Lecture 6. Dynamic Programming

Fibonacci numbers

$F(0) = 0$

$F(1) = 1$

$F(n) = F(n-1) + F(n-2)$

A recursive algorithm

```
function  fib-recur(n)

   if n=0: return 0

   if n=1: return 1

   return  fib-recur(n-1)+fib-recur(n-2)
```

Run time?

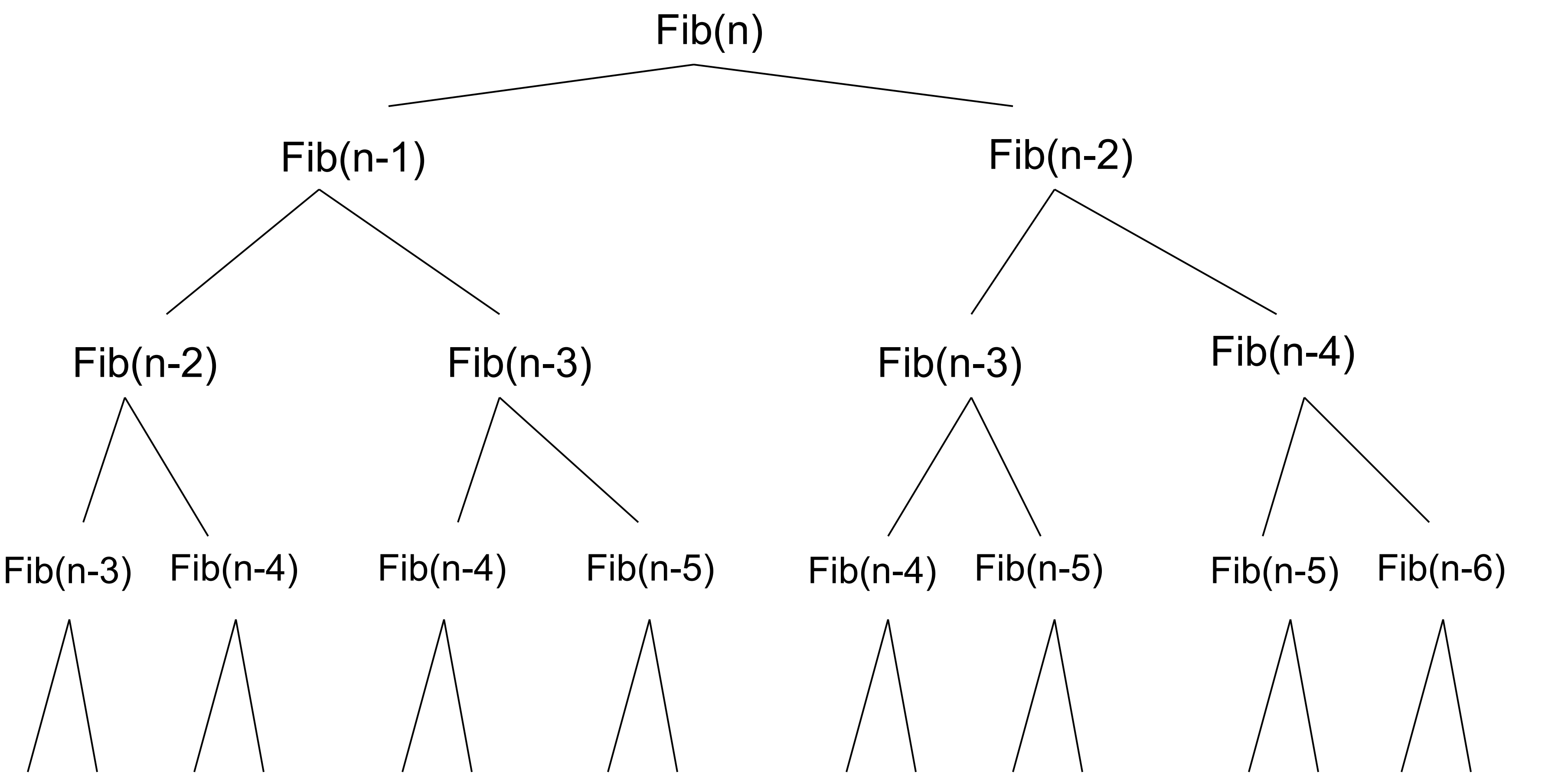
Recurrence relation

$$T(n) = T(n-1) + T(n-2) + c$$

$$O(2^n), \Omega\left(2^{\frac{n}{2}}\right)$$

Why so slow?

- Repeated computation



Avoid repeat by memoization

Memoization: a speed up technique that stores the results of expensive function calls and returns the cached result

```
fib-mem(n)
     if n<2 F(n)=n
     else if F(n) is undefined
          F(n)=fib-mem(n-1)+fib-mem(n-2)
     return F(n)
```

Runtime?

Bottom-up: iterative Version

```
function  fib-iter(n)
     f[0]=0;  f[1]=1;
      for i=2 to n
      f[i]=f[i-1]+f[i-2];
      return f[n];
```

Runtime?

Dynamic Programming

- A powerful algorithm design technique
  - Very common interview questions !!!
- Many applications. For example:
  - Unix diff for comparing two files
  - Bellman-Ford for shortest path routing in networks
  - CKY algorithm for natural language parsing
  - ……
- Coined by Richard Bellman before the age of computer programming
  - Dynamic Programming = planning over time

When to use Dynamic Programming?

- When your problem has the following properties:
  - **Optimal sub-structures**: solution to a problem can be defined using solutions of smaller sub-problems (similar to Divide and Conquer)
  - **Overlapping Sub-problems** (a key difference from divide and conquer): we see repeated sub-problems, thus important to avoid repeatedly solving the same sub-problems

Dynamic Programming: key steps

1. Figure out how to get the solution to a problem based on solutions to smaller sub-problems
