Applied Algorithms

CSCI-B505 / INFO-I500

Lecture 10.

Dynamic Programming - I

M. Oguzhan Kulekci

- Dynamic Programming
  - Fibonacci
  - Binomial Coefficients
  - Edit distance
  - Some Variants of Edit Distance
  - Longest Common Subsequence

Dynamic Programming

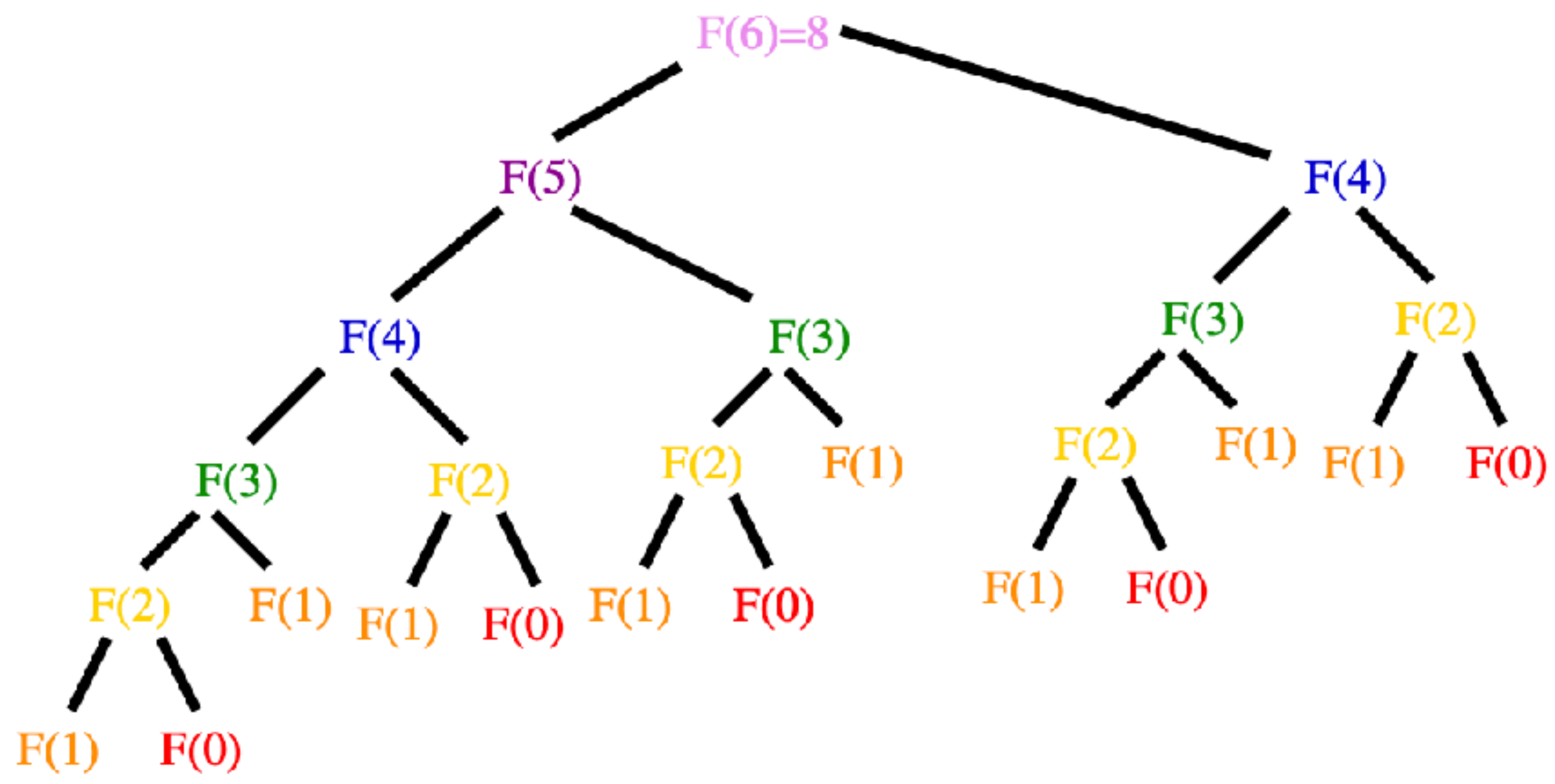
- Find the minimum or maximum of a combinatorial challenge (combinatorial opt.)
- **Exhaustive** search guarantees the optimum, but very **expensive**
- **Greedy approach** (!) is more reasonable, but **no guarantees** (*in general*)
- Dynamic programming aims to compute the optimum with a good complexity by storing the results of some prior computations for the sake of some others later.
- **DP is particularly useful when there is a <u>reductive solution but with significant overlaps</u> between the recursive steps.**

Bad Recursions

- Same computation is repeated many times in the recursion tree !

