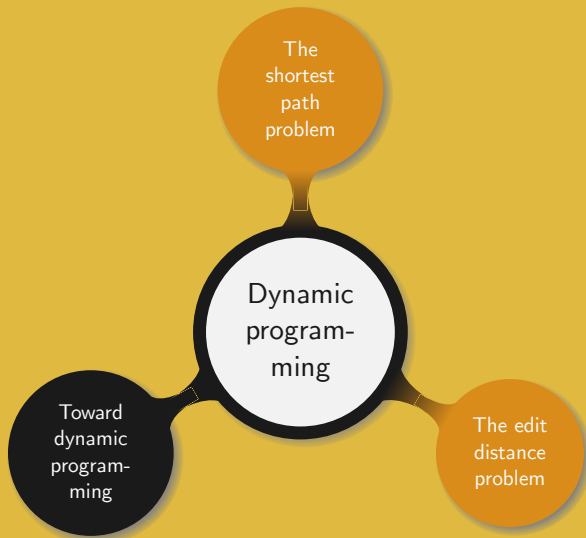


```
1001011111010100010101001111000111000100010111000000100110100001111101
1010010110011100111011101010001110000010001101110101011011111101010000
11011100111000110100110101100110110101010100111101000010101010101000
000110011110001000000110101011011110111100110000001010101011110100001
1100101110000100001001010110000011100010101000100110110010001101100100
0101011010111101010010010111010011110010100111100101110011110100
0110001111101000011010011111010111001100110111101001101
1110100100000100111101111010010001100110011001100110011001100110011001
1001111000111101000101011101010101010101010101010101010101010101010101
01111101111111010001001001001001001001001001001001001001001001001001001
1110110110011110000100101010101010101010101010101010101010101010101010
1101101000100110010100100111100001010011100001000101110010011011010011
0000110101010001100110000111000100101110011110000101001101100100000110
110111101001010101110101111000010100010011101100000010110111100000011
011110100010010101101100111001110101101110010110101011001100110101100
1011011011011111010000000011101110001111010111011001110011010011100000
```

Introduction to Algorithms

3. Dynamic programming

Manuel – Fall 2021



Fibonacci posed the following problem in his book *Liber Abbaci* (Book of Calculations) in 1202:

A certain man puts a pair of rabbits in a place surrounded on all sides by a wall. How many pairs of rabbits can be produced from that pair in a year if it is supposed that every month each pair begets a new pair, which from the second month on becomes productive?

The solution leads to the sequence of Fibonacci numbers:

- At the beginning of month 1, there is $F_0 = 1$ pair, which is not productive.
- At the beginning of month 2, there is still $F_1 = 1$ pair, which is now productive.
- At the beginning of month 3, there are now $F_2 = 2$ pairs, of which one is productive.

- At the beginning of month 4, there are now $F_3 = 3$ pairs and the pair born in month 3 becomes productive.
- As the beginning of month 5, the number of pairs is equal to those of month 4, plus all those that were productive in month 4. These are all pairs that existed in month 2, since all of those will be productive in month 3. Hence, $F_4 = 3 + 2 = 5$.
- In general, at the beginning of every month the number of pairs of rabbits is equal to the number of pairs of the previous month, plus the number of pairs of two months ago, which have since become productive.

Hence,

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2, \quad F_0 = 1, \quad F_1 = 1.$$

Algorithm.