   - This naturally will require you to first define the subproblems, which may not be obvious
   - Pretend you have a solver but can only be used to solve smaller problems
     - e.g., F(n)= F(n-1)+F(n-2)
   - **This is usually the most challenging and creative step**
2. Start from the smallest problems and build up solutions to larger problems – bottom up, iterative
   - Sometimes recursion is used with memoization

3. Sometimes we need to keep track or retrace the chain of solutions to construct the final solution

Longest Increasing Subsequences

Problem:

Given a sequence of numbers $a_1, a_2, \ldots, a_n$, find the longest increasing subsequence(LIS)

$$5 \quad 2 \quad 8 \quad 6 \quad 3 \quad 6 \quad 9 \quad 7$$

5 8 9
2 6 9
2 3 6 7

All three are increasing subsequences
Goal: find the longest one

- Don't need to be contiguous
- May not be unique

5 2 8 6 3 6 9 7

- Q1: what is the longest increasing subsequence if we must end the sequence with 7?
- A1: we don't know, but the number before 7 must not be 8, or 9

- Q2: what could the previous number be?
- A2: any number < 7

- Q3: if you have a solver that tells you the longest increasing subsequence ending at all previous positions, can you figure out the answer to Q1?

Building our solution

Let L[i] be the length of a longest increasing subsequence ending at position i

$$L[1] = 1$$

$$L[i] = \max_{j:1 \le j < i, a_j < a_i} L[j] + 1 \text{ for } i = 2, \dots, n$$

Overall solution: $\max_i$ L[i]

Example

5	2	8	6	3	6	9	7

Iterative algorithm

```
Longest_Increasing_Seq (A,n)
L[1]=1
for i=2 to n:
   L[i]=1
   for j=1 to i-1:
      if a_j < a_i and L[i] < L[j]+1:
         L[i]= L[j]+1
Lis_max=1
for i=1 to n:
   if L[i]>Lis_max  Lis_max = L[i]
Return Lis_max
```

$$L[i] = \max_{j:1 \le j < i, a_j < a_i} L[j] + 1$$

return $max_i\, L[i]$

Run time?

