Lecture 4 Dynamic Programming

Last update: Jan 19, 2021

References:

- Algorithms, Jeff Erickson, Chapter 3.
- Algorithms, Gopal Pandurangan, Chapter 6.

Dynamic Programming

Backtracking is incredible powerful in solving all kinds of hard problems, but it can often be very slow; usually exponential.

Example: Fibonacci numbers is defined as recurrence:

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$$

A direct translation into recursive program to compute Fibonacci number is

```
RecFib(n):
    if n=0 return 0
if n=1 return 1
return RecFib(n-1) + RecFib(n-2)
```

Fibonacci Number

The recursive program has horrible time complexity. How bad? Let's try to compute. Denote $T(n)$ as the time complexity of computing RecFib(n). Based on the recursion, we have the recurrence:

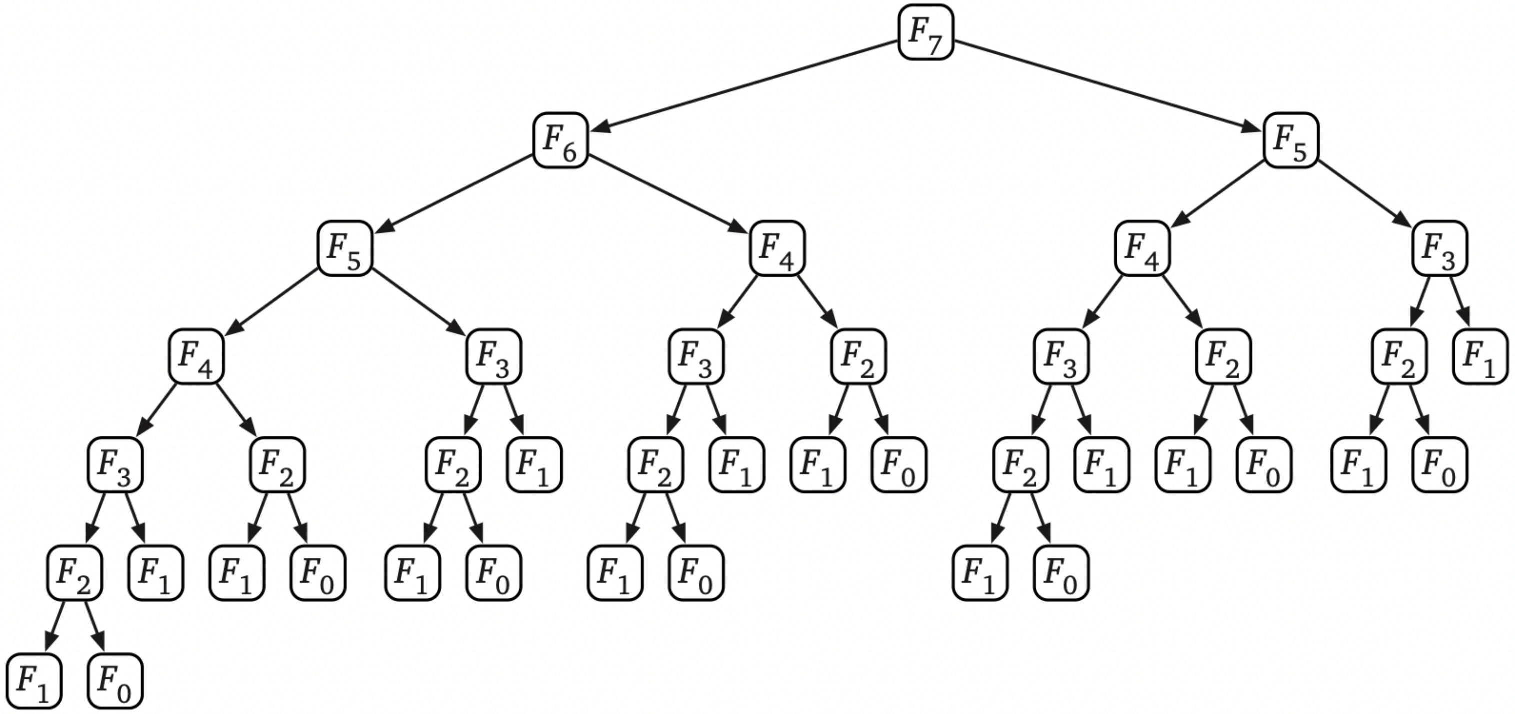
$$T(n) = T(n-1) + T(n-2) + 1, \quad T(0) = T(1) = 1$$

Solving this recurrence, we get

$$T(n) = O(\Phi^n), \quad \Phi = \frac{\sqrt{5}+1}{2} \approx 1.618$$

So the RecFib(n) program runs at exponential time complexity.