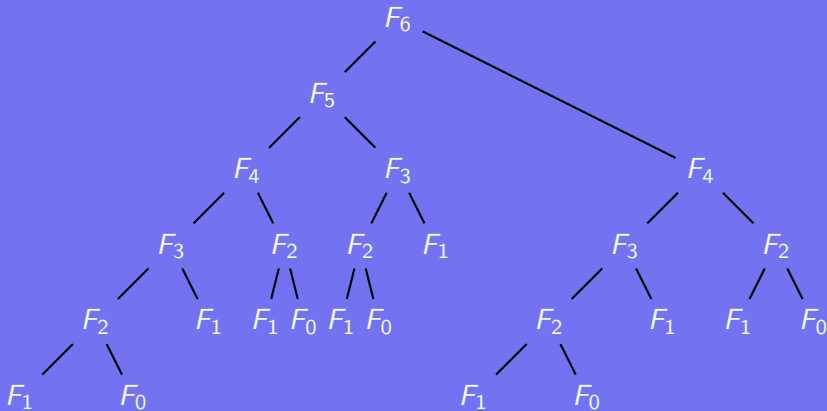
Input : An integer n

Output: F_n

```
1 Function Fib_vn( $n$ ):  
2   if  $n = 0$  then return 0;  
3   if  $n = 1$  then return 1;  
4   return Fib_vn( $n - 1$ ) + Fib_vn( $n - 2$ )  
5 end
```

Since $F_{n+1}/F_n \approx \Phi = \frac{1+\sqrt{5}}{2} > 1.6$, it means that $F_n > 1.6^n$.

Noting that each recursive call has to reach 0 or 1 to be completed it is clear that the complexity of this algorithm is exponential.



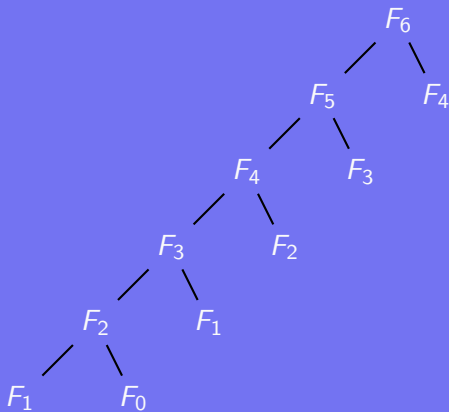
Algorithm.

Input : An integer n

Output: F_n

```
1 Function Fib_n( $n$ ):  
2   | if  $F_n = -1$  then  $F_n \leftarrow \text{Fib\_n}(n-1) + \text{Fib\_n}(n-2)$ ;  
3   | return  $F_n$   
4 end  
5 Function Fib_nm( $n$ ):  
6   |  $F_0 \leftarrow 0$ ;  
7   |  $F_1 \leftarrow 1$ ;  
8   | for  $i \leftarrow 2$  to  $n$  do  $F_i \leftarrow -1$ ;  
9   | return Fib_n( $n$ )  
10 end
```

Both the time and space complexities are linear in n



Algorithm.

Input : An integer n

Output: F_n

```
1 Function Fib( $n$ ):  
2    $F_{old2} \leftarrow 0$ ;  $F_{old1} \leftarrow 1$ ;  
3   if  $n = 0$  then return 0;  
4   for  $i \leftarrow 2$  to  $n - 1$  do  
5      $F \leftarrow F_{old1} + F_{old2}$ ;  $F_{old2} \leftarrow F_{old1}$ ;  $F_{old1} \leftarrow F$ ;  
6   end for  
7   return  $F_{old1} + F_{old2}$   
8 end
```

The time is linear in n and the storage constant

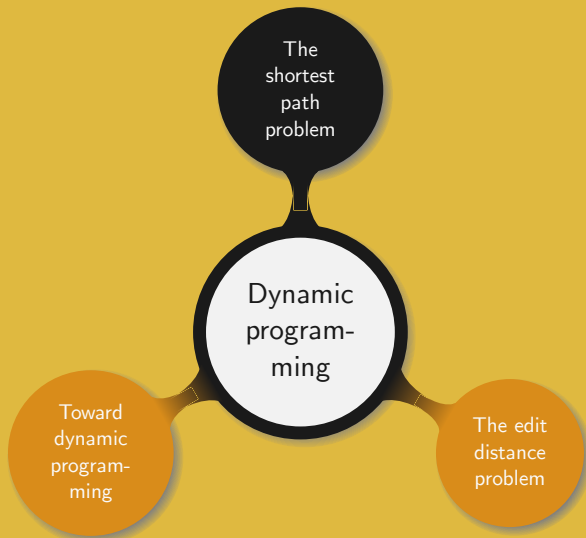
Simple idea behind dynamic programming:

- Break a complex problem into simpler subproblems
- Store the result of the overlapping subproblems
- Do not recompute the same information again and again
- Do not waste memory because of recursion

Simple idea behind dynamic programming:

- Break a complex problem into simpler subproblems
- Store the result of the overlapping subproblems
- Do not recompute the same information again and again
- Do not waste memory because of recursion

Dynamic programming saves both time and space



Problem (Shortest paths in weighted graphs)

Given a connected, simple, weighted graph, and two vertices s and t , find the shortest path that joins s to t .

