Forums

2.4.? Commonly used DP state domains | Feedback: (+50/-1) | [+] [-] | Reply 3 edits | Tue, Jan 4, 201

syg96
182 posts

Problem:
The most creative part of inventing dynamic programming solution is defining recurrent relations. The recurrent relations con parts: state domain and transitions. State domain is a set of states (subproblems) in dynamic programming. For each state the 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 a 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 pro 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 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 forwar 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 lexicog correct since any transition goes from set (i,*) to set (i+1,*), so we see that i is increasing by one. Speaking in backward (recurrer 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 tha (i1,j1) directly or indirectly (i.e. through several other states) depends on (i2,j2). This is definition of order on the state domain u 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 problems have similar state domains. Generally speaking, in this category states are represented by k parameters: (i1, i2, i3, ..., code we define a multidimensional array for state results like: int Result[N1][N2][N3]...[Nk]. Of course there are some transition 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 depends on J = (j1, j2, j3, ..., jk) then I is lexicographically greater that J. This can be achieved by permuting parameters (like usi (i,j)) or reversing them. But it is usually easier to change the order and direction of nested loops. Here is general code of lexicog traversal:

```
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 compu
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 typ
the most frequently used type of state domain in SRMs. DP over subsets is much more popular.

Re: 2.4.? Commonly used DP state domains (response to post by syg96) | Feedback: (+18/-0) | [+] [-] | Reply 5 edits | Tue, Jan 4, 201

syg96
182 posts

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 t
subsets of X as state domain. Often there are additional parameters. So generally we have state domain in form (s,a) where s is
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
(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.
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
hamiltonian cycle is a path which goes through each vertex exactly once plus the edge which closes the cycle, so the answer fo
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 explar

The state domain of DP over subsets is usually ordered by set inclusion. Each forward transition adds some elements to the cu
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
ordered like this, then we can iterate through subsets in lexicographical order of binary masks. Since subsets are usually repre
binary integers, we can iterate through all subsets by iterating through all integers from 0 to 2^N - 1. For example in TSP probl
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 - 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
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 S

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 pie
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 parenthes
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. The
additional parameters, we denote them as "a". So each state is defined by (L,R,a). To calculate answer for each state, all the wa
substring into two pieces are considered. Because of it, states must be iterated through in order or non-decreasing length. Her
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 - 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]);
```

Re: 2.4.? Commonly used DP state domains (response to post by syg96) | Feedback: (+9/-0) | [+] [-] | Reply 2 edits | Tue, Jan 4, 2

Subtrees(vertices) of a given rooted tree

syg96
182 posts

The problem involves a rooted tree. Sometimes a graph is given and its DFS search tree is used. Some sort of result can be calc
subtree. Since each subtree is uniquely identified by its root, we can treat DP over subtrees as DP over vertices. The result for e
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.
ordered naturally be tree order on vertices. Therefore the easiest way to iterate through states in correct order is to launch DF
tree. When DFS exits from a vertex, its result must be finally computed and stored in global memory. The code generally looks

```
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 cas
  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 so
      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 resu
individual trees are then combined in some way. Usually simple summation/maximum or a simple formula is enough but in to
"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 add
parameters and becomes complicated. DPInduction turns out to be another(internal) DP in this case. Its state domain is (k,a) w
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 reco
(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,
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 glo
below internal results for a vertex are initialized after all the sons are traversed recursively and are discarded after DFS exits a v
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 in
    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 so
    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 ad
        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 vert
    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 ab
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 tre
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
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 mems

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).
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 alread
recover solution you have to store globally the array internalres[v,k,a]. If you allocate memory for this array dynamically, the
completely states with k>Ki. Since the sum of all Ki is N, you will get O(N*A) space.

Re: 2.4.? Commonly used DP state domains (response to post by **syg96**) | Feedback: (+10/-0) | [+] [-] | Reply Tue, Jan 4, 201

syg96
182 posts

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 exa
calculate number of ways to tile the rectangular board with dominoes (certain cells cannot be used); or put as many chess figu
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
which exploits special structure in this set. The board we have to cover/tile is represented as an array of layers. We try to consi
by one and store partial solutions after each layer. In simple rectangular board case layer is one row of the board. The profile
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
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 r
number of transitions for each state is usually large, so they all are iterated through by recursive search, sometimes with prun
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 domino

```
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, t
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(
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 comple
(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 s
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
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
profiles.
3. Distinct profile structure.
Set of profiles may be different for each layer. You can store profiles in map in this case.