Example: Fibonacci with recursion …

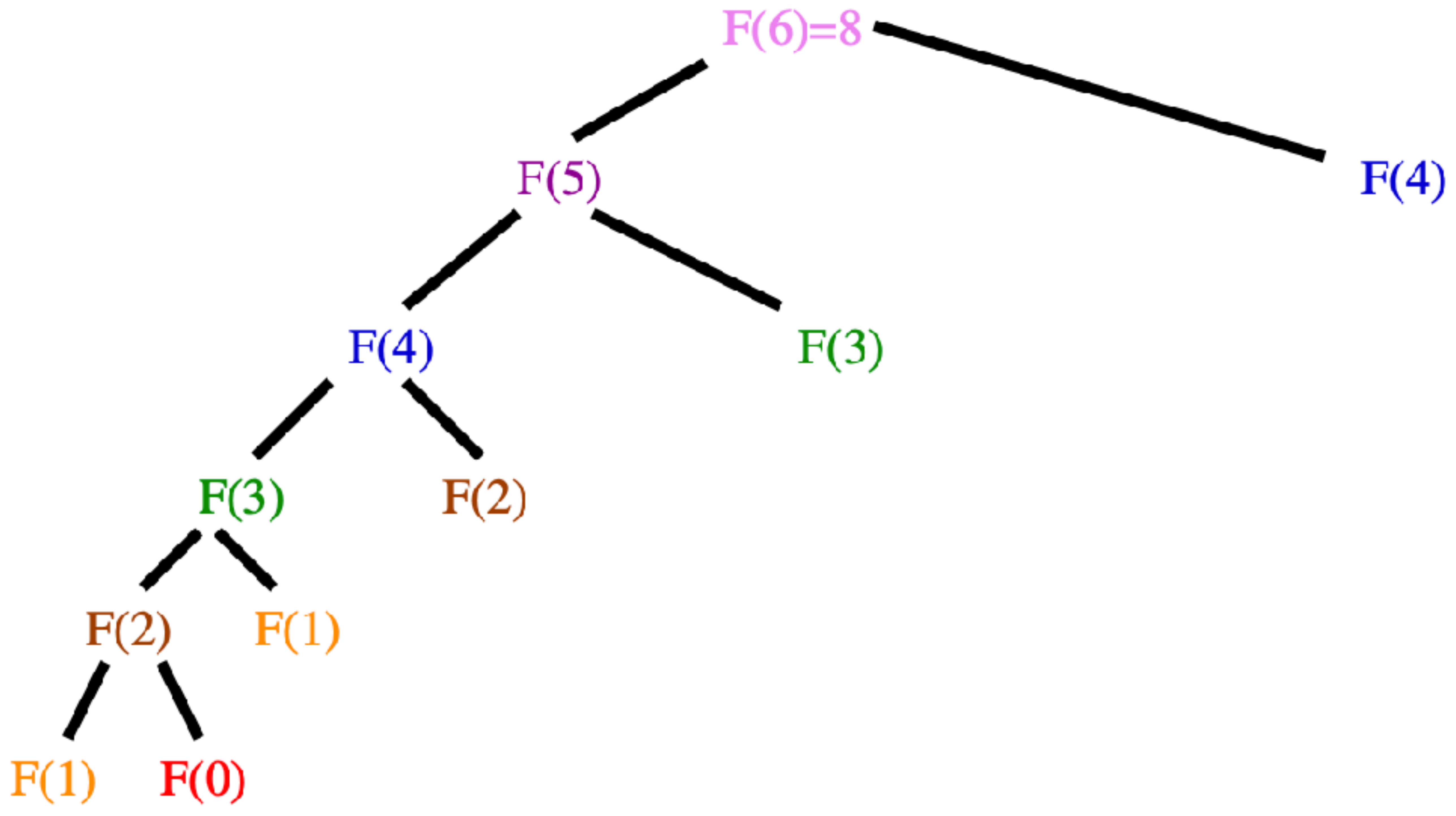
```
long fib_r(int n) {
    if (n == 0) {
        return(0);
    }

    if (n == 1) {
        return(1);
    }

    return(fib_r(n-1) + fib_r(n-2));
}
```