Two main cases for the graph:

- It only has edges with positive labels
- It has edges with positive and negative labels

We now recall Dijkstra's algorithm to solve the first case and then introduce the Bellman-Ford algorithm which takes advantage of dynamic programming in order to treat the second one.

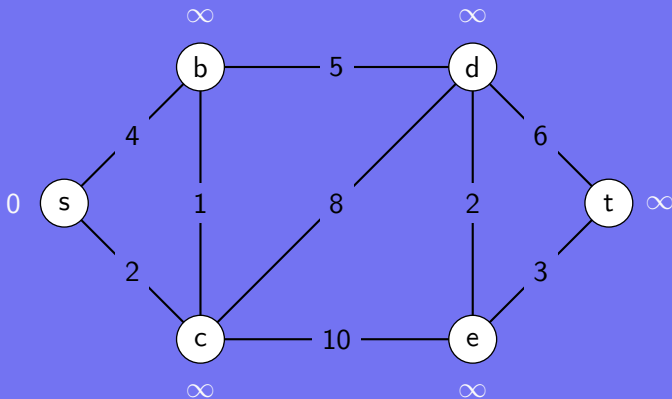
Algorithm. (*Dijkstra*)

Input : A graph $G = \langle V, E \rangle$ with positive edges, two vertices s and t

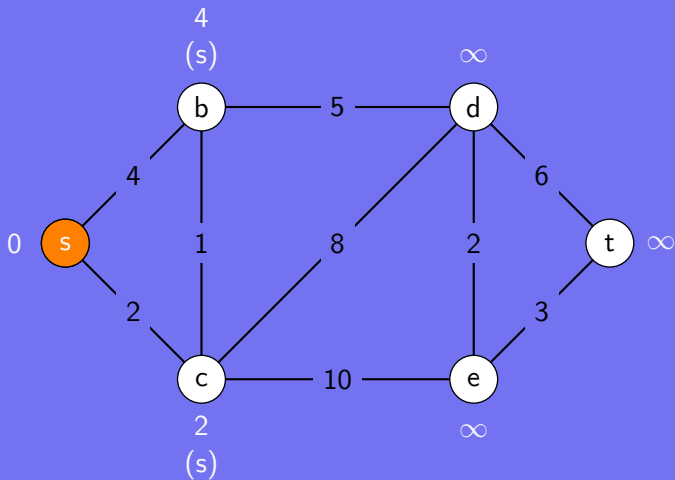
Output : The shortest path between s and t

```
1  $s.dist \leftarrow 0$ ;  $s.prev \leftarrow \text{NULL}$ ;  $S \leftarrow \emptyset$ ;  
2 foreach vertex  $v \in G.V$  do  
3   | if  $v \neq s$  then  $v.dist \leftarrow \infty$ ;  $v.prev \leftarrow \text{NULL}$ ;  
4   |   add  $v$  to  $S$ ;  
5 end foreach  
6  $v \leftarrow s$ ;  
7 repeat  
8   | foreach neighbor  $u$  of  $v$  do  
9   |   |  $tmp \leftarrow v.dist + \text{weight}(v, u)$ ;  
10  |   | if  $tmp < u.dist$  then  $u.dist \leftarrow tmp$ ;  $u.prev \leftarrow v$ ;  
11  |   end foreach  
12  |   remove  $v$  from  $S$ ;  $v \leftarrow$  vertex with minimal distance in  $S$ ;  
13 until  $v = t$ ;  
14 return  $t, t.prev, \dots, s$ 
```