RecFib Recursion Tree



Intuitively, why RecFib() runs exponentially slow. Problem: redundant computation! How about memorize the intermediate computation result to avoid recomputation?

Fib: Memoization

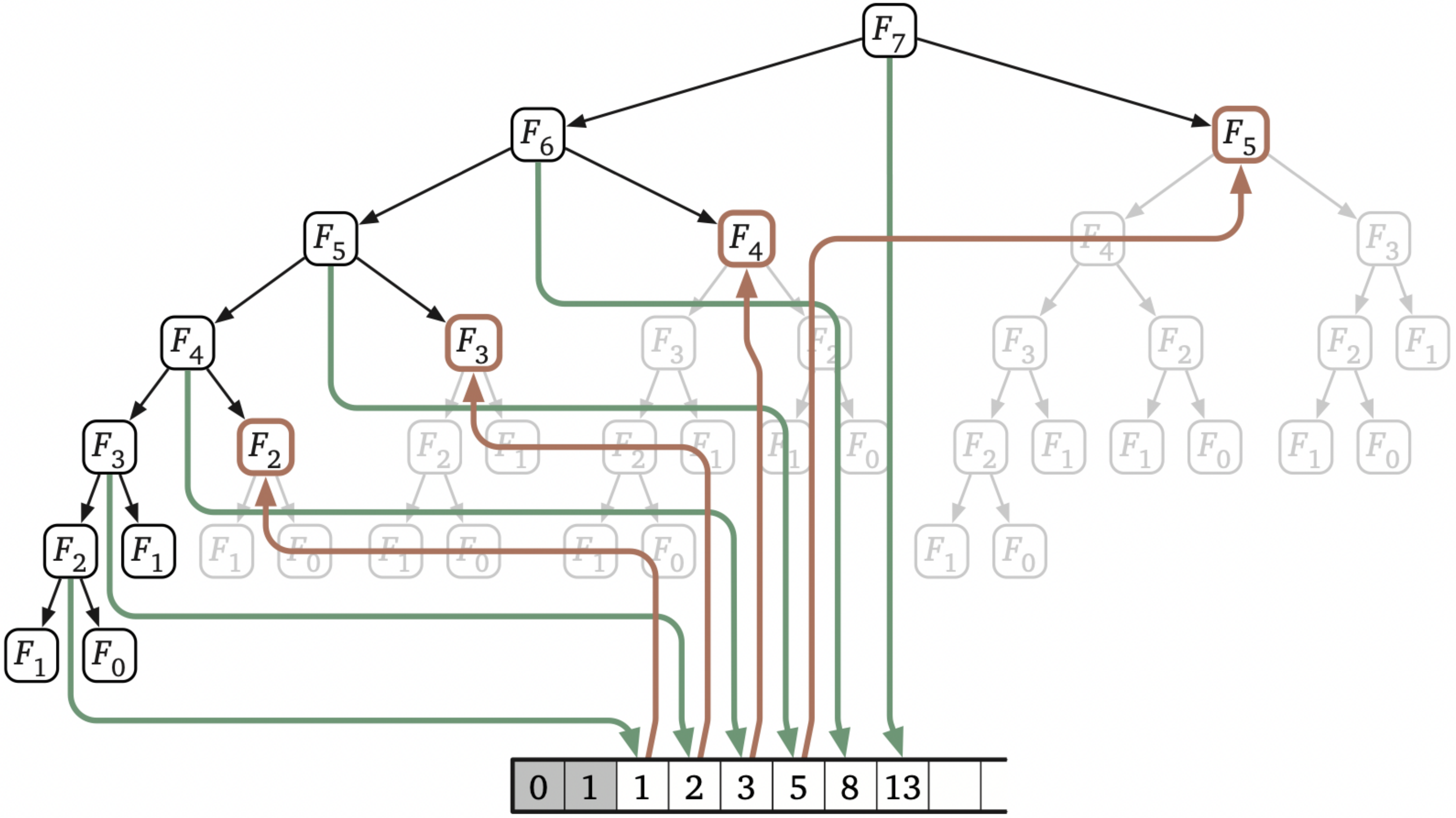
To optimize the performance of RecFib, we can memorize the intermediate F_n into some kind of cache, and look it up when we need it again.

```
MemFib($n$):
    if $n = 0 \,||\, n = 1$
        return n
    if F[$n$] is undefined
        F[$n$]$\leftarrow$MemFib($n$-1)+MemFib($n$-2)
        return F[$n$]
```

How much does it improve upon RecFib()? Assuming accessing F[n] takes constant time, then at most n additions will be performed (we never recompute).

