

Get Time

Forums

Select a Forum

[Search](#) | [Watch Thread](#) | [My Post History](#) | [My Watched](#)
View: Flat ([newest first](#)) |[Previous Thread](#)
[Forums](#)
[TopCoder Cookbook](#)
[Algorithm Competitions - New Recipes](#)
2.4.5 Introducing Dynamic Programming
2.4.5 Introducing Dynamic Programming | Feedback: (+28/-1) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Sat, Feb 15, 2014


syg96
 182 posts
Problem:

About 25% of all SRM problems have the "Dynamic Programming" category tag. The DP problems are popular among problem solvers. The DP problem is original in some sense and you have to think hard to invent the solution for it. Since dynamic programming is surely 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 readers may find it difficult to understand. You can find a lot of DP examples and explanations in an excellent tutorial: [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 problem, and a recurrent formula or relation between the successive states. A state of the problem usually represents a sub-problem 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 W_1, W_2, \dots, W_n ; and the total sum S . Find the minimum number of coins the overall weight is equal to S . One starts with the trivial state (0), and then builds the state (P1), (P2), (P3), and so on until the final state (S) is reached. This 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 minimal number of coins required to get exact overall weight P. The k value is usually the minimal number of coins required to get exact overall weight P. The k value is usually the minimal number of coins required to get exact overall weight P.

A dynamic programming solution would thus start with an initial state (0) and then will build the succeeding states based on the previous states. 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 smaller sum 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 reached. This represents the solution of the problem. One should note that a state can not be processed until all of the preceding states have 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 state. This relation is quite easy to be observed, but for complex problems we may need to do some additional operations or change the relation.

Let's again consider the sample problem described above. Consider a sum P of coin weights V_1, V_2, \dots, V_j . The state with sum P can be reached from a smaller sum Q_i by adding a coin U_i to it so that $Q_i + U_i = P$. Thus there is a limited amount of states which would lead to (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 (Q_i), plus one.

Implementation-wise, the DP results are usually stored in an array. In our coin example the array "mink[0..S]" contains k values. The word, 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 loop) following piece of code contains recurrent equations for the problem, table of results (contents of array mink) and the solution.

```

/* 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 - number of coin types, S - desired overall sum
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 definition

mink[0] = 0;                                  //base of DP: 0 weight can be achieved by 0 coins
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 used
        int tres = mink[P - wgt[i]] + 1;        //the number of coins with the coin is greater than minres
        if (minres > tres) minres = tres;        //choose the minimal overall number of coins
    }
    mink[P] = minres;                          //store the result in mink array
}

```

`int answer = mink[S];``//the answer for the whole problem is the res`Re: 2.4.5 Introducing Dynamic Programming (response to [post](#) by [syg96](#)) | Feedback: (+10/-0) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Sat, Feb 12, 2011

**syg96**

182 posts

[Tutorial \(LCS example\)](#)

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

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 the length of the longest common subsequence. As we see, the DP state is determined by parameters: i and j . The state domain is therefore $(i,j) \rightarrow L$, where i is the length of first word prefix, j is the length of second word prefix. The length of the longest common subsequence of these prefixes. The idea of solution is to take the solution for basic subproblem then add letters to the prefixes one-by-one until we reach the final state (n_1, n_2) 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 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. If they are not equal, 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	j	0	1	2	3	4	5	6	7	8	9	10	11	12
T	i	0	1	2	3	4	5	6	7	8	9	10	11	12
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 memoization is very similar to backward-style dynamic programming solution. Both methods solve recurrent equations, which they deal with state domain - set of states with some result defined. The results for some states are determined from base of recursion. The results for other states depend on the results of previous states. The DP solution iterates through the states in some part set by coder, while memoization iterates through them in order of depth-first search. DP never calculates the DP result for any state just like the recursive solution with full memoization. The memoization approach does not spend time on unnecessary states 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).

Re: 2.4.5 Introducing Dynamic Programming (response to [post](#) by [syg96](#)) | Feedback: (+8/-0) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Sat, Feb 12, 2011



[syg96](#)
182 posts

Most of DP problems can be divided into two types: optimization problems and combinatoric problems. The optimization problem asks you to choose some feasible solution so that the value of goal function is minimized (or maximized). Combinatoric problems require a 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 example: optimization-type. The recurrent equation looks like $R[s] = \min(F1(R[i], R[j], \dots, R[k]), F2(R[u], R[v], \dots, R[w]), \dots, F_l(R[q], R[p], \dots, R[z]))$ the DP results array. Simply speaking, the result is chosen as the best = minimal among the several candidate cases. For each case is calculated from the results of previous DP states. For example in coins problem all the possible last coin cases are considered, then it yields one case in the recurrent formula. The result for the state is a minimum among all such cases. In LCS example the 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 all the recurrent formulas would prefer a case with finite value to such a neutral element. In other words, the state with neutral value 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 a subset of the whole 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 a 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 substructure which is not optimal, it would be discarded somewhere in the middle of DP on taking the minimum. Often the DP solution turns out to be theoretic because it lacks the optimal substructure. For example there is a classical [travelling salesman problem](#). Let the DP state domain be 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 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 remainder modulo some small number is required. The recurrent equation looks like $F(R[j], \dots, R[k]) + F2(R[u], R[v], \dots, R[w]) + \dots + F_l(R[q], R[p], \dots, R[z])$. The only difference from optimization case is the sum instead of min/max. It changes a lot. The summation means that the different ways from F1, F2, ..., Fl cases altogether comprise all the ways for the state.