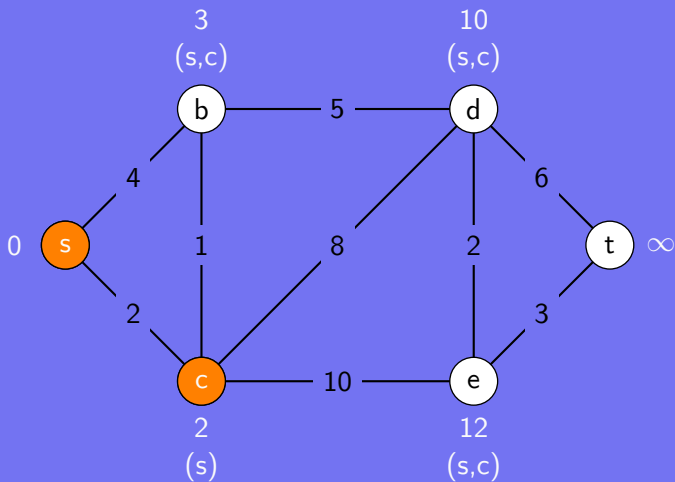
$$S = \{s, b, c, d, e, t\}$$



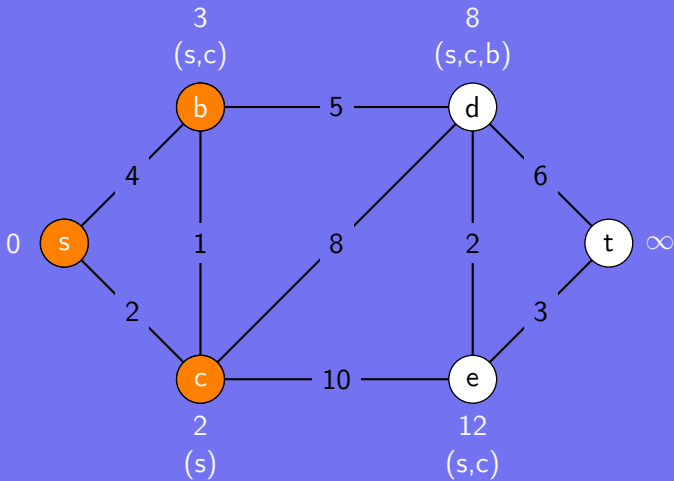
$$S = \{b, c, d, e, t\}$$



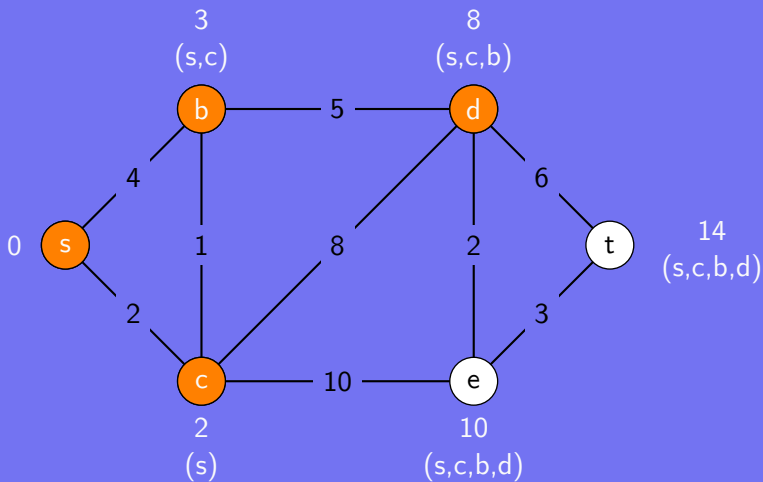
$$S = \{b, d, e, t\}$$



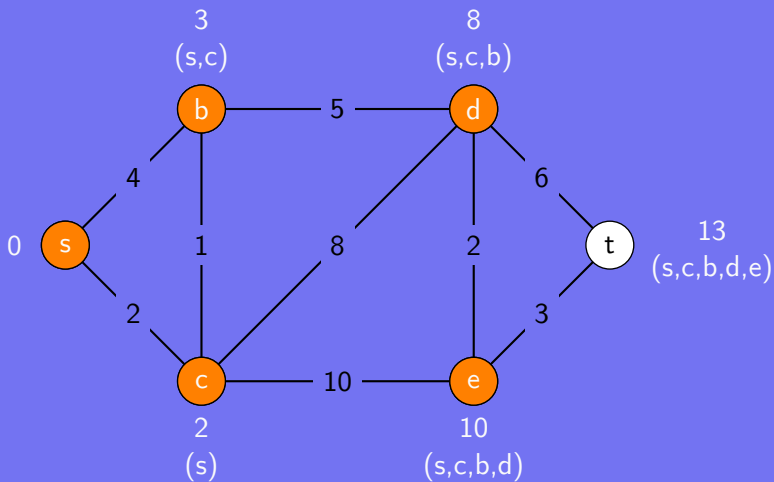
$$S = \{d, e, t\}$$



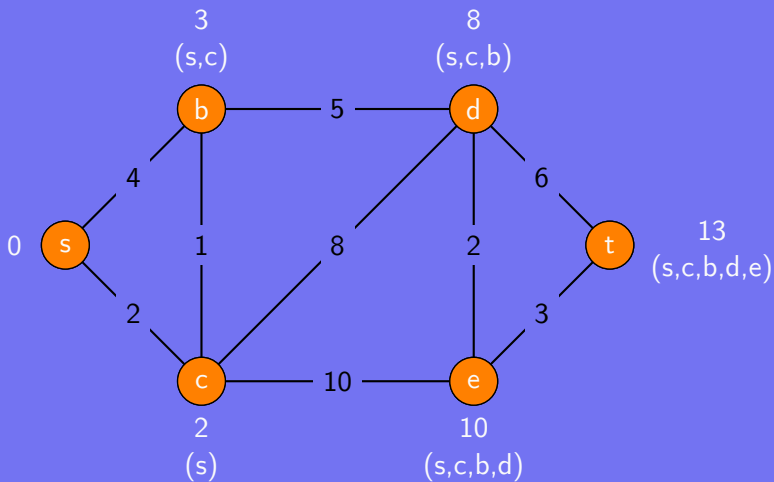
$$S = \{e, t\}$$



$$S = \{t\}$$



$$S = \{t\}, v = t$$



Proposition

If G is a graph with no negative cycle, then there is a shortest simple path going from a source vertex s to a target vertex t .

Proof. Assume no cycle of negative cost exists. If the path repeated a vertex then removing edges to break this cycle would result in a path of no greater cost and with fewer edges. Therefore a shortest simple path exists. \square

Given a graph with n nodes we can find a path of minimum cost which has length at most $n - 1$. Let $opt(i, v)$ denote the minimum cost of a path from a vertex v to a vertex t and featuring at most i edges. The goal is to express $opt(i, v)$ into smaller subproblems such as to determine $opt(n - 1, s)$ using dynamic programming.

Lemma