- Time complexity is **exponential** $\approx 1.6^n$. Why?
- $F(n) > 1.6^n$, **for large** n, and since we will sum up to this by reaching that much of leaves.

Bad Recursions

- A better recursion with caching !




```c
long fib_c(int n) {
    if (f[n] == UNKNOWN) {
        f[n] = fib_c(n-1) + fib_c(n-2);
    }

    return(f[n]);
}


long fib_c_driver(int n) {
    int i;      /* counter */

    f[0] = 0;
    f[1] = 1;

    for (i = 2; i <= n; i++) {
        f[i] = UNKNOWN;
    }

    return(fib_c(n));
}
```

- **Allocating $O(n)$ space returns the result in $O(n)$ time.**

Simple DP for Fibonacci

```
long fib_ultimate(int n)
{
        int i;                    /* counter */
        long back2=0, back1=1;    /* last two values of f[n] */
        long next;                /* placeholder for sum */

        if (n == 0) return (0);

        for (i=2; i<n; i++) {
                next = back1+back2;
                back2 = back1;
                back1 = next;
        }
        return(back1+back2);
}
```

Allocating $O(1)$ space returns the result in $O(n)$ time.

Binomial Coefficients

$$\binom{n}{k} = \frac{n!}{(n-k)! \cdot k!}$$

- It can be directly calculated. However, overflows are possible even with small n, k values.
- **Recursive calculation avoids such overflows**

```
long binomial_coefficient(int n, int k) {
    int i, j;                    /* counters */
    long bc[MAXN+1][MAXN+1];     /* binomial coefficient table */

    for (i = 0; i <= n; i++) {
        bc[i][0] = 1;
    }

    for (j = 0; j <= n; j++) {
        bc[j][j] = 1;
    }

    for (i = 2; i <= n; i++) {
        for (j = 1; j < i; j++) {
            bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
        }
    }

    return(bc[n][k]);
}
```

```
          1
        1   1
      1   2   1
    1   3   3   1
  1   4   6   4   1
1   5   10  10  5   1
```

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

n / k	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	5	1

Approximate String Matching

Given two strings s1 and s2, in how many steps can we alter s1 to become s2 ?

The alterations we are allowed to do are:

- Substitution: Change a specific symbol of s1 to match a symbol of s2,
  e.g., s1: s**h**ot s2:s**p**ot
- Insertion: Insert a new symbol into s1 to match a corresponding symbol on s2
  e.g., s1: ago s2:ago**g**
- Deletion: Delete symbol from s1
  e.g., s1: **h**our s2:our

We will assume each operation has the equal cost of 1.

Approximate String Matching by Recursion

Let edit distance between $P_1P_2 \ldots P_i$ *and* $T_1T_2 \ldots T_j$ *be* $D[i, j]$.

There are three possibilities:

1. $D[i, j] = D[i-1, j-1] + (1 \,|\, 0)$

2. $D[i, j] = D[i, j-1] + 1$ *Extra symbol on T,* $indel(T_j)$,

3. $D[i, j] = D[i-1, j] + 1$ *Extra symbol on P,* $indel(P_i)$

Approximate String Matching by Recursion

```
int string_compare_r(char *s, char *t, int i, int j) {
    int k;           /* counter */
    int opt[3];      /* cost of the three options */
    int lowest_cost; /* lowest cost */

    if (i == 0) {    /* indel is the cost of an insertion or deletion */
        return(j * indel(' '));
    }

    if (j == 0) {
        return(i * indel(' '));
    }
                     /* match is the cost of a match/substitution */

    opt[MATCH]  = string_compare_r(s,t,i-1,j-1) + match(s[i],t[j]);
    opt[INSERT] = string_compare_r(s,t,i,j-1) + indel(t[j]);
    opt[DELETE] = string_compare_r(s,t,i-1,j) + indel(s[i]);

    lowest_cost = opt[MATCH];
    for (k = INSERT; k <= DELETE; k++) {
        if (opt[k] < lowest_cost) {
            lowest_cost = opt[k];
        }
    }

    return(lowest_cost);
}
```

