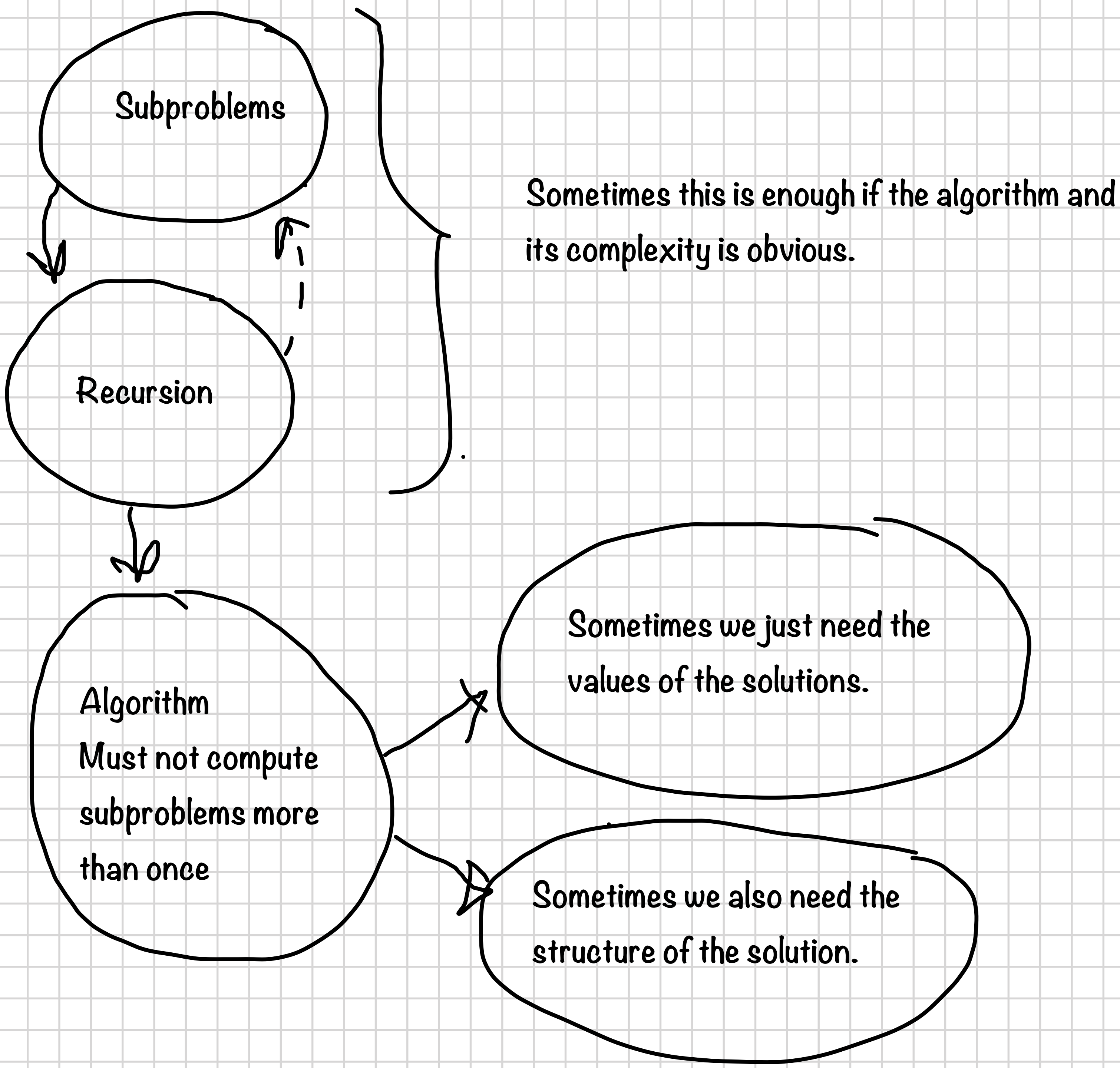
Dynamic Programming cont.

We repeat: The Dynamic Programming Template has three parts.



Let us return to the shortest path problem. Is Dijkstra's algorithm a DP-algorithm? We have subproblems
d[u] = length of shortest path from s to u.
We have a type of recursion
d[v] = d[u] + w[u,v]
The problem is that we don't have a simple way of ordering the subproblems. In that sense, Dijkstra's algorithm isn't a true DP-algorithm.

If we have a directed graph with no cycles (A DAG = Directed Acyclic Graph) things are simpler. In a DAG we can find a so called Topological Ordering.

Topological Ordering: An ordering of the nodes such that
(v[i], v[j]) is an edge $\Rightarrow$ i < j
A topological ordering can be found in time O(| E |) (See textbook).

Let's assume that the start node is v[1]. Set w[i,j] = ∞ if there is no edge (v[i], v[j]). Then

$$\begin{cases} d[1] = 0 \\ d[k] = \min_i \left(d[i] + w[i,k] \right) \quad 1 \le i \le k \end{cases}$$

The algorithm runs in $O(n^2)$

Matrix Chain Multiplication

We want to compute the product of two matrices A and B. A is a $p \times q$- matrix and B is a $q \times r$-matrix. The cost (number of products of elements) is pqr.

Let us assume that we want to compute a chain of matrices. We want to find the best way to multiply them. If we have three matrices A, B, C then we know from the associative law of multiplication that (AB)C = A(BC). But the costs of computing the product will normally differ!

If we have a chain of matrices M[1] M[2] ... M[n] what is the best way of computing the product?

Subproblems:
Set c[i,j] = smallest possible cost of computing M[i] M[i+1] ... M[j].