So the time complexity is O(n), a huge improvement.

MemFib Recursion Tree



Dynamic Programming

Once we see how the array F[*1..n*] is filled, we can **intentionally** fill the array **in order**, instead of relying on the recursion.

```
IterFib($n$):
    F[0] $\leftarrow$ 0
    F[1] $\leftarrow$ 1
    for $i \leftarrow 1$ to n
        F[$i$] $\leftarrow$ F[$i$-1] + F[$i$-2]
    return F[$n$]
```

This is a dynamic programming (DP) algorithm: basically recursion, but with intentional evaluation order, usually filling out a table systematically.

Text Segmentation Again

In backtracking lecture we have developed recursive algorithm to solve the text segmentation problem, which has worst case time complexity of $O(2^n)$. Let's review the recursion. The key idea is to define:

$\boxed{\text{Splittable}(i) \text{ is } \textbf{true} \text{ iff } A[i..n] \text{ can be segmented into words.}}$

Following this definition, we have the recursion:

$$\text{Splittable}(i) = \begin{cases} \textbf{true} & \text{if } i > n \\ \bigvee_{j=i}^{n} (\text{IsWord}(i,j) \wedge \text{Splittable}(j+1)) & \text{otherwise} \end{cases}$$

where $\text{IsWord}(i,j)$ is short for IsWord(A[i..j]).

The directly translated recursive algorithm has worst case time complexity of $O(2^n)$.

But note that,

- there are only n possible distinct way of calling Splittable(i);
- there are only n^2 possible distinct way of calling IsWord(i, j)

(All the work is actually done in IsWord()).

How come we have worst case time complexity of $O(2^n)$? We must be calling the same Splittable(i) (and as a result, the same IsWord(i, j)) over and over again.

What if we save the results of computed Splittable(i) in a table, so that we can do lookup instead of recomputation?

Let's make a table SplitTable[*1..n+1*].Each SplitTable[i] only depends on the elements SplitTable[j] with $j > i$.

So we can fill up the table from right to left, or end to beginning.

```
FastSplittable($A[1..n]$):
    SplitTable[$n+1$] $\leftarrow$ **true**
    for $i \leftarrow n$ down to 1
        SplitTable[$i$] $\leftarrow$ **false**
        for $j \leftarrow i$ to $n$
            if IsWord($i, j$) and SplitTable[$j+1$]
                SplitTable[$i$] $\leftarrow$ **true**
    return SplitTable[1]
```

Now what's the time complexity?

To summarize, we did achieve tremendous speedup over the naive recursive algorithm, by **intentionally** evaluate the recursive calls **in-order**.

We turn recursive function calls into filling tables!

Let's see how the implemented TextSegment works.

I've taken the text of the story Cinderalla and remove all space, os it looks something like:

SnowWhiteandtheSevenDwarfsOnceuponatimeinagreat-castleaPrincesdaughtergrewuphappyandcontentedinspiteofajeal-ousstepmother...

In total 10647 characters.

- Backtracking takes 8442 milliseconds
- DP takes 29 milliseconds

See code and associated dictionary/text files at course webpage.

The Pattern: Smart Recursion

Dynamic programming is not about filling in tables; it's about smart recursion!

(although it does end up being filling tables...)

How to develop dynamic programming algorithms? Two steps:

1. **Formulate the problem recursively**. Write down a recursive formula of the whole problem, in terms of the **answers** of smaller sub-problems. This is the hard part

   i. **Specification**. Describe the problem that you want to solve in coherent and precise English—now *how* to solve, but *what* to solve; without this step, it's impossible to determine the correctness of your solution

   ii. **Solution**. Give a clear recursive formula for the whole problem in terms of the answers to smaller instances of the exact same problem.