Re: 2.4.? Commonly used DP state domains (response to post by **syg96**) | Feedback: (+5/-0) | [+] [-] | Reply 1 edit | Tue, Jan 4, 201

syg96
182 posts

Examples:

InformFriends

We are asked to find assignment man->fact which maximizes the number of facts with constraint that everyone must know all
optimal assignment people can be divided into fact-groups. Each fact-group consists of people who are told the same fact. An
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
united with them is the whole set. After possible fact-groups are calculated, we have to determine maximal number of non-int
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. T
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 thr
subsets of s and try them as fact-groups.

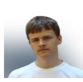
```
n = matr.size();
int i, j, u;
for (i = 0; i<(1<<n); i++) {                      //for all subsets i
  int mask = 0;
  for (j = 0; j<n; j++) if (i&(1<<j)) {           //iterate through people in it
    mask |= (1<<j);
    for (u = 0; u<n; u++) if (matr[j][u]=='Y')    //accumulate the total set of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

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,
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 throu
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
What makes this problem exceptional is the application of Knuth's optimization. This trick works only for optimization DP ove
which optimal middle point depends monotonously on the end points. Let mid[L,R] be the first middle point for (L,R) substrin
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 a
a proof, read about optimal binary search trees in Knuth's "The Art of Computer Programming" volume 3 binary search tree s
Applying this optimization reduces time complexity from O(k^3) to O(k^2) because with fixed s (substring length) we have m_ri
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 - length(size) of substring
  for (int L = 0; L+s<=k; L++) {                //L - left point
    int R = L + s;                              //R - right point
    if (s < 2) {
      res[L][R] = 0;                            //DP base - 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];
```

Re: 2.4.? Commonly used DP state domains (response to post by **syg96**) | Feedback: (+5/-0) | [+] [-] | Reply Tue, Jan 4, 201



syg96
182 posts

BlockEnemy

Since tree is given in the problem statement, we should try DP on subtrees first=) Given any correct solution for the whole tre
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
domain (v)->R where v represents subtree and R represents minimal effort to solve the problem for this subtree. But we'll disc
connecting subtrees correctly is impossible because we need to know whether there is an occupied town connected with outs
subtree. We call such a solution for subtree dangerous. Now we add the boolean parameter (solution is safe/dangerous) in the
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
solution for leaves is easy to obtain. If v is non-occupied leaf, then it forms a safe solution, and dangerous solution is impossib
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
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 t
edge is destructible? We can destruct it by paying additional effort. In this case a dangerous solution turns into safe one. The r
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) l
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 matte
}
```

Re: 2.4.? Commonly used DP state domains (response to post by **syg96**) | Feedback: (+3/-0) | [+] [-] | Reply Tue, Jan 4, 201


syg96
182 posts

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
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 v
They are required to check sum when filling the next layer. To perform the transition, we have to iterate through all possible v
five sticks thicknesses and choose only variants that satisfy two equations on cell sums. If we perform it bruteforce, we will get
algorithm, that's slow. It can be easily optimized if we calculate thicknesses of two sticks explicitly by using cell sum equations
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[
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 requi
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 c
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 - (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 - (v + b + q);     //c is uniquely determined by bottom equat
              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
```

Re: 2.4.? Commonly used DP state domains (response to post by syg96) | Feedback: (+4/-0) | [+] [-] | Reply Tue,

syg96
182 posts

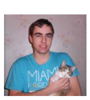
GameWithGraphAndTree

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 tl
choose root of the tree because it is easier to handle rooted tree. Clearly, we should consider all possible submapping of all su
subsets of the graph. The number of these submapping is huge, so we have to determine which properties of these submappi
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 comb
tree we need to run another "internal" DP. Remember that internal DP is local for each vertex v of the tree. The first paramete
already merged - this is quite standard. Also we'll use additional parameters p and s inside. The state domain is (k,p,s)->IR whe
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 ex
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 t
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 indeper
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 cou
to optimize the solution even further to achieve the required performance. The recipe "Optimizing DP solution" describes how
accepted.

```
int gres[MAXN][MAXN][SIZE];                               //global DP on subtrees: (v,p,s)->(
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 - number of sor
    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 - image of v
      for (int s = 0; s<(1<<n); s++)                      //iterate through s - full image o:
        for (int q = 0; q<n; q++) if (graph[p][q])        //consider adding mapping which maj
          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 solutic
    }
  }
  memcpy(gres[v], ires[sk], sizeof(ires[sk]));            //since partial v-subtree with k=sl
}                                                         //we have GR[v,p,s] = IR[sk,p,s]
...
    DFS(0);                                               //launch DFS from root = 0-th verte
    int answer = 0;                                       //consider all variants for i - ima
    for (int i = 0; i<n; i++) add(answer, gres[0][i][(1<<n)-1]);
    return answer;                                        //sum this variants up and return a
  }
};
```

END OF RECIPE

Re: 2.4.? Commonly used DP state domains (response to post by syg96) | Feedback: (+2/-0) | [+] [-] | Reply 2 edits | Tue, Jan 4, 201



caustique
35 posts

This recipe is great, in my opinion, considering there's a few interesting tutorials about dp in the web (not just knapsack and LC
some more content) if any. Sought for one some time ago (especially wanted to read about dp over subsets) and now I have a
it. I mean it's good not only for Cookbook but for me so thank you very much.

