University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

5 YEARS Pitt SCI

Fall 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming Deadlines
  - Lab 9 and Hw 9: reopened till this Friday @ 11:59 pm
  - Lab 10: tonight @ 11:59 pm
  - Homework 10: this Friday 12/2 @ 11:59 pm
  - Lab 11: Monday 12/5 @ 11:59 pm
  - Homework 11: Friday 12/9 @ 11:59 pm
  - Assignment 3: ~~Monday 11/28~~ Friday 12/9 @ 11:59 pm
  - Assignment 4: Friday 12/9 @ 11:59 pm

Previous Lecture

- Dynamic Programming Examples
  - Unbounded Knapsack
  - 0/1 Knapsack

This Lecture

- Dynamic Programming Examples
  - Subset Sum
  - Edit Distance
  - Longest Common Subsequence

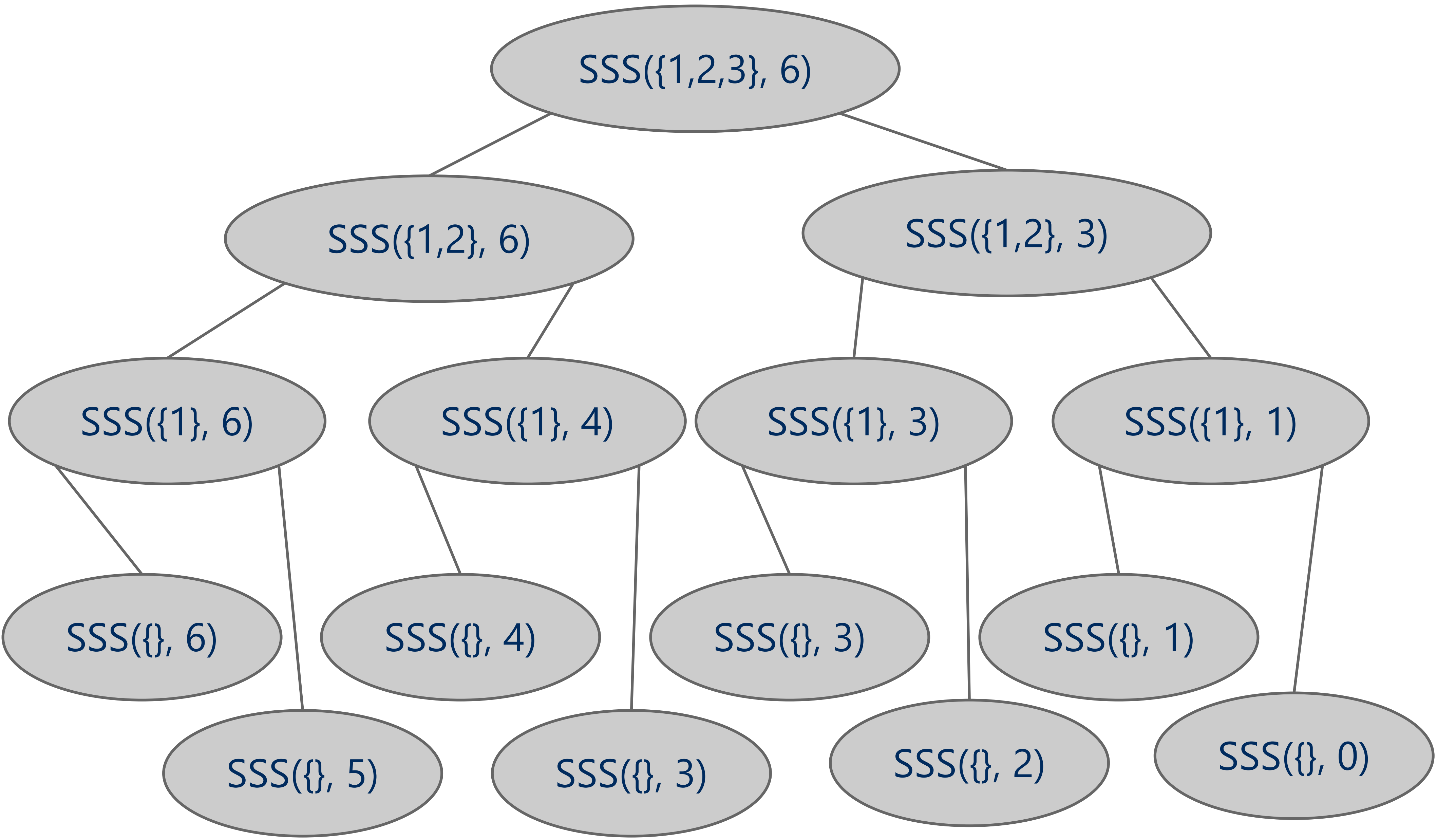
Muddiest Points

- **Q:** **Is the unbounded knapsack solution different from backtracking with pruning state repeats, or is there some fundamental difference b/w the 2?**
- backtracking with pruning state repeats is pretty much the same as memoization
- The fundamental difference is that backtracking is recursive but dynamic programming is iterative

Subset sum

- Given a set of non-negative integers S and a value k, is there a subset of S that sums to exactly k?

Subset sum calls



Subset sum recursive solution

```
boolean SSS(int set[], int sum, int n) {
    if (sum == 0)
        return true;
    if (sum != 0 && n == 0)
        return false;
    //try adding item n-1
    if (set[n-1] > sum)
        return SSS(set, sum, n-1);
    return SSS(set, sum, n-1)
        || SSS(set, sum-set[n-1], n-1);
}
```

- What would a dynamic programming table look like?

The Subset Sum dynamic programming solution

```
set  = [1, 2, 3]
```

`subset[n+1][sum+1]`

i\j	0	1	2	3	4	5	6
0							
1							
2							
3							

subset[i][j] is true iff a subset of the first ***i*** items sums up to ***j***

The Subset Sum dynamic programming solution

```
set  = [1, 2, 3]
```

```
subset[n+1][sum+1]
```

i\j	0	1	2	3	4	5	6
0	true	false	false	false	false	false	false
1	true						