$$D[i,j] = D[i-1,j-1] + (1\,|\,0)$$

$$D[i,j] = D[i,j-1] + 1$$

$$D[i,j] = D[i-1,j] + 1$$

Recursive Program with exponential time ($\approx 3^n$) since at each recursion, three children are born.

$D[5][6]$

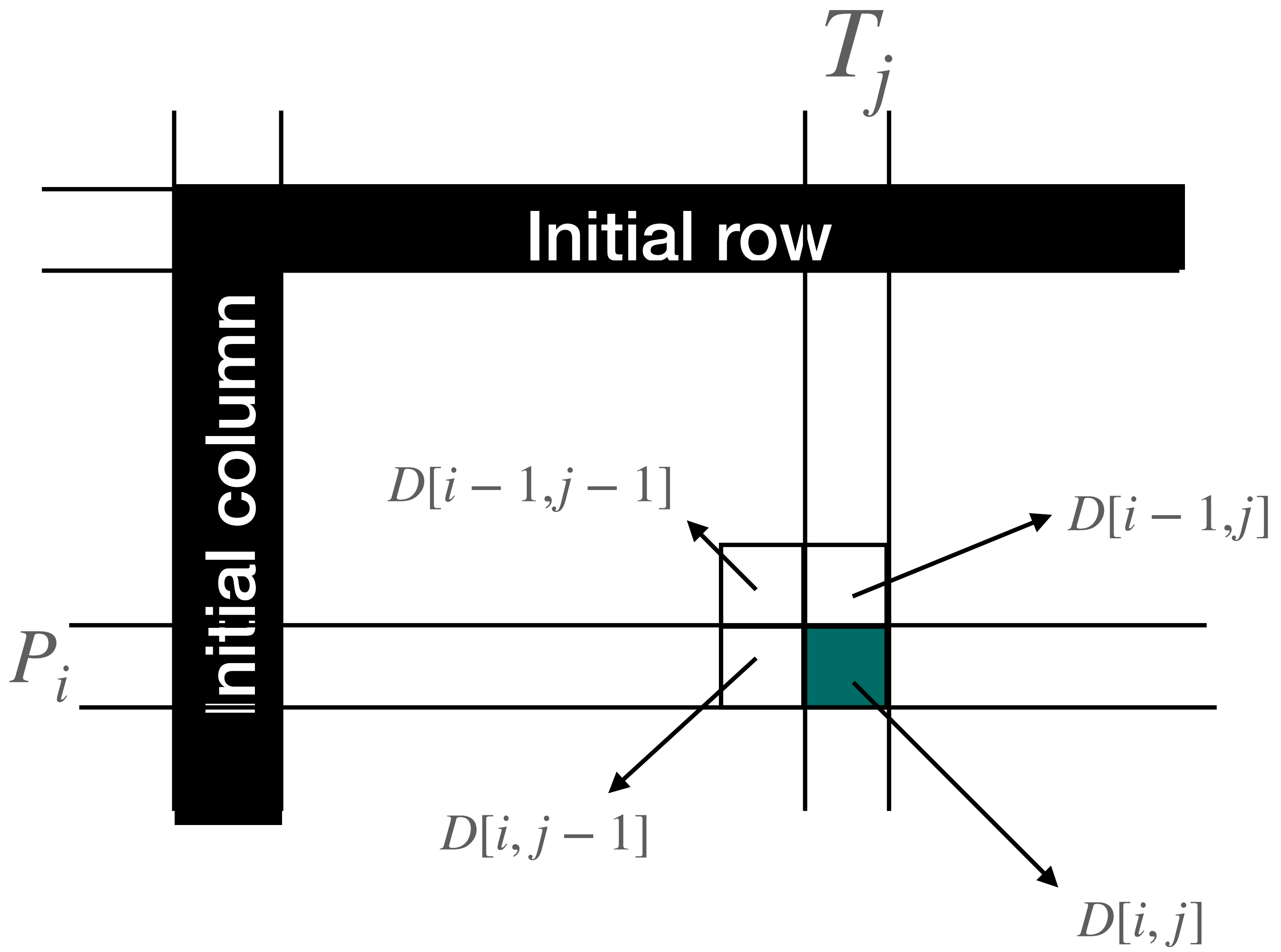
$D[4][5]$ $D[5][5]$ $D[4][6]$

$D[3][4]$ $D[4][4]$ $D[3][5]$ $D[4][4]$ $D[5][4]$ $D[4][5]$ $D[3][5]$ $D[4][5]$ $D[3][6]$

BAD RECURSION ! ?

Approximate String Matching by Dynamic Programming

P	T pos	0	y 1	o 2	u 3	- 4	s 5	h 6	o 7	u 8	l 9	d 10
:		**0**	1	2	3	4	5	6	7	8	9	10
t:	1	**1**	1	2	3	4	5	6	7	8	9	10
h:	2	2	**2**	2	3	4	5	5	6	7	8	9
o:	3	3	3	**2**	3	4	5	6	5	6	7	8
u:	4	4	4	3	**2**	3	4	5	6	5	6	7
-:	5	5	5	4	3	**2**	3	4	5	6	6	7
s:	6	6	6	5	4	3	**2**	3	4	5	6	7
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6
a:	8	8	8	7	6	5	4	3	3	**4**	5	6
l:	9	9	9	8	7	6	5	4	4	4	**4**	5
t:	10	10	10	9	8	7	6	5	5	5	5	**5**



- *Initial row and column values are fixed.*
- *We just fill the matrix with a row-major or column major traversal.*

Approximate String Matching by Dynamic Programming

```c
int string_compare(char *s, char *t, cell m[MAXLEN+1][MAXLEN+1]) {
    int i, j, k;        /* counters */
    int opt[3];         /* cost of the three options */

    for (i = 0; i <= MAXLEN; i++) {
        row_init(i, m);
        column_init(i, m);
    }

