int r = 0; r<=d; r++) {               //iterate r - 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 - 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;
```

Re: 2.4.6 Optimizing DP solution (response to post by **syg96**) | Feedback: (+6/-0) | [+] [-] | Reply — Wed, Jan 12, 2011 at 9:13 PM IST

syg96
182 posts

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 - 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 - 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);
  }
```

Re: 2.4.6 Optimizing DP solution (response to post by **syg96**) | Feedback: (+5/-1) | [+] [-] | Reply — Wed, Jan 12, 2011 at 9:13 PM IST

syg96
182 posts

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) - virtual flight distance
int64 res[2][MAXK];                                       //D(v,t) - 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] - 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;
```

Re: 2.4.6 Optimizing DP solution (response to post by **syg96**) | Feedback: (+4/-1) | [+] [-] | Reply — Wed, Jan 12, 2011 at 9:13 PM IST

syg96
182 posts

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
```

Re: 2.4.6 Optimizing DP solution (response to post by **syg96**) | Feedback: (+4/-1) | [+] [-] | Reply — Wed, Jan 12, 2011 at 9:14 PM IST

syg96
182 posts

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 - 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
        }
    }
```

END OF RECIPE

Re: 2.4.6 Optimizing DP solution (response to post by **syg96**) | Feedback: (+5/-0) | [+] [-] | Reply — Wed, Jan 12, 2011 at 11:44 PM IST

hacker007
125 posts

An extremely wonderful post. It would be perfect if you be willing to accomplish the "General DP things" part of your plan.
This draft currently lack forward/backward dynamics stuff and could be better.

Re: 2.4.6 Optimizing DP solution (response to post by **hacker007**) | Feedback: (+3/-1) | [+] [-] | Reply — Tue, Jan 24, 2012 at 2:06 AM IST

a_syco
12 posts

Can I like the above all post infinite times ?? ,please !!!!!!

Forums ▸ TopCoder Cookbook ▸ Algorithm Competitions - New Recipes ▸ 2.4.6 Optimizing DP solution
Previous Thread ▸ | Next Thread ▸