Next

Edit distance problem

Edit Distance

- Given two strings s and t, the edit distance between s and t is the minimum number of editing operations needed to turn s into t

- Editing operations
  - Insertion
  - Deletion
  - Substitution

Example

```
s:  I N T E * N T I O N
    | | | | | | | | | |
t:  * E X E C U T I O N
    d s s   i s
```

- Distance: 5 (assuming unit cost for each operation)

Application: Computational Biology

- Given a sequence of bases

  AGGCTATCACCTGACCTCCAGGCCGATGCCC
  TAGCTATCACGACCGCGGTCGATTTGCCCGAC

- An alignment:

  -**AG**G**CTATCAC**CT**GACC**T**C**CA**GG**C**CGA**--**TGCCC**---
  T**AG**-**CTATCAC**--**GACC**G**C**--**GG**T**CGA**TT**TGCCC**GAC

Application: NLP

Evaluating Machine Translation and speech recognition

```
R Spokesman confirms    senior government adviser was shot
H Spokesman said    the senior            adviser was shot dead
              S      I              D                        I
```

Optimal substructure?

- Q: if this is an optimal alignment

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
```

will this be optimal for sure?

```
I N T E * N T I O
| | | | | | | | |
* E X E C U T I O
```

$s = s_1\ s_2\ \dots\ s_{m;}\ \ t = t_1\ t_2\ \dots\ t_n$

- We can create sub-problems by considering the prefixes of s and t

D(i,j) = the edit distance between $s_1 s_2 \dots s_i$ and $t_1 t_2 \dots t_j$

Q: To figure out D(i,j), what are the possible choices we can make regarding the last positions i and j?

A: Three possibilities:

Align i with j

$$\boxed{\begin{array}{l} s_1\ s_2 \dots\ \ s_{i-1}\ \ s_i \\ t_1\ t_2\ \dots\ \ t_{j-1}\ \ t_j \end{array}}$$

Align i with *

$$\boxed{\begin{array}{l} s_1\ s_2 \dots\dots\ \ s_{i-1}\ \ s_i \\ t_1\ t_2\ \dots\ \ t_{j-1}\ \ t_j\ * \end{array}}$$

Align j with *

$$\boxed{\begin{array}{l} s_1\ s_2 \dots\ \ s_{i-1}\ s_i\ * \\ t_1\ t_2\ \dots\ \ t_{j-1}\ \ \ \ t_j \end{array}}$$

- D(i,j): the minimum of the three possible choices:

Align i with j

$s_1\ s_2 \ldots\ \ s_{i-1}\ \ s_i$

$t_1\ t_2\ \ldots\ \ t_{j-1}\ \ t_j$

Choice 1:

If $s_i=t_j$:
 D(i, j) = D(i-1, j-1)

Otherwise:
 D(i, j) = D(i-1, j-1) +1

Align i with *

$s_1\ s_2 \ldots\ldots\ \ s_{i-1}\ \ s_i$

$t_1\ t_2\ \ldots\ \ t_{j-1}\ \ t_j\ \ *$

Choice 2:

D(i, j) = D(i-1, j) +1

Align j with *

$s_1\ s_2 \ldots\ \ s_{i-1}\ s_i\ \ *$

$t_1\ t_2\ \ldots\ \ t_{j-1}\ \ \ t_j$

Choice 2:

D(i, j) = D(i, j-1) +1

Recurrence relation for D(i,j)

For $i, j \geq 1$

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 & \boxed{\text{deletion}} \\ D(i, j-1) + 1 & \boxed{\text{insertion}} \\ D(i-1, j-1) + \text{diff}(s_i, t_j) & \boxed{\text{align i with j}} \end{cases}$$

$$\text{diff}(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$$

Base case? D(0,0) = 0

any others?

Edit Distance