    for (i = 1; i < strlen(s); i++) {
        for (j = 1; j < strlen(t); j++) {
            opt[MATCH] = m[i-1][j-1].cost + match(s[i], t[j]);
            opt[INSERT] = m[i][j-1].cost + indel(t[j]);
            opt[DELETE] = m[i-1][j].cost + indel(s[i]);

            m[i][j].cost = opt[MATCH];
            m[i][j].parent = MATCH;
            for (k = INSERT; k <= DELETE; k++) {
                if (opt[k] < m[i][j].cost) {
                    m[i][j].cost = opt[k];
                    m[i][j].parent = k;
                }
            }
        }
    }

    goal_cell(s, t, &i, &j);
    return(m[i][j].cost);
}
```

```c
#define MATCH      0      /* enumerated type symbol for match */
#define INSERT     1      /* enumerated type symbol for insert */
#define DELETE     2      /* enumerated type symbol for delete */
```

	T		y	o	u	-	s	h	o	u	l	d
P	pos	0	1	2	3	4	5	6	7	8	9	10
:		**0**	1	2	3	4	5	6	7	8	9	10
t:	1	**1**	1	2	3	4	5	6	7	8	9	10
h:	2	2	**2**	2	3	4	5	5	6	7	8	9
o:	3	3	3	**2**	3	4	5	6	5	6	7	8
u:	4	4	4	3	**2**	3	4	5	6	5	6	7
-:	5	5	5	4	3	**2**	3	4	5	6	6	7
s:	6	6	6	5	4	3	**2**	3	4	5	6	7
h:	7	7	7	6	5	4	3	**2**	**3**	4	5	6
a:	8	8	8	7	6	5	4	3	3	**4**	5	6
l:	9	9	9	8	7	6	5	4	4	4	**4**	5
t:	10	10	10	9	8	7	6	5	5	5	5	**5**

The edit distance between P and T

DP solution of edit distance works in $O(n \cdot m)$-time and -space.