2. **Build solutions to your recurrence from the bottom up**. Write an algorithm that starts with base case, and works its way up to final solution, by considering the intermediate subproblems in correct order.

   i. **Identify the subproblems**. How can the recursive algorithms call itself, with what parameters?

   ii. **Choose a memoization data structure**. Usually a table (multi-dimensional array) that contains the solutions to every sub-problems identified.

   iii. **Find a good evaluation order**. Order the subproblems so that each one comes after all the subproblems it depends on.

iv. **Analyze space and running time**.

v. **Write down the algorithm.**

Longest Increasing Subsequence

We consider another problem we solved via backtracking, the Longest Increasing Subsequence (LIS) problem.

Problem. Given an array of numbers $A[1 \ldots n]$, compute the length of its longest increasing subsequence.

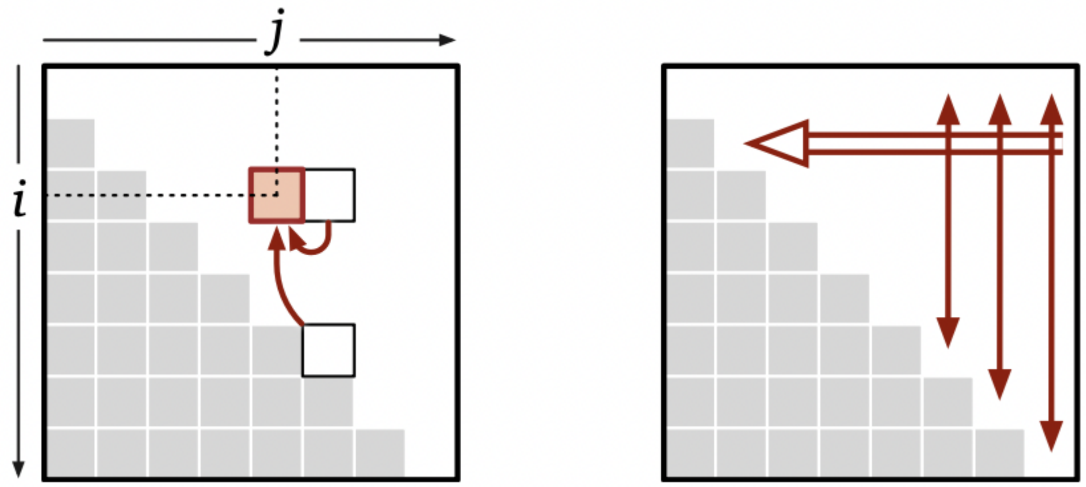
Our backtracking solution is based on the recursion:

$L(i, j)$ denotes the length of LIS of array $A[j \ldots n]$, **with every element bigger than $A[i]$**.

$$L(i,j) = \begin{cases} 0 & \text{if } j > n \\ L(i, j+1) & \text{if } A[i] > A[j] \\ \max\{L(i,j+1), 1 + L(j, j+1)\} & \text{otherwise} \end{cases}$$

Naive recursion algorithm has worst case time complexity of O(2^n), but note that we have only n^2 distinct $L(i,j)$ to begin with.

We can fillup the $L(i,j)$ table in **some order**. What order do we follow? We look at the dependencies. It seems $L(i,j)$ depends on possibly $L(i,j+1), L(j,j+1)$. To illustrate:



From the left figure, we immediately see that all the dependencies are on the right. So if we fill the table from right to left, column by column, we would satisfy all the dependencies.

To put it into algorithm:

```
FastLIS($A[1..n]$):
    $A[0] \leftarrow -\infty$
    for $i \leftarrow 0$ to n
        $L[i, n+1] \leftarrow 0$
    for $j \leftarrow n$ down to 1
        for $i \leftarrow 0$ to $j - 1$
            keep $\leftarrow 1 + L[j, j+1]$
            skip $\leftarrow L[i, j+1]$
            if $A[i] \geqslant A[j]$
                $L[i, j] \leftarrow$ skip
            else
                $L[i, j] \leftarrow \max\{\text{keep}, \text{skip}\}$
    return $L[0, 1]$
```

What's the time/space complexity?

That's not the only possible recursion or solution. Let's consider the following definition:

$L[i]$ denotes the length of Longest Increasing Subsequence (LIS) of $A[i \ldots n]$ **that starts with $\boldsymbol{A[i]}$**

Let's try reducing the problem of $L[i]$. By definition, the LIS starts with $A[i]$. What about the rest of the sequence? The rest must starts with some $A[j]$ (which is bigger than $A[i]$), and must be LIS of $A[j \ldots n]$ (otherwise, $A[i] + \text{rest}$ wouldn't be longest). So we have the following recursion:

$$L[i] = 1 + \max\{L[j] \colon j > i \text{ and } A[j] > A[i]\}$$

Now we only need to fill a one-dimensional array $L[i]$. The order is simple: we just fill it from right to left.