Re: 2.4.? Commonly used DP state domains (response to post by syg96) | Feedback: (+5/-2) | [+] [-] | Reply Fri, Jan 7, 201

dimkadimon

Thank you! This is the most useful recipe I have read so far. Perhaps it will finally help me to understand DP and move to yell

3739 posts

Re: 2.4.? Commonly used DP state domains (response to post by **syg96**) | Feedback: (+2/-1) | [+] [-] | Reply Fri, Jan 7, 201


dimkadimon
3739 posts

For the multidimensional array, I don't understand what does Result[i1][i2][i3]...[ik] represent in terms of the Knapsack probler mean the maximum cost that we can obtain by using i1 number of items 1, i2 number of items2, ..., ik number of k-th item? If s longer 0-1 Knapsack, but more like unbounded Knapsack. By the way, it will be great if you can use the Knapsack example for a representation types.

Re: 2.4.? Commonly used DP state domains (response to post by **syg96**) | Feedback: (+0/-0) | [+] [-] | Reply Fri, Jan 7, 201


dimkadimon
3739 posts

For the subsets of a given set you should mention that M[x,y] is the distance between cities x and y. You should use M in the co matr). You should explain the reasoning behind R[X,a]+M[a,0]: R[x,a] is the shortest path that visits every node once starting fro R[X,a]+M[a,0] is the shortest Hamilton cycle for X.

Re: 2.4.? Commonly used DP state domains (response to post by **syg96**) | Feedback: (+3/-0) | [+] [-] | Reply Fri, Jan 7, 2011


dimkadimon
3739 posts

For substrings of a given string. Can you avoid using 'l' in the code, because it looks too similar to one. You can use L and R inst "tres=?;"? What is a in this case and is it actually used in the code?

Re: 2.4.? Commonly used DP state domains (response to post by **dimkadimon**) | Feedback: (+6/-0) | [+] [-] | Reply 1 edit | Sat, Jan 8, 201


syg96
182 posts

1. The Result[i1,i2,...,ik] has nothing is common with knapsack. It is array of results for general problem with multidim-like stat Knapsack was an example of such a problem with dim=2.

2. I added explanation for M matrix and about the way answer is got. About M and matr - it is usual thing. Mathematics is used t variable names (1-2 letters) whereas in programming short names is bad habbit. I have a strong feeling against naming a matri letter, sorry=) By the way, didn't you notice that state result is denoted as R in text but as res in code?...

3. Changed l,m,r to L,M,R. (l+1) looks really weird=) Question mark represents something that depends on the problem. It is us element i.e. 0 in combinatoric problem, +inf in minimization problem, -inf in maximization problem. I don't want to explain it h may be exceptions. That's why I just put question mark. Look one of examples and you'll see 10^18 in this place.

a represents additional parameters. It must be explained somewhere in the recipe=) I added a note on this to the first place it i

I don't like that additional parameters are represented by letter "a". Because it is english article also... I've enclosed parameter in most places in text so that the do not confuse reader.

Re: 2.4.? Commonly used DP state domains (response to post by **syg96**) | Feedback: (+9/-2) | [+] [-] | Reply Sun, Jan 9, 201

supersmecher
2 posts

This is a great post! Thanks for sharing it and congrats you are a smart guy!

Forums TopCoder Cookbook Algorithm Competitions - New Recipes 2.4.? Commonly used DP state domains
Previous Thread | Next Thread

Twitter

Follow

Recent Blog Posts Updated

Apr 23 @timmhicks – Tim Hicks Happy Hump Day topcoders! We are excited to announce that we will be releasing a new look for the very popular /tc by...Read More

Apr 23 Do you ever find yourself hitting "send" on an email and wondering if it'll arrive in the recipient's inbox? Sending email has become so ubiquitous, simple and...Read More

Apr 22 @ClintonBon – Clinton Bonner We know what you're thinking. Great, another 'puff piece' on the 'wisdom of crowds' and

About topcoder

The topcoder community gathers the world's experts in design, development and data science to work on interesting and challenging problems for fun and reward. We want to help topcoder members improve their skills, demonstrate and gain reward for their expertise, and provide the industry with objective insight on new and emerging technologies.

About Us

how all we need to do is post...Read More

View More

© 2014 topcoder. All Rights reserved.
Privacy Policy | Terms

Get Connected

Your email address

Submit