Approximate String Matching by Dynamic Programming

```
#define MATCH       0         /* enumerated type symbol for match */
#define INSERT      1         /* enumerated type symbol for insert */
#define DELETE      2         /* enumerated type symbol for delete */
```

```
void reconstruct_path(char *s, char *t, int i, int j,
                      cell m[MAXLEN+1][MAXLEN+1]) {
    if (m[i][j].parent == -1) {
        return;
    }

    if (m[i][j].parent == MATCH) {
        reconstruct_path(s, t, i-1, j-1, m);
        match_out(s, t, i, j);
        return;
    }

    if (m[i][j].parent == INSERT) {
        reconstruct_path(s, t, i, j-1, m);
        insert_out(t, j);
        return;
    }

    if (m[i][j].parent == DELETE) {
        reconstruct_path(s, t, i-1, j, m);
        delete_out(s, i);
        return;
    }
}
```

	T		y	o	u	-	s	h	o	u	l	d
P	pos	0	1	2	3	4	5	6	7	8	9	10
	0	**-1**	1	1	1	1	1	1	1	1	1	1
t:	1	**2**	0	0	0	0	0	0	0	0	0	0
h:	2	2	**0**	0	0	0	0	0	1	1	1	1
o:	3	2	0	**0**	0	0	0	0	0	1	1	1
u:	4	2	0	2	**0**	1	1	1	1	0	1	1
-:	5	2	0	2	2	**0**	1	1	1	1	0	0
s:	6	2	0	2	2	2	**0**	1	1	1	1	0
h:	7	2	0	2	2	2	2	**0**	**1**	1	1	1
a:	8	2	0	2	2	2	2	2	0	**0**	0	0
l:	9	2	0	2	2	2	2	2	0	0	**0**	1
t:	10	2	0	2	2	2	2	2	0	0	0	**0**

- The parent information can be separately maintained to reconstruct the path.
- It is also possible to gather it from the cost matrix as well.

```
t h o u - s h   a l t
  y o u - s h o u l d
D S M M M M M I S M S
```

Some Variants of the Edit Distance

- Edit distance has many different variants to solve different problems.
- All fill the DP matrix with some slight differences that make a significant effect.
    - Different cost functions
    - The position of the final result on the matrix
    - Different initialization of the matrix
    - Different traceback actions

Here are some examples

Some Variants of the Edit Distance

Substring Matching: On a long text T we aim to spot P, wherever it matches best.

Example: Searching for the best alignment of a relatively short DNA sequence on a long DNA sequence of a human around 3 gigabases long.

Edit Distance

		A	T	T	C	T	G	A	C	T	A	C	A	T
	0	1	2	3	4	5	6	7	8	9	10	11	12	13
G	1													
A	2													
T	3													
T	4													
A	5													
C	6													
A	7													

Substring Match

		A	T	T	C	T	G	A	C	T	A	C	A	T
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
G	1													
A	2													
T	3													
T	4													
A	5													
C	6													
A	7													

Find minimum cost on the last row

Some Variants of the Edit Distance

P: democrats
T: republicans
LCS(P,T): ecas

Longest Common Subsequence: Between P and T we investigate the longest match of possibly scattered sequence of symbols

```
function LCSLength(X[1..m], Y[1..n])
    C = array(0..m, 0..n)
    for i := 0..m
        C[i,0] = 0
    for j := 0..n
        C[0,j] = 0
    for i := 1..m
        for j := 1..n
            if X[i] = Y[j]
                C[i,j] := C[i-1,j-1] + 1
            else
                C[i,j] := max(C[i,j-1], C[i-1,j])
    return C[m,n]
```

