INF01056
Aula 9/10
Dynamic Programming

Prof. João Comba

Baseado no Livro Programming Challenges, Cormen et al, David Luebke notes

Dynamic Programming

- Skiena:
  - Divide and Conquer typically splits the problem in half, solves each half, then stitches the halves back together to form a solution
  - Dynamic Programming typically removes one element from the problem, solves the smaller problem, and then uses the solution to this smaller problem to add back the element in the proper way

Fibonacci Numbers

```
Fibonacci (n)
    if (n = 0) return (0)
    else if (n=1) then return (1)
    else return (fibonacci (n-1) + fibonacci (n-2))
```

Fibonacci Numbers

```
Fibonacci (n)
    if (n = 0) return (0)
    else if (n=1) then return (1)
    else return (fibonacci (n-1) + fibonacci (n-2))
```

```
Fibonacci (n)
    F[0] = 0
    F[1] = 1
    for i=1 to n F[i] = F[i-1] + F[i-2]
```

Dynamic Programiing

- Recipe (Skiena):
  - formulate the answer as a recurrence relation or recursive algorithm
  - show that the number of different values of your recurrence is bounded by a (hopefully small) polynomial
  - Specify an order of evaluation for the recurrence so you always have the partial results you need available when you need them

Dynamic Programming

- Recipe (Cormen)
  - Characterize the structure of an optimal solution (min or max, for example)
  - Recursively define the value of an optimal solution
  - Compute the value of an optimal solution in a bottom-up fashion
  - Construct an optimal solution from computed information

Knapsack problem

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W.

So we must consider weights of items as well as their value.

Item	Weight	Value
A	3	4
B	4	5
C	7	10
D	8	11
E	9	13

Knapsack tamanho 17 ?

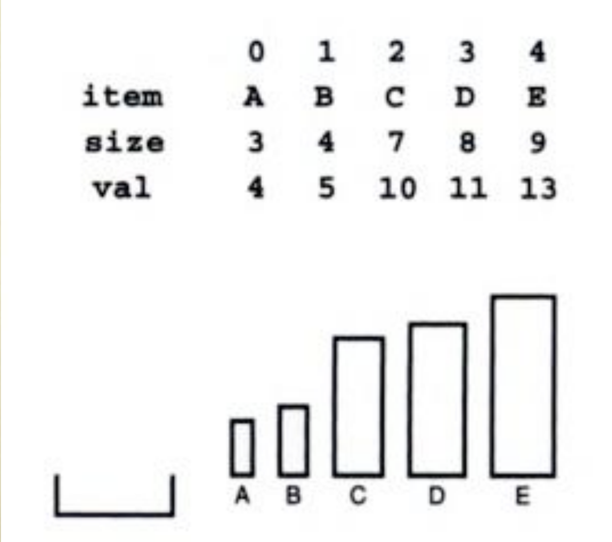
Knapsack problem

There are two versions of the problem:
(1) "0-1 knapsack problem" and
(2) "Fractional knapsack problem"

(1) Items are indivisible; you either take an item or not. Solved with *dynamic programming*
(2) Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.

Knapsack Problem

```
for (j = 1; j <= N; j++) {
  for (i=1; i<=M; i++) {
    if (i >= size[j])
      if (cost[i] < cost[i-size[j]]+val[j]) {
        cost[i] = cost[i-size[j]] + val[j];
        best[i] = j;
      }
  }
}
```



	0	1	2	3	4
item	A	B	C	D	E
size	3	4	7	8	9
val	4	5	10	11	13

Longest Common Subsequence (LCS)

- Application: comparison of two DNA strings
  - Ex: X= {A B C B D A B }, Y= {B D C A B A}
- Longest Common Subsequence:
  - X = A B C B D A B
  - Y = B D C A B A
- Brute force algorithm would compare each subsequence of X with the symbols in Y

LCS Algorithm

- if $|X| = m$, $|Y| = n$, then there are 2^m subsequences of X; we must compare each with Y (n comparisons)

- So the running time of the brute-force algorithm is O($n\ 2^m$)

- Notice that the LCS problem has optimal substructure: solutions of subproblems are parts of the final solution.