Let P be an optimal path $opt(i, v)$ in a graph G , and (v, w) be the first edge in P . Then

$$opt(i, v) = \min(opt(i-1, v), opt(i-1, w) + \text{weight}(v, w))$$

Proof. The recursive formula is clear as soon as one notes that two cases can occur.

First if P has at most $i-1$ edges then $opt(i, v) = opt(i-1, v)$. On the other hand if P has i edges and the first one is (v, w) , then

$$opt(i, v) = \text{weight}(v, w) + opt(i-1, w).$$



Algorithm. (*Bellman-Ford*)

Input : A directed weighted graph $G = \langle V, E \rangle$ with no negative cycle, two vertices s and t

Output: The shortest path between s and t

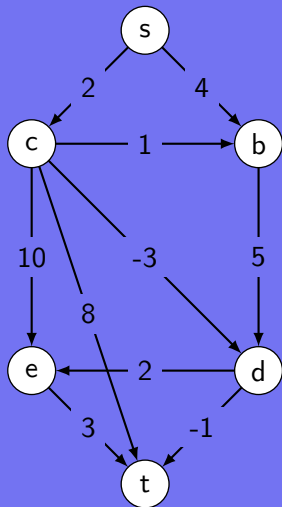
```
1 foreach  $v \in G.V$  do
2   | if  $v \neq s$  then  $v.dist \leftarrow \infty$ ;  $v.prev = \text{NULL}$ ;
3   | else  $v.dist \leftarrow 0$ ;
4 end foreach
5 for  $i \leftarrow 1$  to  $|G.V| - 1$  do
6   | foreach  $(u, v) \in G.E$  do
7   |   |  $tmp \leftarrow u.dist + \text{weight}(u, v)$ ;
8   |   | if  $tmp < v.dist$  then  $v.dist \leftarrow tmp$ ;  $v.prev = u$ ;
9   | end foreach
10 end for
11 return  $t, t.prev, \dots, s$ 
```