OK, let's consider a variant. What if we want to know the the Longest Increasing Subsequence itself?

Our recursion array $L[i]$ only encodes the length. We need another data structure to record the subsequence itself.

We need somekind of *breadcrumb* to signal path toward optimal length, so that we can trace back the subsequence itself.

Let's introduce array $B[i]$, which gives us:

$$B[i] = \text{argmax}_j \{L[j]: j > i \text{ and } A[j] > A[i]\}$$

basically B[i] records the optimal j in the recursion. From $B[i]$ we can traceback the optimal solution corresponding to the length in $L[1]$.

Let's look at an example how this works.

i															
A[i]	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9
T[i]															
B[i]															

Edit Distance

The **edit distance** between two strings is the minimum number of letter insertion, letter deletion, and letter substitution required to transform one string into the other.

E.g., the edit distance between `FOOD` and `MONEY` is at most 4:

$$\underline{\mathtt{F}}\mathtt{OOD} \longrightarrow \underline{\mathtt{M}}\mathtt{OOD} \longrightarrow \mathtt{MON}_{\wedge}\mathtt{D} \longrightarrow \mathtt{MONE}\underline{\mathtt{D}} \longrightarrow \mathtt{MONEY}$$

Four steps (subst, subst, insert, and subst) transforms `FOOD` to `MONEY`. It's sometimes useful to align the two words:

```
F O O   D
M O N E Y
```

Different letters in the same column means subst; gap in the first word means insertion; gap in second means deletion.

It's easy to see in this case, no less than 4 steps are needed, so the edit distance between FOOD and MONEY is 4.

In longer string pairs, it's not obvious what the edit distance is. For example:

```
A L G O R   I   T H M
A L   T R U I S T I C
```

This has 6 steps. Is this the minimum number of transformation that is needed? It's hard to say. Let's come up with an algorithm to tell us the edit distance (the minimum number of transformation needed to between any two strings).

How to approach this problem (finding optimal solution)? Let's try reduction, which means we take a small step and defer to a smaller problem.

Suppose the gap representation (last page) represents the shortest edit distance, then we have:

optimal substructure: if we remove the last column, then the remaining columns (prefixes) must represent the shortest edit distance between them too.

It can be easily proved by contradiction; namely, if there are shorted edit distance representation for the prefixes, we add the last column, then we would get a shorter edit distance than we have. This contradicts the assumption.

If we can figure out what to do with the last column, then we can let the "recursion fairy" to solve the prefixes. Recursion!

$\boxed{\text{Define } E[i,j]\text{: the edit distance between } A[1\ldots i] \text{ and } B[1\ldots j].}$

We want to compute $E[m,n]$, for the given strings $A[1...m], B[1...n]$.

We consider the last column in the gap representation of the shortest edit distance. There are three possibilities:

- Insertion:

ALGORITHM	
ALTRUISTI	C

  in which case: $E[i,j] = E[i,j-1]+1$

- Deletion

ALGORITH	M
ALTRUISTIC	

  in which case $E[i,j] = E[i-1,j]+1$

- Substitution:

`ALGORITH`	`M`
`ALTRUISTI`	`C`

in which case $E[i,j] = E[i-1,j-1]+1$

(or $E[i,j] = E[i-1,j-1]$, if the last characters are the same)

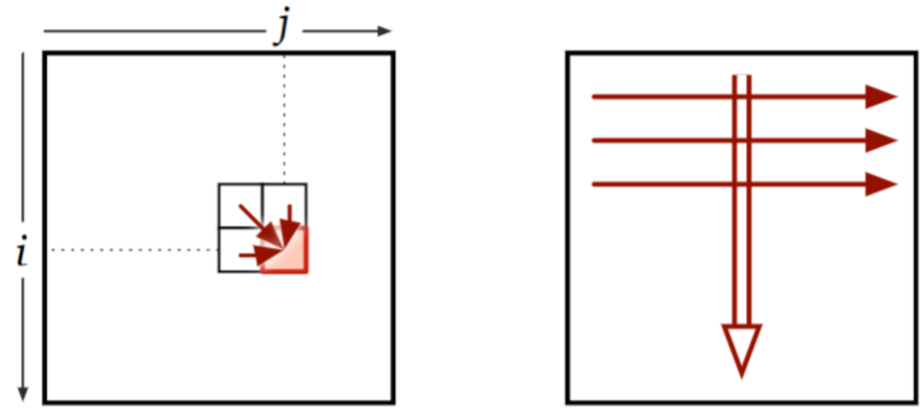
The base cases are simple: $E[0,j] = j$, $E[i,0] = i$.