Recursion:

Let us first assume that the matrices have dimensions d[0]×d[1], d[1]×d[2], ... ,d[n-1]×d[n].

$$\begin{cases} c[i,i] = 0 \text{ for all } 1 \le i \le n \\ c[i,j] = \min\limits_{k} \left(c[i,k] + c[k+1,j] + d[i-1]d[k]d[j] \right) \text{ where } i \le k < j \end{cases}$$

Why?

(M[i] M[i+1] ... M[k]) (M[k+1] ... M[j])

c[i,k] operations
d[i-1]×d[k] - matrix

c[k+1, j] operations
d[k]×d[j] - matrix

d[i-1] d[k] d[j] operations

All together: c[i,k] + c[k+1] + d[i-1] d[k] d[j] operations

Now we have to find an algorithm using the recursion. Essentially we have to find suitable loops. We can try to first compute all c[i,j] with |j-i| = 1, then with |j-i| = 2 and so on. If we do this we are able to use the recursion formula.

```
For i ← 1 to n
    c[i,i] ← 0
For diff ← 1 to n-1
    For i ← 1 to n - diff
        j ← i + diff
        min ← c[i+1,j] + d[i-1] d[i+1] d[j]
        best_k ← i
        For k ← i+1 to j - 1
            If min > c[i,k] + c[k+1,j] + d[i-1] d[k] d[j]
                min ← c[i,k] + c[k+1,j] + d[i-1] d[k] d[j]
                best_k ← k
        c[i,j] ← min
```

The value of c[1,n] gives the minimum number of operations.

The complexity is $O(n^3)$

Editing distance and sequence alignment

We have two strings x[1], x[2], ... , x[m] and y[1], y[2], ... , y[n]. We want to align them so the number of positions where the alligned sequences are different is . minimal. We are allowed to put gaps into the sequences.

Ex: The sequences EXPONENTIAL and POLYNOMIAL can be aligned as

EXPO_ _NENTIAL
_ _POLYNOM_IAL

Let
D[p,q] = distance of best alignment of a[1], ..., a[p] and b[1], ..., a[q]
We measure distance by adding a number α for each match between a character and a blank and adding β for a match between two different characters.

Then we get the recursion formula

$D[p,0] = \alpha p \quad D[0,q] = \alpha q$ for all p,q

$D[p,q] = \min(D[p,q-1] + \alpha,\ D[p-1,q] + \alpha, D[p-1,q-1] + \beta\, \mathrm{diff}[a[p],b[q]])$

if $p>1$ and $q>1$

Pretty Print

We have a set of n words. They have lengths l[i] (number of characters). We want to print them on a page. Each line on the page contains space for M characters. There must be a space 1 between each pair of words.

Set $s[i,j] = \sum_{k=i}^{j} l[k] + j - i$.

This will be the number if characters left on the line if the words i to j are put on the line. Let E = M - s[i,j] be the excess of space on the line. We want to put the words (in correct order) on lines so that the excesses are as small as possible. We can use a penalty function f( ) and try to make a split of the words such that f(E1) + f(E2) + ... i.e. the sum of the penalties from the lines is as small as possible.

It's natural to use the Last Line Excluded rule (LLE), i.e. we give no penalty for excess on the last line.

We now want to find the best way to arrange the words. It's simplest to first ignore LLE.

Let w[k] = least penalty when using the first k words and not using LLE.

Recursion:

w[0] = 0

$w[k] = \min_i (w[i-1] + f(M - s[i,k]))$ where the min is taken over all $1 \le i \le k$ such that $s[i,k] \le M$

To get the solution with LLE we compute

$\min_j w[j]$ such that $s[j+1, n] \le M$

We will return to the Subset Sum problem once more. Remember that we defined

v[i,m] = 1 if there is a subset of a[1], a[2], ... , a[i] with sum m and v[i,m] = 0 otherwise.

We got the recursion formula

$v[1, 0] = 1$

For all i such that $2 \leq i \leq n$ and all m M such that $a[i] \leq m$

$v[i,m] = \max(v[i-1,m], v[i-1,m-a[i]])$

In lecture 5 we gave an algorithm that solved the problem. It's possible to give a recursive algorithm as well. A first try could look like:

```
vrek[i,m] =
    If m < 0
        Return 0
    If m = 0
        Return 1
    If i = 1 and m = a[1]
        Return 1
    If vrek[i-1, m] = 1
        Return 1
    If vrek[i-1, m-a[i]] = 1
        Return 1
    Return 0
```

We make the call vrek[n,M] to get the answer.

But this solution is no good. The problem is that the algorithm uses repeated calls to subproblems that already have been solved.

To get a better algorithm will have to keep track of all computed values of subproblems. To do this, we use an array comp[i,m],

.

.

```
Set all comp[i,j] to FALSE
Set all v[i,j] to 0
vrek[n,M]

vrek[i,m] =
    If comp[i,m]
        Return v[i,m]
    If m < 0
        Return 0
    If m = 0
        Return 1
    If vrek[i-1, m] = 1
        comp[i,m] ← TRUE
        v[i,m] ←1
        Return 1
    If vrek[i-1, m-a[i]] = 1
        comp[i,m] ← TRUE
        v[i,m] ←1
        Return 1
    comp[i,m] ←TRUE
```

This technique of remembering already computed values is called Memoization. Sometimes it can be useful, but in most cases the bottom-up method should be preferred.

A simpler Subset Sum problem

One thing that makes the original Subset Sum problem hard is that we are allowed to use each number just once. If we can use the numbers multiple times we get a simpler DP-problem.

Set v[m] = 1 if we can get m as a subset sum and 0 otherwise.

Then we can compute the values by

$$\begin{cases} v[m] = 0 \text{ for all } m < 0 \\ v[0] = 1 \\ v[m] = \max_k (v[m-a[k]]) \quad 1 \le k \le n \end{cases}$$