Theorem

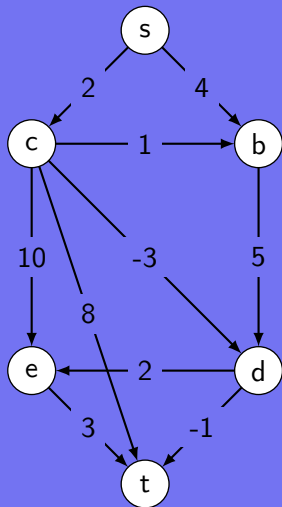
Bellman-Ford algorithm is correct and runs in time $\mathcal{O}(mn)$ on a graph composed of n vertices and m edges.

Proof. The correctness of the algorithm follows from the recursive formula introduced in lemma 3.16.

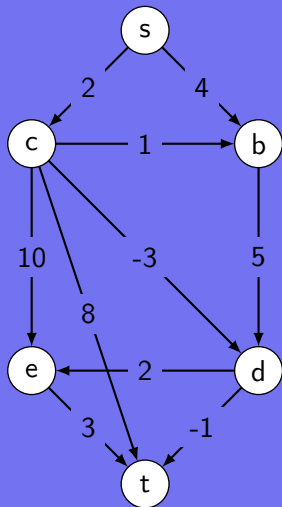
From lines 5 to 9 of Bellman-Ford algorithm (3.17) the loop on all the m edges is applied n times. \square



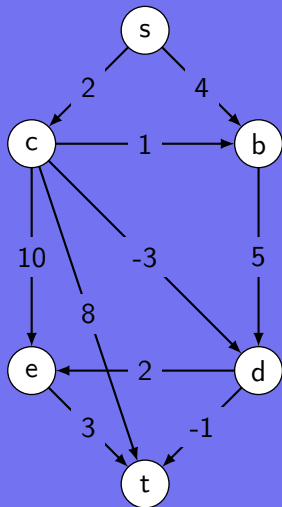
| | 0 | 1 | 2 | 3 | 4 | 5 |
|----------|----------|---|---|---|---|---|
| <i>s</i> | 0 | | | | | |
| <i>b</i> | ∞ | | | | | |
| <i>c</i> | ∞ | | | | | |
| <i>d</i> | ∞ | | | | | |
| <i>e</i> | ∞ | | | | | |
| <i>t</i> | ∞ | | | | | |



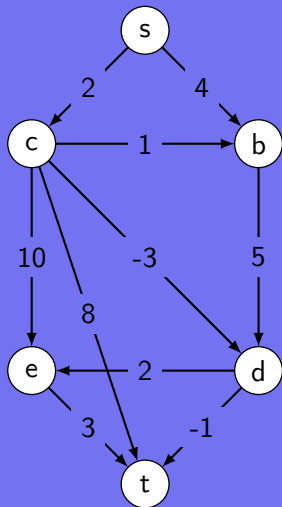
| | 0 | 1 | 2 | 3 | 4 | 5 |
|----------|----------|----------|---|---|---|---|
| <i>s</i> | 0 | 0 | | | | |
| <i>b</i> | ∞ | 4 | | | | |
| <i>c</i> | ∞ | 2 | | | | |
| <i>d</i> | ∞ | ∞ | | | | |
| <i>e</i> | ∞ | ∞ | | | | |
| <i>t</i> | ∞ | ∞ | | | | |