```
For i=0 to m: D(i,0) = i
For j=1 to n: D(0,j) = j
For each  i = 1…m

    For each  j = 1…n
```

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \mathrm{diff}(s_i,\ t_j) \end{cases}$$

```
Return D(m,n)
```

The Edit Distance Table

9	N										
8	O										
7	I										
6	T										
5	N										
4	E										
3	T										
2	N										
1	I										
0	#										
j=		#	E	X	E	C	U	T	I	O	N

i= 0 1 2 3 4 5 6 7 8 9

Computing alignments

- Getting the edit distance isn't sufficient
  - We often need to **align** each character of the two strings to each other
- We do this by keeping a "backtrace"
- Every time we enter a cell, remember where we came from
- When we reach the end,
  - Trace back the path from the upper right corner to read off the alignment

Adding Backtrace to Minimum Edit Distance

- Base conditions:

  `D(i,0) = i`   `D(0,j) = j`

  Termination:

  `return D(m,n)`

- Recurrence Relation:

  `For each  i = 1…M`

  `    For each  j = 1…N`

$$
D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \boxed{\text{deletion}} \\ D(i,j-1) + 1 & \boxed{\text{insertion}} \\ D(i-1,j-1) + \mathrm{diff}(s_i,\ t_j) & \boxed{\text{align}} \end{cases}
$$

$$
ptr(i,j) = \begin{cases} \text{LEFT} & \boxed{\text{insertion}} \\ \text{DOWN} & \boxed{\text{deletion}} \\ \text{DIAG} & \boxed{\text{align}} \end{cases}
$$

Result of Backtrace

- Two strings and their **alignment**:

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
```

Performance

- Time:

  $O(nm)$

- Space:

  $O(nm)$

- Backtrace

  $O(n+m)$

Edit Distance

```
For i=0 to m: D(i,0) = i
For j=1 to n: D(0,j) = j
For each  i = 1…m

   For each  j = 1…n
```

$$D(i,j) = \min \begin{cases} D(i-1,j) + 1 \\ D(i,j-1) + 1 \\ D(i-1,j-1) + \mathrm{diff}(s_i, t_j) \end{cases}$$

```
Return D(m,n)
```

The Edit Distance Table

9	N										
8	O										
7	I										
6	T										
5	N										
4	E										
3	T										
2	N										
1	I										
0	#										
j=		#	E	X	E	C	U	T	I	O	N

i= 0 1 2 3 4 5 6 7 8 9

Computing alignments

- Getting the edit distance isn't sufficient
  - We often need to **align** each character of the two strings to each other
- We do this by keeping a "backtrace"
- Every time we enter a cell, remember where we came from
- When we reach the end,
  - Trace back the path from the upper right corner to read off the alignment

Adding Backtrace to Minimum Edit Distance

- Base conditions:

  `D(i,0) = i          D(0,j) = j`

  Termination:

  `return D(m,n)`

- Recurrence Relation*:*

```
For each  i = 1…M
     For each  j = 1…N
```

$$
D(i,j) = \min \begin{cases} D(i-1,j) + 1 & \boxed{\text{deletion}} \\ D(i,j-1) + 1 & \boxed{\text{insertion}} \\ D(i-1,j-1) + \mathrm{diff}(s_i,\ t_j) & \boxed{\text{align}} \end{cases}
$$

$$
ptr(i,j) = \begin{cases} \text{LEFT} & \boxed{\text{insertion}} \\ \text{DOWN} & \boxed{\text{deletion}} \\ \text{DIAG} & \boxed{\text{align}} \end{cases}
$$

Result of Backtrace

- Two strings and their **alignment**:

```
I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
```

Performance

- Time:

  $O(nm)$

- Space:

  $O(nm)$

- Backtrace

  $O(n+m)$