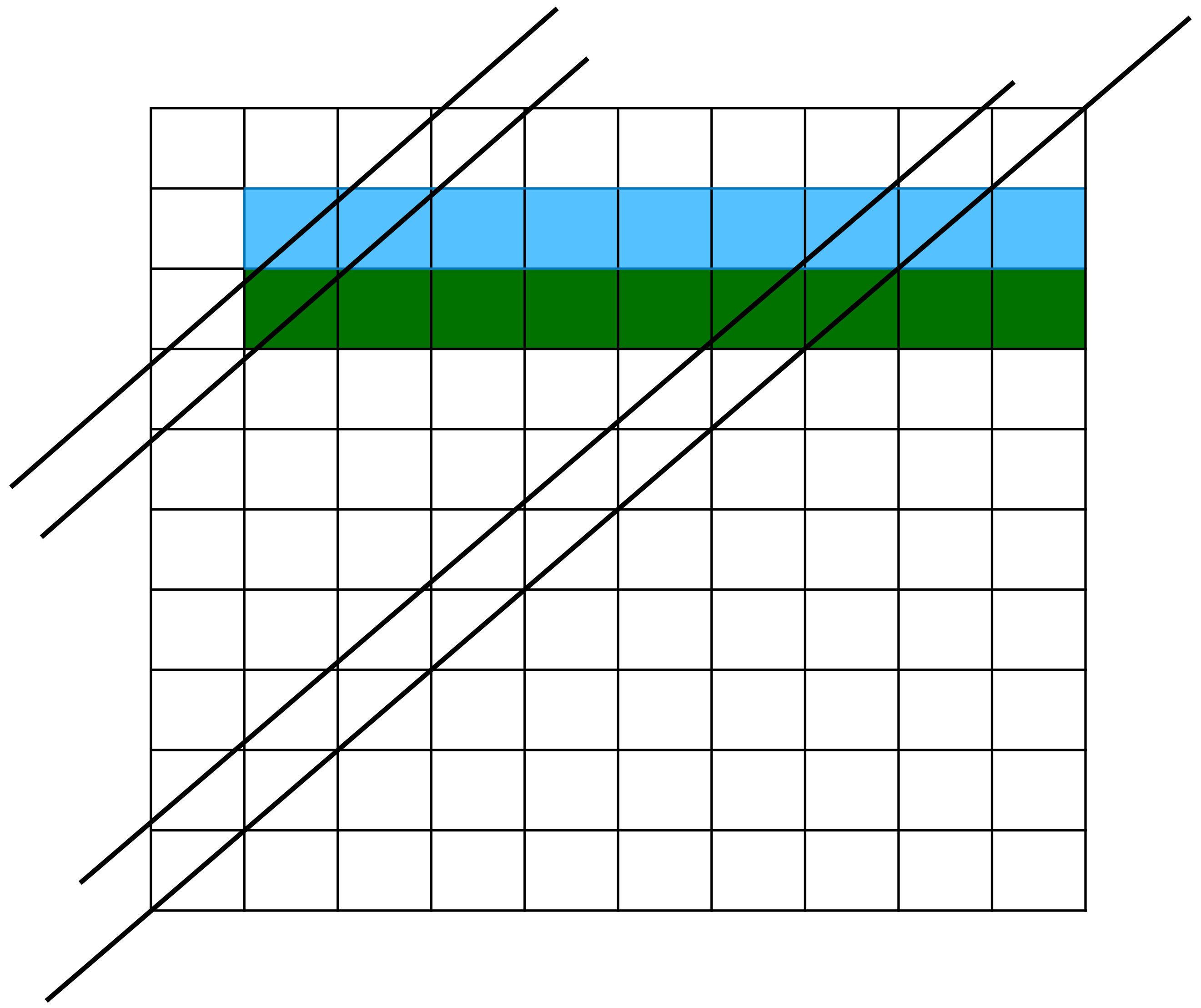
		d	e	m	o	c	r	a	t	s
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
r	**0**	0	0	0	0	0	1	1	1	1
e	**0**	0	1	**1**	1	1	1	1	1	1
p	**0**	0	1	1	1	1	1	1	1	1
u	**0**	0	1	1	1	1	1	1	1	1
b	**0**	0	1	1	1	1	1	1	1	1
l	**0**	0	1	1	1	1	1	1	1	1
i	**0**	0	1	1	1	1	1	1	1	1
c	**0**	0	1	1	1	2	2	**2**	2	2
a	**0**	0	1	1	1	2	3	3	**3**	3
n	**0**	0	1	1	1	2	3	3	3	3
s	**0**	0	1	1	1	2	3	3	3	**4**

Another way is to use normal edit distance, but preventing substitutions ! How ?

Edit Distance Computation Challenges

The matrix needs $O(n\,.\,m)$, quadratic, space. Is there a way to reduce it ?

The computation takes $O(n\,.\,m)$ time, would it be possible to speed it up via parallelization?



Reading assignment

- Read the Dynamic Programming chapters from the text books, particularly from Cormen and Skiena.