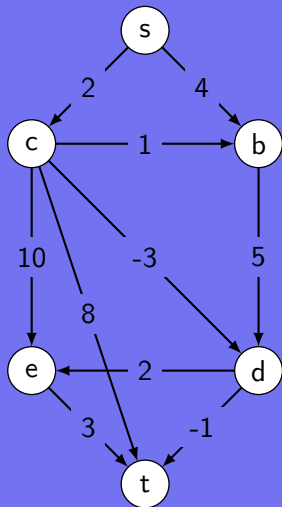
| | 0 | 1 | 2 | 3 | 4 | 5 |
|----------|----------|----------|----|---|---|---|
| <i>s</i> | 0 | 0 | 0 | | | |
| <i>b</i> | ∞ | 4 | 3 | | | |
| <i>c</i> | ∞ | 2 | 2 | | | |
| <i>d</i> | ∞ | ∞ | -1 | | | |
| <i>e</i> | ∞ | ∞ | 12 | | | |
| <i>t</i> | ∞ | ∞ | 10 | | | |



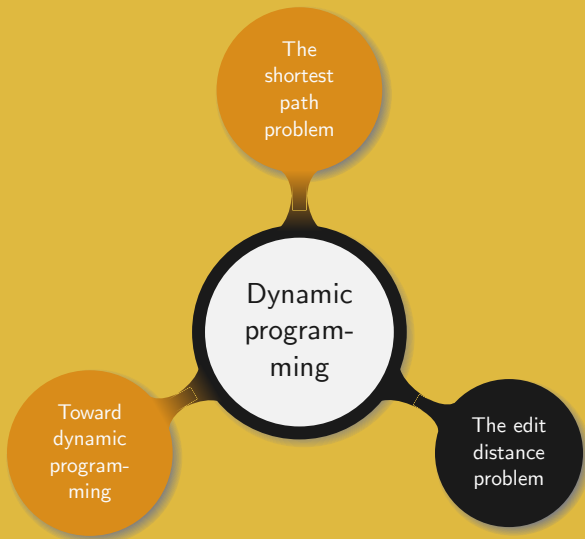
| | 0 | 1 | 2 | 3 | 4 | 5 |
|----------|----------|----------|----|----|---|---|
| <i>s</i> | 0 | 0 | 0 | 0 | | |
| <i>b</i> | ∞ | 4 | 3 | 3 | | |
| <i>c</i> | ∞ | 2 | 2 | 2 | | |
| <i>d</i> | ∞ | ∞ | -1 | -1 | | |
| <i>e</i> | ∞ | ∞ | 12 | 1 | | |
| <i>t</i> | ∞ | ∞ | 10 | -2 | | |



| | 0 | 1 | 2 | 3 | 4 | 5 |
|----------|----------|----------|----|----|----|---|
| <i>s</i> | 0 | 0 | 0 | 0 | 0 | |
| <i>b</i> | ∞ | 4 | 3 | 3 | 3 | |
| <i>c</i> | ∞ | 2 | 2 | 2 | 2 | |
| <i>d</i> | ∞ | ∞ | -1 | -1 | -1 | |
| <i>e</i> | ∞ | ∞ | 12 | 1 | 1 | |
| <i>t</i> | ∞ | ∞ | 10 | -2 | -2 | |



| | 0 | 1 | 2 | 3 | 4 | 5 |
|----------|----------|----------|----|----|----|----|
| <i>s</i> | 0 | 0 | 0 | 0 | 0 | 0 |
| <i>b</i> | ∞ | 4 | 3 | 3 | 3 | 3 |
| <i>c</i> | ∞ | 2 | 2 | 2 | 2 | 2 |
| <i>d</i> | ∞ | ∞ | -1 | -1 | -1 | -1 |
| <i>e</i> | ∞ | ∞ | 12 | 1 | 1 | 1 |
| <i>t</i> | ∞ | ∞ | 10 | -2 | -2 | -2 |



Given two strings determine whether they match

Comments on the problem:

- Simple to implement
- How to render misspellings?

This is useless in practice where we are more interested in measuring how far two strings are from each others.

Given two strings determine whether they match

Comments on the problem:

- Simple to implement
- How to render misspellings?

This is useless in practice where we are more interested in measuring how far two strings are from each others.