To summarize the Edit function has the recurrence:

$$E[i,j] = \min \begin{cases} E[i,j-1]+1 \\ E[i,j-1]+1 \\ E[i-1,j-1]+[A[i] \neq B[j]] \end{cases}$$

Let's turn this recursion into dynamic program, using our mechanical recipe.

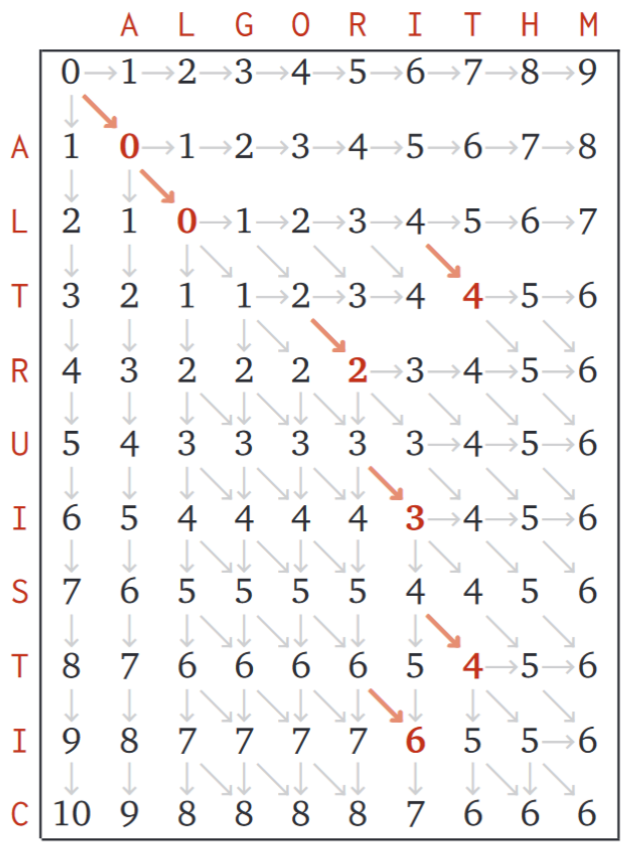
- **Subproblem**: each recursive subproblem is identified by two indices $0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n$.
- **Memoization**: we can put all possible $E[i,j]$ into a two dimensional array $E[0 \ldots m, 0 \ldots n]$.
- **Dependencies & evaluation order**:



each $E[i,j]$ depends on its north, west, and north-west neighbors. So we can evaluate left to right, top to bottom.

- **Space and Time:** Space and time complexity is $O(mn)$

The memoization table for the input string `ALGORITHM` and `ALRUISTIC` is shown below:



		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

Arrow: operations. Horizontal=deletion, vertical=insertion, diagonal=substitution.

Bold red arrow is free substitution.

Any path from the top-left corner to bottom-right corner is a optimal edit sequence.

How to reconstruct?

Subset Sum

We developed recursive algorithm for subset sum problem, which can be formulated like this:

$$S[i,t] = \textbf{true} \text{ iff some subset of } A[i..n] \text{ sums to } t$$

We need to compute $S[i,t]$. This function satisfies the following recurrence:

$$S[i,t] = \begin{cases} \text{true} & \text{if } t = 0 \\ \text{false} & \text{if } i > n \\ S[i+1,t] & \text{if } t < X[i] \\ S[i+1,t] \vee S[i+1,t-A[i]] & \text{otherwise} \end{cases}$$

Let's follow the mechanical recipe to turn this recursion into dynamic program:

- Subproblem:
- Data structure:
- Dependency & Evaluation order
- Space and Time

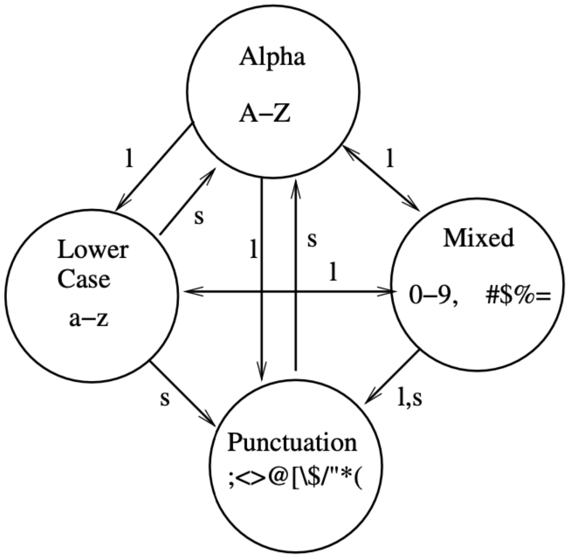
Story: Text Compression for Bar Codes

(Algorithm Design Manual, Skiener, Chapter 8.9)