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: $(P) \rightarrow k$ 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 0. A result of 0 in combinatoric case usually represents number of ways to do smth, so if the result is zero then there is no way to do it. A 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.

Re: 2.4.5 Introducing Dynamic Programming (response to [post](#) by [syg96](#)) | Feedback: (+7/-0) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Sat, Feb 12, 2011

Forward vs backward DP style



[syg96](#)
182 posts

All the DPs described above are done in backward style. The schema is: iterate through all the states and for each of them calculate result by looking backward and using the already known DP results of previous states. This style can also be called recurrent: 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 states and from each state perform some transitions leading forward to other states. Each transition modifies the currently stored result. When the state is considered, its result is already determined completely. The forward formulation of recurrent equations, so it is more complex to prove the correctness of solution strictly mathematically. The recurrent relations for forward-style DP are obtained by considering one partial solution for the state and trying to continue it to larger states. To prove 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 add an 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 processed, the result for state (P+Wi) is updated.

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 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 infinities before calculation. The DP base is state (0,0)->0 which represents the pair of empty prefixes. When we consider partial >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 worse than 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 infinities
lcs[0][0] = 0;                //set DP base: (0,0)->0
for (int i = 0; i<=n1; i++)   //iterate through all states
    for (int j = 0; j<=n2; j++) {
        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];
```

Re: 2.4.5 Introducing Dynamic Programming (response to [post](#) by [syg96](#)) | Feedback: (+6/-0) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Sat, Feb 12, 2011



syg96
182 posts

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 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 solution after DP.

In the most common case each transition is an atomic improvement of some partial solution and recurrent equation is $R[s] = \min(F_1(R[u], u), F_2(R[v], v), \dots, F_k(R[w], w))$. In other words, the result for the state is produced from a single best previous state with some modification. In such a case the DP solution can be reconstructed trivially from the DP solution path. The DP solution path is 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 and produces the result equal to DP result of state (f), then the variant if possible. There is always at least one possible variant that produces the DP result for the state, though there can be many of them. If the variant originating from (p) state is possible, then at least one best solution path goes through state (p). Therefore we can move to state (p) and search the path from starting state to 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 continued. When we perform a transition (u) ==> (s) which produces better result than the currently stored in (s) then we set the back-link of (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. That way 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 path tracing. It is good only in the rare case when there is not enough memory to store the back-links required in the second approach. The second way is preferred since it is simple to use and supports both backward and forward style DP solutions. If the result for a state originates from more than one previous DP state, then you can store the links to all the previous states. The path reconstruction can be implemented 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 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 - array for back-links to previous state
int k; //item - 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
}

```

END OF RECIPE

The coins tutorial was taken from [Dumitru's DP recipe](#).

Re: 2.4.5 Introducing Dynamic Programming (response to [post](#) by [syg96](#)) | Feedback: (+5/-0) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Sat, May 14, 2011

[TarifEzaz](#)
15 posts

In your code for solving the LCS problem, you've written the following:

```

if (str1[i-1] == str2[j-1])
    lcs[i][j] = max(lcs[i][j], lcs[i-1][j-1] + 1); //handle case 3

```

Now according to the optimal substructure of an LCS, if (i-1)th and (j-1)th character matches, then lcs[i][j] could only be lcs[i-1][j]. happens because imagine: Z_k is the LCS of prefixes X_i and Y_j, where X and Y are the original sequence. Let z_k, x_i and y_j, be the kth character of Z_k, X_i and Y_j respectively. Now if z_k != x_i, then we could easily append x_i to the end of z_k to get a (k+1) length subsequence is a contradiction as we assumed Z_k to be the LCS. On the other hand if z_k = x_i, then LCS Z_(k-1) should be of length k-1 and should be constructed from X_(i-1) and Y_(j-1). If X_(i-1) and Y_(j-1) had a k-length subsequence then we would have taken that instead, causing contradiction. I'm almost copy-pasting the idea that is described in CLR book :)

So I think the max operation is redundant. Thanks for your awesome recipe!

Re: 2.4.5 Introducing Dynamic Programming (response to [post](#) by [TarifEzaz](#)) | Feedback: (+6/-0) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Sat, May 14, 2011



[syg96](#)
182 posts

Yes, you are right.

But I won't change the code. I think it is a bit simpler right now. I have described three theoretically possible cases, so let's DP the best value among all of them.

By the way, it is a very important principle of algorithmic programming competitions: the simpler the solution is, the better. But theory and implementation=)

Re: 2.4.5 Introducing Dynamic Programming (response to [post](#) by [syg96](#)) | Feedback: (+0/-1) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Thu, Feb 2, 2012

[yousuf_shawon](#)
2 posts

In a problem lexicographically smallest lcs should be found out.
If two strings s1="abc", and s2="abcacb", then possible lcs are "ac", "acc", "abc".
So lexicographically smallest is "abc".
How to find the lexicographically smallest lcs.
Please help.

Re: 2.4.5 Introducing Dynamic Programming (response to [post](#) by [yousuf_shawon](#)) | Feedback: (+0/-0) | [\[+\]](#) [\[-\]](#) | [Reply](#)

Sat, Jul 12, 2014

[rit2012008](#)
13 posts

store back-link and when you have a chance that you can go Dp[i-1][j] or Dp[i-1][j-1] then look at last used value that which state has lexicographically smallest element :D

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 how all we need to do is post...[Read More](#)

[View More](#)

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](#)

© 2014 topcoder. All Rights reserved.
[Privacy Policy](#) | [Terms](#)

Get Connected

Your email address

[Submit](#)