2	true						
3	true						

subset[i][j] is true iff a subset of the first ***i*** items sums up to ***j***

The Subset Sum dynamic programming solution

```
set  = [1, 2, 3]
```

```
subset[n+1][sum+1]
```

i\j	0	1	2	3	4	5	6
0	true	false	false	false	false	false	false
1	true						
2	true		go left by set[i-1]		OR		
3	true						

Subset sum bottom-up dynamic programming

```
boolean SSS(int set[], int sum, int n) {
    boolean[][] subset = new boolean[n+1][sum+1];
    for (int i = 0; i <= n; i++) subset[i][0] = true;
    for (int i = 1; i <= sum; i++) subset[0][i] = false;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= sum; j++) {
            subset[i][j] = subset[i-1][j];
            //try adding item i-1
            if (j >= set[i-1])
                subset[i][j] ||= subset[i-1][j-set[i-1]];
        }
    }
    return subset[n][sum];
}
```

Edit Distance

- Given two strings
  - a string S of length n
  - a string T of length m
- We want to find the minimum number of character changes to convert one string to the other
  - called Levenshtein Distance (LD)
- Consider changes to be one of the following:
  - Change a character in a string to a different char
  - Delete a character from one string
  - Insert a character into one string

Edit Distance

- For example:
  LD("WEASEL", "SEASHELL") = 3
  - Why? Consider "WEASEL":
    - Change the W in position 1 to an S
    - Add an H in position 5
    - Add an L in position 8
  - Result is SEASHELL
    - We could also do the changes from the point of view of SEASHELL if we prefer
- How can we determine this?
  - We can define it in a recursive way initially
  - Then we will use dynamic programming to improve the run-time

Edit Distance

- We want to calculate D[n, m] where n is the length of S and m is the length of T
  - From this point of view we want to determine the distance from S → T
    - If we reverse the arguments, we get the (same) distance from T to S (but the edits may be different)