Symbol Technology invents a new bar code scheme PDF-417 that can encode around 1kb text, and it looks something like this:

Figure 1. Gettysburg Address in PDF-417

To maximize the texts stored in the bar code, Symbol Techology groups characters into four modes, and **in each mode** only 5 bits are needed to encode a character. To switch to a different mode, a 5 bits command is needed to change the mode. There are two mode change command: **shift** and **latch**. Shift changes mode temporarily for the next character only, and will immediately change back to original mode. Latch will permanently change mode, until next mode command.



Now, because there are two mode change command, there are multiple possible encodings for the same text string. The problem is

$$\boxed{\text{What's the shortest encoding for a given text string? In other words, the least amount of mode change commands.}}$$

Some heuristics and greedy algorithm: if a single punctuation is between text, then maybe shift is the best; otherwise maybe latch is better.

Can we find the optimal encoding?

We could try backtracking to emumerate all encodings, but that costs exponential.

Is it possible to devise a dynamic programming algorithm then?

The key insight is the definition:

Let $M[i,j], 1 \leqslant j \leqslant 4, 1 \leqslant i \leqslant n$ denote the shortest encoding of first i characters **ending in mode *j***

Coming up with the "first i characters" is common in dynamic programing, but the additional constraint "ending in mode j" is creative, and necessary for recursion.

Following the definition, we can easily find the recurrence:

$$M[i,j] = \min_{1 \leqslant m \leqslant 4} \{M[i-1,m] + c(S_i, m, j)\}$$

where $c(S_i, m, j)$ is the cost of encoding the i-th character S_i with mode changing from m to j.

This solves the optimal encoding in linear time! It turns out that the optimal encoding saves on average 8% space compared to greedy.

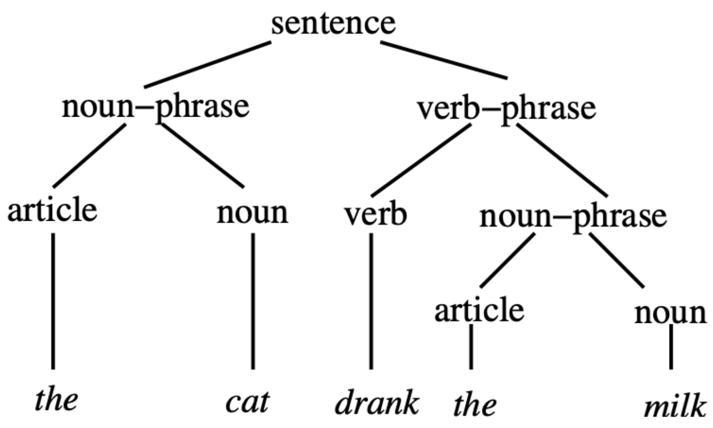
Parsing Context-Free Grammar

Problem: given a string, parse it into a *parse tree* according to a given context-free grammar.

Context free grammar and parse tree:

each rule/production defines an interpretation for the named symbol on the left, as a sequence of symbols on the right. Symbols could be terminal or non-terminal.

sentence ::= noun–phrase
verb–phrase
noun–phrase ::= article noun
verb–phrase ::= verb noun–phrase
article ::= *the, a*
noun ::= *cat, milk*
verb ::= *drank*



Some simplifying assumptions:

- the string is length n and grammar is constant size
- the string is already tokenized; i.e. each character is a *token*. (so it has n tokens)
- Each rule is of Chomsky normal form: either two nonterminals $X \to YZ$, or one terminal symbol $X \to \alpha$.

Stop and think. How to construct the parse tree? Let's try recursion.

Key: The root of parse tree ($X \to YZ$) splits S at position i, then Y must generate the left substring $S[1 \dots i]$, and Z must generate right substring $S[i+1 \dots n]$.

This motivates the following definition:

Boolean $M[i, j, X]$ is true iff substring $S[i \ldots j]$ is generated by nonterminal X.

