Convex hull trick

Recurrence:

*dp[i]=min(dp[i],dp[j]+b[j]*a[i]);*

Sample problem*: you have N cities on a straight line, you have to deliver a mail from last city to the first one as fast as possible, and also you have a mailman in every city; every mailman has his own speed and also his own penalty time which he spends on preparing to travel after receiving a mail.*

Naive implementation gives you *O(N^2)*. Expression in brackets can be considered as **linear function** on *a[i]*. This observation allows us to speed up our solution.
In case *b[j]>=b[j+1], a[i]<=a[i+1]* it is possible to reach *O(N)*. In case some of these conditions are missing you can still reach *O(N*log(N))*.

The idea is to keep **lower envelope** of given set of linear functions, described by our formula.

Considering it as geometry problem - In naive solution you are looking at *y*-values of all lines for given *x*, and you want to consider only one line, for which you know that it is best one for given position.

Easy way to store it is making a list of lines which belong to this envelope, sorted in order they appear in envelope.

In case you are adding lines in order of increasing slope - it is possible to update a tail of envelope in linear time (by removing lines which are not in envelope anymore, in linear time, and then adding a new line). In case you have *a[i]>a[i-1]*, you can keep pointer on the line which is a part of envelope at the point of current query.

To check that you have to remove last line from envelope, you should find **an intersection between new line and a line from envelope**. If intersection belongs to envelope - everything is fine, otherwise you should remove a line from envelope.

In case some conditions are missing - you need more complicated data structures (list of envelopes, pair of sets, cartesian tree or any other structure you like) to store/update envelope and binary search to answer queries.

Link to read more: http://wcipeg.com/wiki/Convex_hull_trick.

Some more problems: check provided contest :)

Knuth's optimization

Recurrence:

dp[i][j]=min(dp[i][j],dp[i][k]+dp[k][j]+cost[i][j]);

Sufficient Condition of Applicability:

Cut[i][j-1]<=Cut[i][j]<=Cut[i+1][j],

where *Cut[i][j]* is first position of *k* for which we can get best value of *dp[i][j]*.

Original problem is about building optimal binary search tree. You task is to build a binary search tree which provides the **smallest possible search time** (or expected search time) for a given sequence of accesses (or access probabilities).

Number of possible binary search trees is exponential, therefore full search isn't going to help you much. Obvious DP solution is *O(N^3)* - DP over segments of input sequence, *DP[l][r]* is answer for subarray *[l..r]*.

In 1971 Knuth provided a way to improve it to *O(N^2)*. While in original algorithm you are checking all possible positions of *Cut[l][r]*, you can actually check only some smaller range, bounded by *Cut[i][j-1]* and *Cut[i+1][j]*. In order to show that given optimization improves running time to *O(N^2)* you can **write down a telescoping sum**, where number of operations for particular state (i,j) can be described as *Cut[i+1][j]-Cut[i][j-1]+1*.

Almost all addends in this sum will be repeated **twice** - *Cut[i][j]* is used when calculating *DP[i-1][j]* and *DP[i][j+1]*, and it will be canceled (unless i=1 or j=n). Remaining part is *O(N^2)*.

When solving problems during a contest, in some cases it makes sense to make an assumption about applicability of Knuth's optimization if you are facing troubles with proving it. In this case you can easily **stress-test your solution** with naive *O(N^3)* DP.

Divide and Conquer Optimization

Recurrence:

dp[i][j]=min(dp[i][j],dp[i-1][k]+cost[k][j]);

Sufficient Condition of Applicability:

Cut[i][j]<=Cut[i][j+1]

Example of a problem: *you have to split sequence N letters into K consecutive groups for keys of cell phone keyboard. You know frequency of every letter, and also you know that you'll have to press a key x times to type a letter which is placed on x-th position on given key Your task is to minimize total number of you times you have to press a key in order to type whole text.*

Naive idea is to try all possible values of *k*. Once again, there is a way to improve it by using provided condition. Let's say we know that *Cut[10][50]* is equal to 25. Now if we are calculating *Cut[10][51]* - it makes no sense to try k=24, because we know that *Cut[10][51]>=Cut[10][50]*.

Let's extend this idea to whole algorithm. If we have to solve our problem for range *[l..r]* on level *i* - let's **divide this range in two parts** by some position *q*, and then use *Cut[i][q]* as a bound when solving range *[l..q-1]* or *[q+1..r]*.

In order to show that it improves complexity to *O(N^2*log(N))* you can write down scheme of solving particular level in a tree-style. Resulting tree will have *O(log(N))* levels, and for every level total number of operations is *O(N)*.

A problem to practice: http://codeforces.com/contest/321/problem/E