If n = 0 // BASE CASES

return m (m appends will create T from S)

else if m = 0

return n (n deletes will create T from S)

else

Consider character n of S and character m of T

- Now we have some possibilities

Edit Distance

- If characters **match**
  - return D[n-1, m-1]
    - Result is the same as for the strings with the last character removed (since it matches)
  - Recursively solve the same problem with both strings one character smaller
- If characters **do not match** -- more poss. here
  - We could have a **mismatch** at that char**:**
    - return D[n-1, m-1] + 1
    - Example:
      - `S = -------------X`
      - `T = -------------Y`
    - Change X to Y, then recursively solve the same problem but with both strings one character smaller

Edit Distance

- S could have an **extra** character
  - return D[n-1, m] + 1
  - Example:
    - `S = -----------XY`
    - `T = ------------X`
  - Delete Y, then recursively solve the same problem, with S one char smaller but with T the same size
- S could be **missing** a character there
  - return D[n, m-1] + 1
  - Example:
    - `S = ------------Y`
    - `T = -----------YX`
  - Add X onto S, then recursively solve the same problem with S the original size and T one char smaller

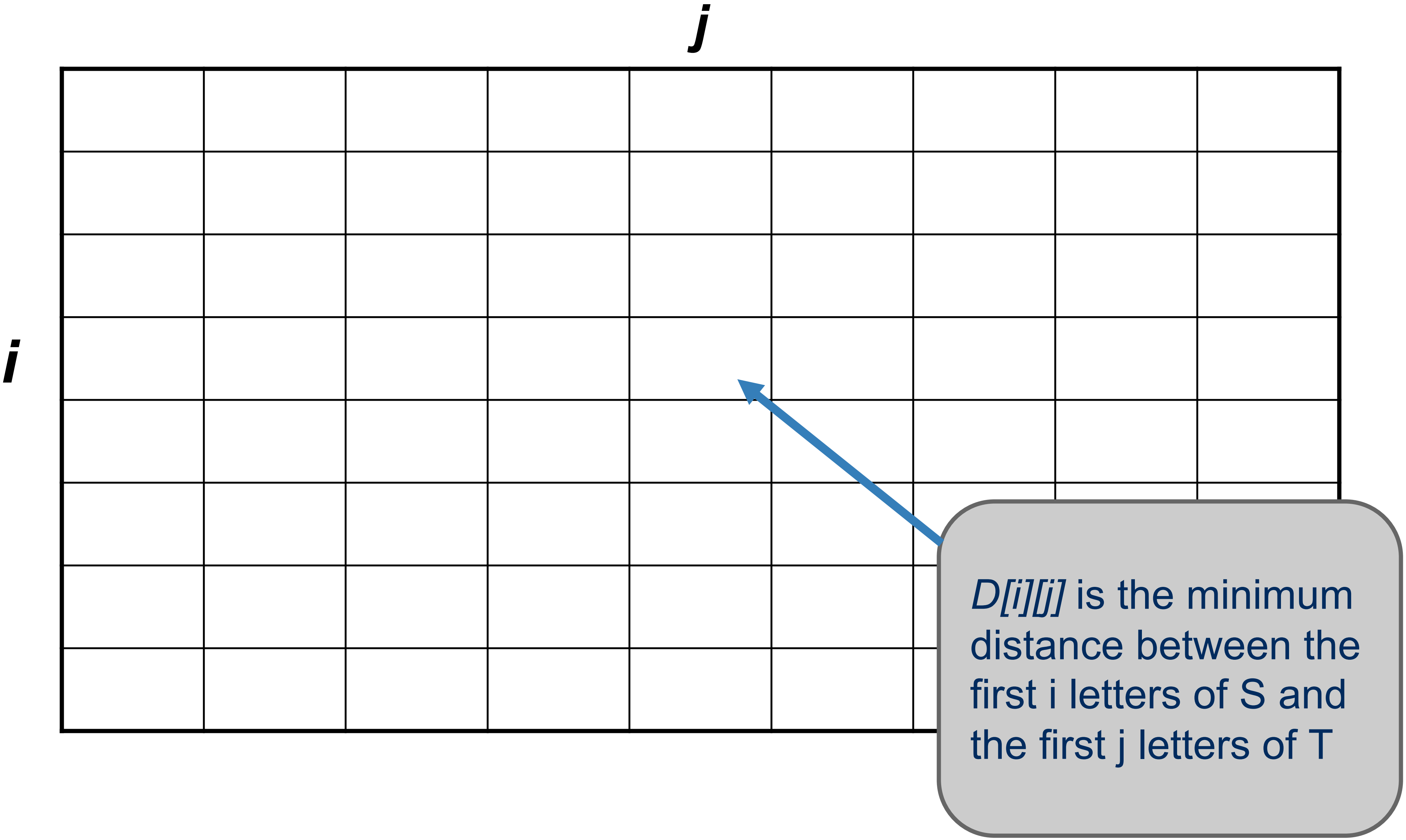
Edit Distance

- Unfortunately, we don't know which of these gives the minimum distance until we try them all!
- So to solve this problem we must try them all and choose the one that gives the minimum result
  - This yields 3 recursive calls for each original call (in which a mismatch occurs)
  - and thus can give a worst-case run-time of Theta(3^n)
- How can we do this more efficiently?
  - Let's build a table of all possible values for n and m using a two-dimensional array
  - Basically we are calculating the same D[][] values but from the bottom up rather than from the top down