Immediately following this definition we find the recurrence:

$$M[i, j, X] = \bigvee_{(X \to YZ) \in G} \left(\bigvee_{i=k}^{j} M[i, k, Y]\, M[k+1, j, Z] \right)$$

Space complexity?

Time complexity?

Parsimonius Parserization. Programs often contain minor syntax errors. Given context free grammar G and string S, find the minimal number of character substitutions to make S acceptable by grammer G.

This seems quite hard. However, just by changing the definition of the $M[i,j,X]$ a bit:

Integer $N[i, j, X]$ is minimum number of changes to substring $S[i \dots j]$ such that it is generated by nonterminal X.

The recurrence?

$$N[i,j,X] = \min_{(X \to YZ) \in G} \left(\min_{i=k}^{j} N[i, k, Y] + N[k+1, j, Z] \right)$$

Regular Expression Matching

(Leetcode 10) Given an input string (`s`) and a pattern (`p`), implement regular expression matching with support for `'.'` and `'*'` where:

- `'.'` Matches any single character.
- `'*'` Matches zero or more of the preceding element.

The matching should cover the **entire** input string (not partial). E.g.

```
Input:
 s = "aab", p = "c*a*b"
Output:
 true
Explanation: c can be repeated 0 times, a can be
repeated 1 time. Therefore, it matches "aab".
```

At this point, seeing the left-right structure we should immediately think in terms of dynamic programming, and set out to find a recurrence. First try:

Let $T[i, j]$ denotes whether substring $s[1 \dots i]$ matches pattern $p[1 \dots j]$.

Now reducing the problem $T[i, j]$ to a smaller one. Look at the last character of string, $s[i]$. For $T[i, j]$ to be true, there are three scenarios:

1. $T[i-1, j-1] = \text{true}$, and $s[i]$ matches $p[j]$.
2. $T[i, j-1] = \text{true}$, and $p[j]$ is some * which matches empty.
3. $T[i-1, j] = \text{true}$, and $p[j]$ continues to match $s[i]$. (how do we check this?)

What are the base cases?

Exercise: **Wild Card Matching (leetcode 44)**

Given an input string (`s`) and a pattern (`p`), implement wildcard pattern matching with support for `'?'` and `'*'` where:

- `'?'` Matches any single character.
- `'*'` Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

The Ordered Partition Problem

Problem: integer partition without rearragement
Input: An arrangement S of non-negative numbers $s_1, \ldots, s_n$ and integer k
Output: Partition S into k or fewer ranges, to minimize the maximum sum over all the ranges, without reordering any of the numbers.

How to solve? If we think in terms of recursion, then we can look at the last partition.

Matrix Chain Multiplication

Problem: Given a chain of matrices: $(A_1, A_2, \ldots, A_n)$ where matrix A_i has dimension $p_{i-1} \times p_i$. We are interested in finding the fastest way to compute the product of the chain of matrices: $A_1 A_2 \ldots A_n$:

- Find a way to parenthesize the product $A_1 A_2 \ldots A_n$ to minimize the number of multiplication.
- Two matrix $A_i A_{i+1}$ costs $p_{i-1} p_i p_{i+1}$ number of multiplications.
- Matrix multiplication is associative: $(AB)C = A(BC)$.

Example: three matrices: A_1,A_2, A_3 are of sizes 10x1000, 1000x100, 100x200 respectively. What's the costs of two ways of parenthesizing them?

Backtracking solution: how many ways to parenthesize n matrices? It's at least exponential. In fact, assume that $P(k)$ denotes the number of different ways to parenthesize product of k matrices, then

$$P(n) = \sum_{k=1}^{n-1} P(k)\, P(n-k)$$

We don't know exactly how to solve this recurrence, but we can easily show that it's at least exponentional (by mathematical induction)

$$P(n) = \Omega(2^n)$$

```
BestMatrixChainMulti($P[1..n]$):
    for $k \leftarrow 1$ to $n - 1$
        $B_1 \leftarrow$ BestMatrixChainMulti($P[1 \ldots k]$)
        $B_2 \leftarrow$ BestMatrixChainMulti($P[k + 1 \ldots n]$)
        $C_k \leftarrow B_1 + B_2 + p_0 \, p_k \, p_n$
    return the smallest $C_k$
```

This is a recursive algorithm based on divide and conquer. Can we optimize it with dynamic programming? Do you note a lot of redundant computations?