- Subproblems: "find LCS of pairs of prefixes of X and Y"

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with *i = j = 0* (empty substrings of x and y)

- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. *c[0,0] = 0*)

- LCS of empty string and any other string is empty, so for every i and j: *c[0, j] = c[i,0] = 0*

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** *x[i]=y[j]*: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{i-1} , plus 1

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i]=y[j], \\ \max(c[i,j-1],c[i-1,j]) & \text{otherwise} \end{cases}$$

- **Second case:** *x[i] != y[j]*

- As symbols don’t match, our solution is not improved, and the length of LCS(X_i , Y_j) is the same as before (i.e. maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just use length of LCS(X_{i-1}, Y_{j-1}) ?

LCS Length Algorithm

```
LCS-Length(X, Y)
m = length(X)  // get the # of symbols in X
n  = length(Y) // get the # of symbols in Y
for i = 1 to m    c[i,0] = 0   // special case: $Y_0$
for j = 1 to n    c[0,j] = 0   // special case: $X_0$
for i = 1 to m                 // for all $X_i$
    for j = 1 to n             // for all $Y_j$
        if ( $X_i$ == $Y_j$ )
            c[i,j] = c[i-1,j-1] + 1
        else c[i,j] = max( c[i-1,j], c[i,j-1] )
return c
```

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB

X = A B C B

Y = B D C A B

LCS Example (0)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		yj	B	D	C	A	B
0	xi						
1	A						
2	B						
3	C						
4	B						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[4,5]

LCS Example (1)

ABCB
BDCAB

	j	0	1	2	3	4	5
i	Yj		B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0					
2	B	0					
3	C	0					
4	B	0					

```
for i = 1 to m  c[i,0] = 0
for j = 1 to n  c[0,j] = 0
```

LCS Example (2)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	yj		B	D	C	A	B
0	xi	0	0	0	0	0	0
1	A	0	0				
2	B	0					
3	C	0					
4	B	0					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (3)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0		
2	B	0					
3	C	0					
4	B	0					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (4)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	
2	B	0					
3	C	0					
4	B	0					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (5)

ABCB

BDCAB

	j	0	1	2	3	4	5
i		yj	B	D	C	A	B
0	xi	0	0	0	0	0	0
1	A	0	0	0	0	1	→ 1
2	B	0					
3	C	0					
4	B	0					

if (x_i == y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (6)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1				
3	C	0					
4	B	0					

if (X_i == Y_j)
c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (7)

ABCB
BDCAB

i	j	0	1	2	3	4	5
	Yj		B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (8)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

if ($X_i == Y_j$)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (10)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	↓ 1 →	↓ 1			
4	B	0					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (11)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2		
4	B	0					

if ($X_i == Y_j$)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (12)

ABCB
BDCAB

	j	0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0					

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (13)

ABCB
BDCAB

j		0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1				

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (14)

ABCB
BDCAB

	j	0	1	2	3	4	5
i	Yj		B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Example (15)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

if (X_i == Y_j)

c[i,j] = c[i-1,j-1] + 1

else c[i,j] = max(c[i-1,j], c[i,j-1])

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

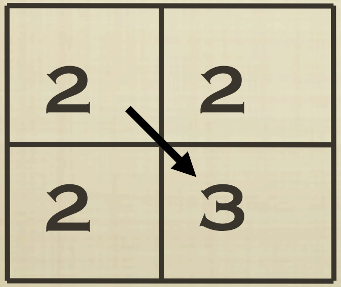
How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each *c[i,j]* depends on *c[i-1,j]* and *c[i,j-1]*

or *c[i-1, j-1]*

For each c[i,j] we can say how it was acquired:



2	2
2	3

For example, here

c[i,j] = c[i-1,j-1] +1 = 2+1=3

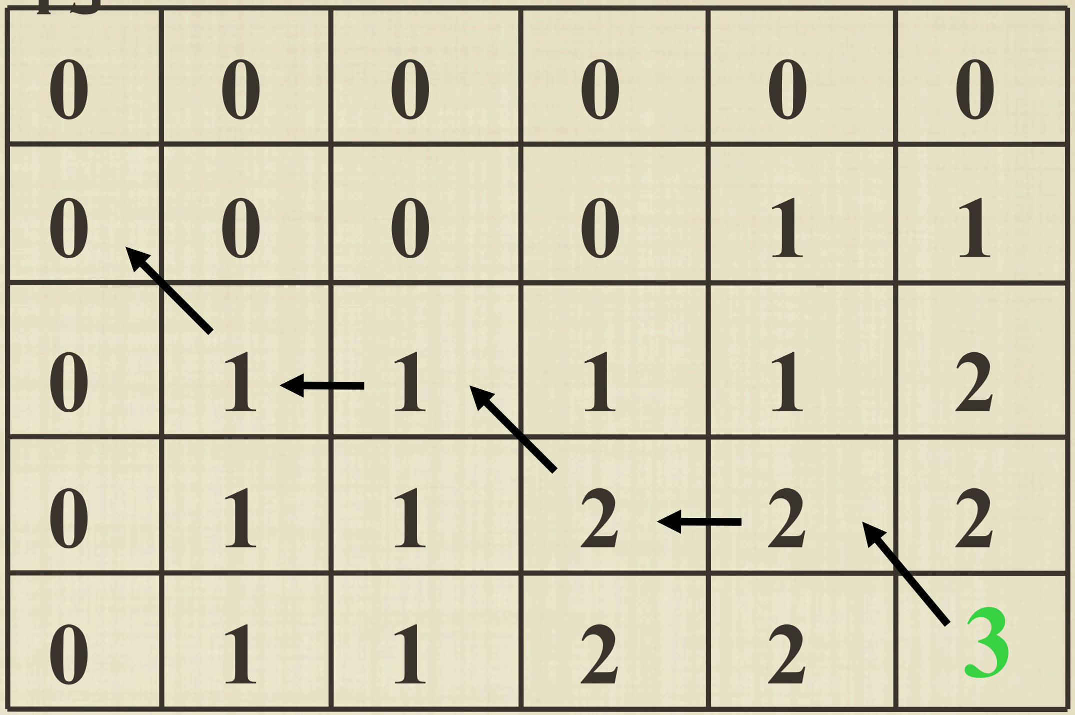
How to find actual LCS - continued

- Remember that

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

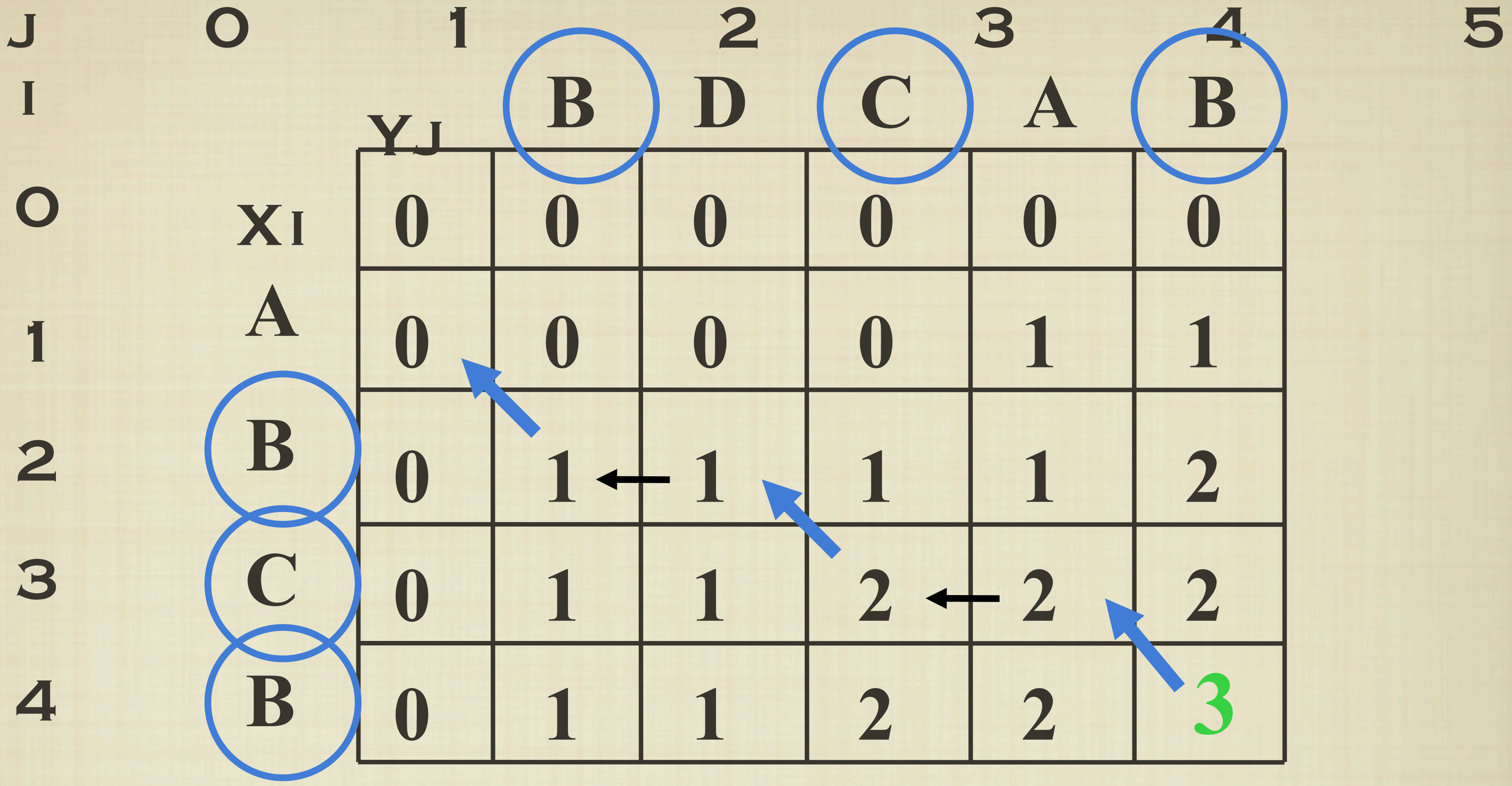
- So we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]* (because *x[i]* is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

Finding LCS



j		0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

Finding LCS (2)



j		0	1	2	3	4	5
i		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	1	1	2
3	C	0	1	1	2	← 2	2
4	B	0	1	1	2	2	3

LCS (reversed order): B C B

LCS (straight order): B C B

(this string turned out to be a palindrome)

Edit Distance

```
#define MAXLEN          101     /* longest possible string */

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

typedef struct {
        int cost;               /* cost of reaching this cell */
        int parent;             /* parent cell */
} cell;
```

Edit Distance

```
cell m[MAXLEN+1][MAXLEN+1]; /* dynamic programming table */


/**********************************************************************/

int string_compare(char *s, char *t)
                                    {
    int i,j,k;      /* counters */
    int opt[3];     /* cost of the three options */

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

    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 );
}
```

Edit Distance

```
reconstruct_path(char *s, char *t, int i, int j)
                        {
/*printf("trace (%d,%d)\n",i,j);*/

    if (m[i][j].parent == -1) return;

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