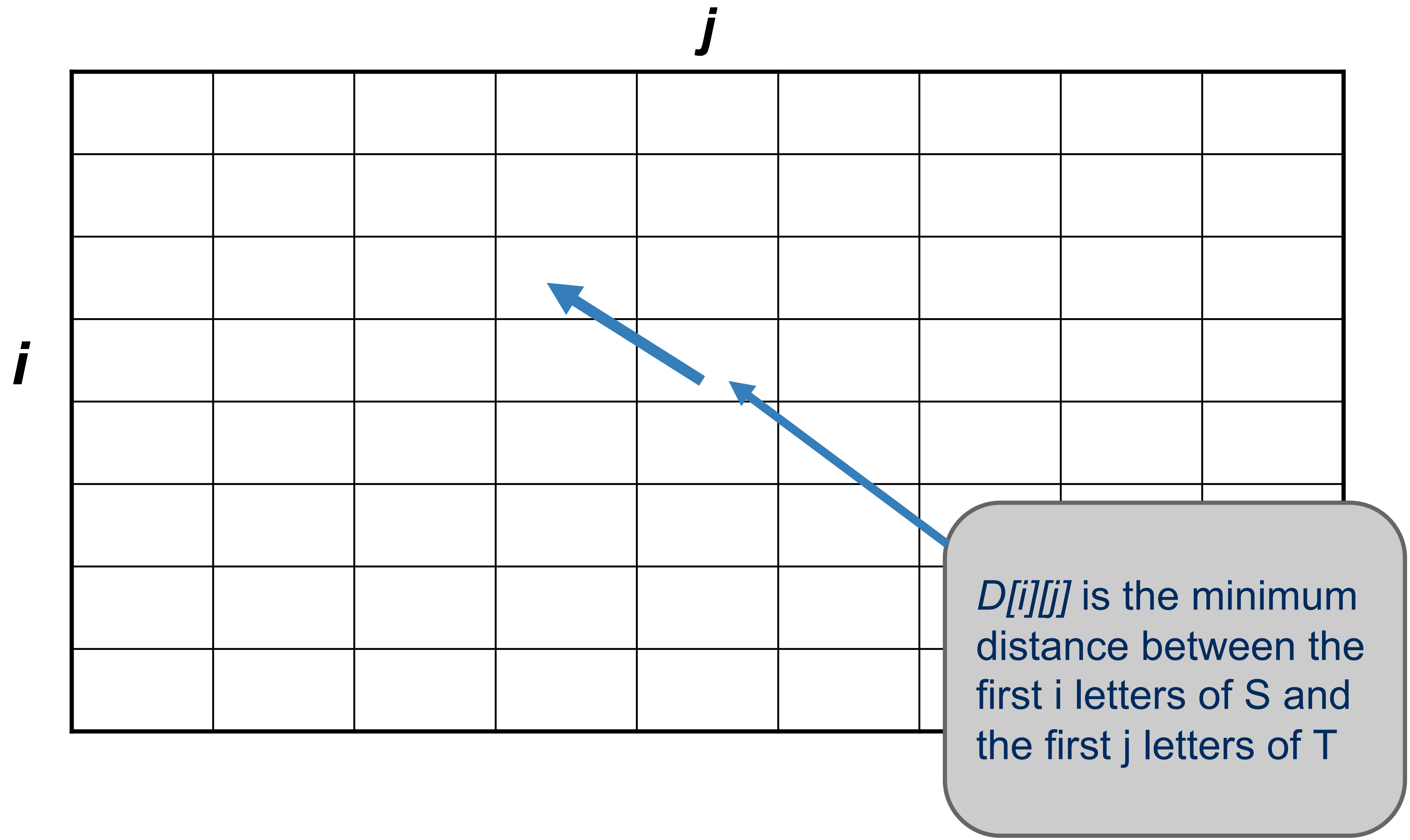
Edit Distance

- For each new cell D[i, j] = D[i-1, j-1] if we have a match
- For each new cell D[i, j] when we have a mismatch we are taking the minimum of the cells
  - D[i-1, j] + 1
