Garage Sale

Design state as $dp[mask][u]$. u means the node you currently at, the $mask$ is a set contains garages you already visited.
The state transition equation is like:

$dp[mask][j] = \max_{k \in mask} dp[mask \backslash \{j\}][k] + g[j] - d[k][j]$

The final answer will be the maximum value of $dp[mask][0]$ over all $mask$.

Please check travelling saleman problem for more.

Coin changing

The first problem is to decide if you can make a change for value v using unlimited coins of denomination $x_1, \ldots, x_n$.

Let bool variable $dp[i][j]$ means if you can make change for value j using the first i types of coins. The transition equation is:

$dp[i][j] = dp[i-1][j - x(i)] \vee dp[i][j - x(i)] \vee dp[i-1][j]$

$dp[i][j - x(i)] -> dp[i][j]$ 5 -> 10

$dp[n][v]$ will be the final answer.

Coin changing II

If you understand the last problem, this problem will be easy.

State design is the same. Let bool variable $dp[i][j]$ means if you can make change for value j using the first i types of coins. The transition equation is:

$dp[i][j] = dp[i-1][j - x(i)] \vee dp[i-1][j]$

$dp[i][j - x(i)]$ is removed since only one x_i coin can be used.

$dp[n][v]$ will be the final answer.

Coin changing III

The last problem restricts that the number of coins used is at most k.

State design is a little different. Let $dp[i][m][j]$ means if you can make change for value j using m coins from the first i types of coins. Then the transition equation is:

$dp[i][m][j] = dp[i-1][m-1][j - x(i)] \vee dp[i][m-1][j - x(i)] \vee dp[i-1][m][j]$

$dp[n][k][v]$ will be the final answer.

Vertex Cover

Please see wiki page tree for basic definitions.

We could define any node to be the root node. Visit all nodes from the root node then parent node and child node are also defined. Then we define **subtree**:

- If the node is a leaf node (has no child) then the subtree is the node itself.
- If the node has child nodes, the subtree of this node is the union of itself and all subtree of its children.

Let $dp[0][u]$ be the size of smallest vertex cover of the subtree of node u if u is not included in vertex cover, similarly let $dp[1][u]$ be the size if u is included. Then the transition equation is:

$dp[0][u] = \sum_{v \text{ is child of } u} dp[1][v]$
$dp[1][u] = \sum_{v \text{ is child of } u} \min(dp[0][v], dp[1][v])$

Since if u is not chosen, all its children most be chosen. Otherwise we can arbitrary choose or not choose all of its children.

The the minimum value of $dp[0][root]$ and $dp[1][root]$ is the answer.

3 Partition

A straightforward idea is exhaustive search: Given the current array S and current value of 3 partitions a, b and c, try to assign some element s_i to each partition then check feasibility recursively. (Like $(S \backslash s_i, a + s_i, b, c)$ and etc.)

```
function feasi(S, a, b, c)
            if (S-si, a + si, b, c)
                return True
            if (S-si,a,b+si,c)
            ...
```

The idea of dynamic programming is introduced since we might visit some state (S, a, b, c) for multiple time. It should be noted that

- if we assign values in S by some pre-defined order, then S is uniquely determined once a, b, c is given.
- Another observation is a, b, c is unordered, exchange any pair of value won't make any difference.

Then the state should be designed as sorted tuple (a, b, c). Save the feasibility when we visit any state to avoid redundant calculation.

Sequence Alignment I

This is a simple variant of the edit distance algorithm. The recursion is modified to:

$E(i, j) -> x[1...i] \ and \ y[1...j]$

$E(i, j) = \max\{E(i-1, j) + \delta(x_i, -), E(i-1, j-1) + \delta(x_i, y_i), E(i, j-1) + \delta(-, y_j)\}$

The initialization has also to be modified to deal properly with the new scoring for gaps. We have, for $i, j > 0$:

$E(0,0) = 0$
$E(i,0) = E(i-1,0) + \delta(x_i, -)$
$E(0,j) = E(0,j-1) + \delta(-, y_i)$

The correctness follows by the same argument as for the edit distance algorithm. The running time is $O(mn)$.