    - Delete a char from S
  - D[i, j-1] + 1
    - Append a char to S
  - D[i-1, j-1] + 1
    - Change char at this point in S if necessary

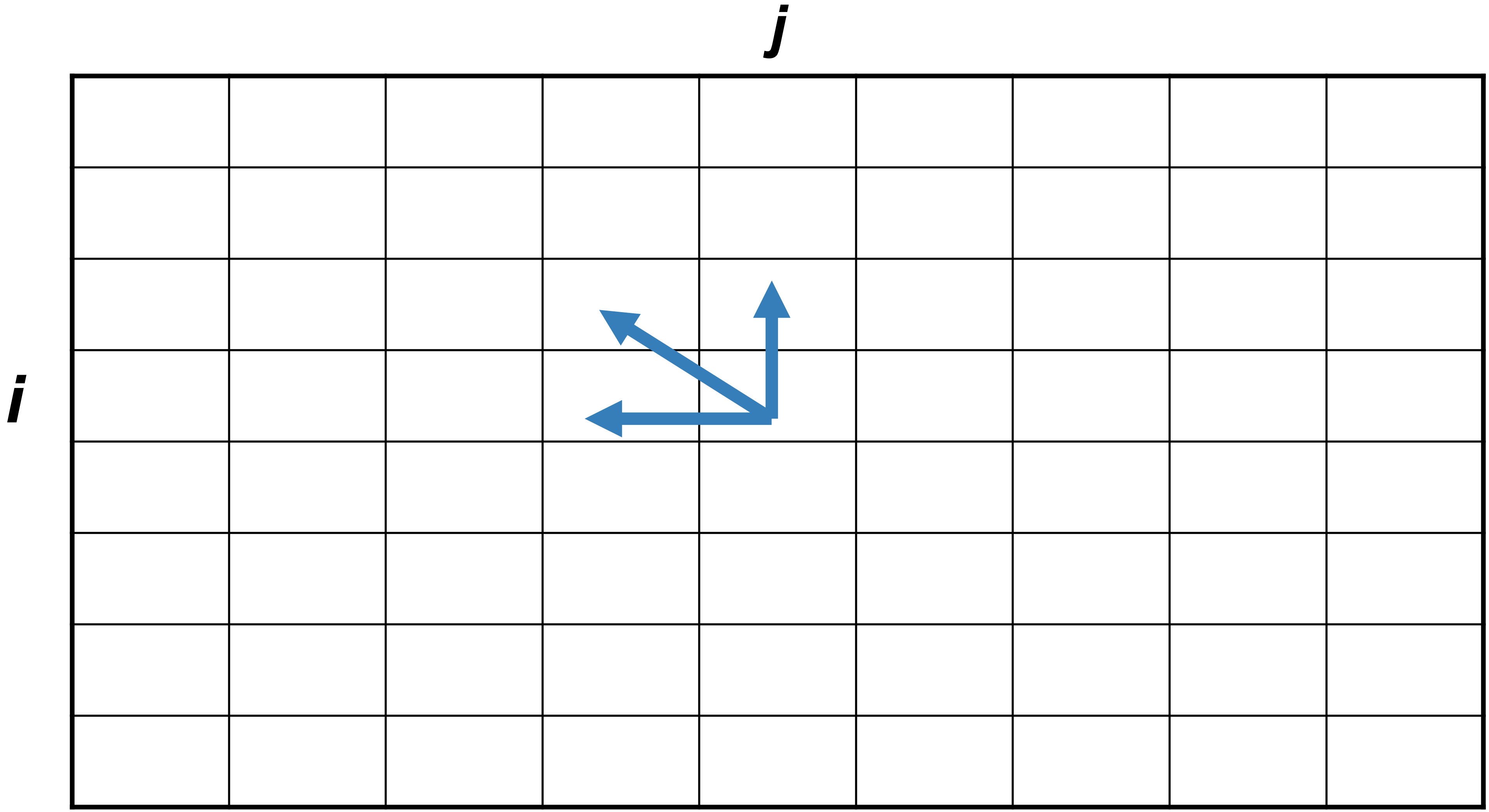
Edit Distance in case of matching letters i and j



Edit Distance in case of matching letters i and j



Edit Distance in case of mismatching letters i and j



Edit Distance

- At the end the value in the **bottom right corner** is our edit distance
- Example:
  - We are starting with PROTEIN
  - We want to generate ROTTEN
    - Note the initialization of the first row and column
    - Let's fill in the remaining squares

Edit Distance

		P	R	O	T	E	I	N
R								
O								
T								
T								
E								
N								

Edit Distance

		P	R	O	T	E	I	N
	0	1	2	3	4	5	6	7
R	1							
O	2							
T	3							
T	4							
E	5							
N	6							

Edit Distance

		P	R	O	T	E	I	N
	0	1	2	3	4	5	6	7
R	1	1	1	2	3	4	5	6
O	2	2	2	1	2	3	4	5
T	3	3	3	2	1	2	3	4
T	4	4	4	3	2	2	3	4
E	5	5	5	4	3	2	3	4
N	6	6	6	5	4	3	3	***3***

Edit Distance

- Why is this cool?
  - Run-time is Theta(MN)
    - As opposed to the 3^n of the recursive version
  - Unlike the pseudo-polynomial subset sum and knapsack solutions, this solution does not have any anomalous worst-case scenarios
    - There is a price, which is the space required for the matrix
    - Optimized versions can reduce this from Theta(MN) space to Theta(M+N) space

Longest Common Subsequence

- Given two sequences, return the longest common subsequence
  - A **Q** S R **J** K **V** B **I**

    Q B W F **J** **V** **I** T U
- We'll consider a relaxation of the problem and only look for the *length* of the longest common subsequence

LCS dynamic programming example

`x = A Q S R J B I`

`y = Q B I J T U T`

i\j	0	Q	B	I	J	T	U	T
0								
A								
Q								
S								
R								
J								
B								
I								

LCS dynamic programming solution

```
int LCSLength(String x, String y) {
   int[][] m = new int[x.length + 1][y.length + 1];
   for (int i=0; i <= x.length; i++) {
           for (int j=0; j <= y.length; j++) {
                 if (i == 0 || j == 0) m[i][j] = 0;
                 if (x.charAt(i) == y.charAt(j))
                       m[i][j] = m[i-1][j-1] + 1;
                 else
                       m[i][j] = max(m[i][j-1], m[i-1][j]);
           }
      }
   return m[x.length][y.length];
}
```

Change making problem

Consider a currency with n different denominations of coins d_1, d_2, ..., d_n. What is the minimum number of coins needed to make up a given value k?

So, how can we solve the change making problem optimally?

We will see a dynamic programming algorithm in the recitations