Sequence Alignment II

$c_0 + c_1 * k$

Compared with the last problem, it might be that an optimal alignment for $E(i,j)$ is not an extension of the optimal for

$E(i-1,j)$,$E(i—1,j-1)$, $E(i,j-1)$ as an alignment with smaller previous score but terminating with a gap might have a smaller gap penalty and beat all extensions of optimal alignments. This suggests that we should not keep a single matrix of subproblems, but 3.

E will be the matrix of subproblems where the alignments are constrained to be a substitution or a match in their last position. E_x, will be the matrix of subproblems over the alignments which have a gap in the last position of string x. E_y is defined similarly for y. Given these definitions, we just need to work out the recursion correctly case by case.

$E(i,j) = \max\{E(i-1,j-1), E_x(i-1,j-1), E_y(i-1,j-1)\} + \delta(x_i, y_j)$ $E(i,j)$ means the last char is aligned

$E_x(i,j) = \max\{E(i-1,j) - c_0 - c_1, E_x(i-1,j) - c_1, E_y(i-1,j) - c_0 - c_1\}$
$E_x(i,j)$ means there is a gap in str x

$E_y(i,j) = \max\{E(i,j-1) - c_0 - c_1, E_x(i,j-1) - c_0 - c_1, E_y(i,j-1) - c_1\}$
$E_y(i,j)$ means there is a gap in str y

The output will just be the maximum of $E(m,n)$,$E_x(m,n)$ and $E_y(m,n)$. This takes $O(mn)$ as we still have $O(mn)$ subproblems, each evaluated in constant time.

Sequence Alignment III

This only requires a simple modification to sequence alignment I. $E(i,j)$ becomes the score of the best scoring substring of $x[1,\cdots,i]$ and $y[1,\cdots,j]$. The recursion is then modified to take into account the possibility that the best local alignment be empty:

$E(i,j) = \max\{E(i-1,j) + \delta(x_i, -), E(i-1,j-1) + \delta(x_i, y_j), E(i,j-1) + \delta(-, y_j), 0\}$

The maximum element in E is the answer.

Exon Chaining

For $i \in \{1, \cdots, n\}$, let $W(i)$ be the the weight of the best subset of consistent partial matches in $x[1, \cdots, i]$. To compute $W(i)$, we consider the two following cases:

- the best subset of partial matches contains a match j with $r_j = i$,
- the best subset of partial matches does not contain such j

In the first case, $W(i)$ will be the sum of w_j and the weight of the best match on the remaining of the string, i.e. $W(l_j — 1)$. In the second case, we will just have $W(i) = W(i — 1)$. This shows that the following recursion is correct:

$W(i) = \max\left\{W(i-1), \max_{j:r_j=i}\left\{W\left(l_j - 1\right) + w_j\right\}\right\}$

The algorithm will then proceed computing $W(i)$ in ascending order of i and will output $W(n)$ as best total weight achievable. To reconstruct the actual sequence of partial matches, it suffices to keep track, for all $W(i)$ of which j maximizes the expression in the recursion, when the second maximum is the larger. We can then follow these pointers from $W(n)$ backwards to identify the optimal alignment.

The running time is $O(n + m)$, where m is the number of partial matches, as we have n subproblems and each partial match is considered once in the maximizations.

Reconstructing Evolutionary Tree by Maximum Parsimony

The very first observation is you can divide the string by single bits and find the answer for these bits individually. Then we only deal with the problem with a single character.

If node u has 2 leaf nodes 'A' and 'C', then the character on node u could be 'A' or 'C', both with cost 1. We won't fill in another character here because it won't make the cost lower even if we consider the parent node of u. If the characters of 2 leaf nodes are equal then just simply assign this character to node u without any cost.

We start at the root node r (actually root node could be any node but leaf node), find the possible character set $S(a)$ and $S(b)$ for 2 children of r recursively.

- if $S(a) \cap S(b) \neq \varnothing$, then $S(r) = S(a) \cap S(b)$ with no extra cost.
- $S(a) \cap S(b) = \varnothing$, then $S(r) = S(a) \cup S(b)$ with 1 cost.

$S(r)$ won't contain some character which is not in $S(a)$ or $S(b)$, the reason is explained above, it won't make the cost lower.