Typical applications:

- Spell checker
- DNA sequencing
- Changes in language usage
- Plagiarism

Problem (Edit distance)

The number of changes that need to be performed to convert a string into another one defines the *distance* between the two strings. The three types of alterations considered are:

- *Substitution*: a single character is replaced by a different one
- *Insertion*: a single character is added such as to decrease the distance between the two strings
- *Deletion*: a single character is deleted in order to match the other string

Given two strings and assigning distance one to each of these operations determine the *edit distance* between them.

Naive idea: search exact places where to add/delete characters

Alternative view:

- What information is needed to decide on what operation to perform?
- What can happen to the last character for each string?

Naive idea: search exact places where to add/delete characters

Alternative view:

- What information is needed to decide on what operation to perform?
- What can happen to the last character for each string?

If an optimal solution is known for all the characters but the last, then it becomes simple to find an overall best solution: check the three possibilities, add the cost of each of them to the previous minimal cost and select the best option.

Given a reference string S and a string T we want to determine their edit distance. At each step, i.e. letter, a decision can be taken upon the previous results:

- If $S_i = T_j$, then consider $dist_{i-1,j-1}$. Otherwise consider $dist_{i-1,j-1}$ and pay a cost 1 for the difference
- If $S_{i-1} = T_j$, then it could be that T has one more character than S . In that case consider $dist_{i-1,j}$ and pay a cost 1 for the insertion of a character in S
- If $S_i = T_{j-1}$, then it could be that T has one less character than S . In that case consider $dist_{i,j-1}$ and pay a cost 1 for the deletion of a character in S

As those three possibilities cover all the cases, taking their minimum yields the edit distance.

Description of the strategy:

- If either of the index is 0 then set *dist* to the other index
- Compute

$$dist_{i,j} = \min(dist_{i-1,j} + 1, dist_{i,j-1} + 1, dist_{i-1,j-1} + (\text{match}(S_i, T_j) ? 0 : 1))$$

How fast would be the algorithm?

Description of the strategy:

- If either of the index is 0 then set *dist* to the other index
- Compute

$$dist_{i,j} = \min(dist_{i-1,j} + 1, dist_{i,j-1} + 1, dist_{i-1,j-1} + (\text{match}(S_i, T_j) ? 0 : 1))$$

How fast would be the algorithm?

- At each position in the string, three branches are explored
- Only one branch reduces both indices
- Exponential time $\Omega(3^n)$

How to do better?

Description of the strategy:

- If either of the index is 0 then set *dist* to the other index
- Compute

$$dist_{i,j} = \min(dist_{i-1,j} + 1, dist_{i,j-1} + 1, dist_{i-1,j-1} + (\text{match}(S_i, T_j) ? 0 : 1))$$

How fast would be the algorithm?

- At each position in the string, three branches are explored
- Only one branch reduces both indices
- Exponential time $\Omega(3^n)$

How to do better?

- What is the maximum number of pairs?
- The same pairs are recalled many times
- Use a lookup table to decrease the computational cost

Algorithm.

Input : Two strings S and T

Output: The edit distance between S and T

```
1 for  $i \leftarrow 0$  to  $|S|$  do  $dist_{i,0} \leftarrow i$ ;
2 for  $i \leftarrow 1$  to  $|T|$  do  $dist_{0,i} \leftarrow i$ ;
3 for  $i \leftarrow 1$  to  $|S|$  do
4   for  $j \leftarrow 1$  to  $|T|$  do
5      $tmp_0 \leftarrow dist_{i-1,j-1} + (\text{match}(S_i, T_j) ? 0 : 1)$ ;
6      $tmp_1 \leftarrow dist_{i,j-1} + 1$ ;           /* skip a letter in  $S$  */
7      $tmp_2 \leftarrow dist_{i-1,j} + 1$ ;       /* skip a letter in  $T$  */
8      $dist_{i,j} \leftarrow \min(tmp_0, tmp_1, tmp_2)$ ;
9   end for
10 end for
11 return  $dist_{i,j}$ 
```

| | | P | O | L | Y | N | O | M | I | A | L |
|---|----|----|---|---|---|---|---|---|---|---|----------|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| E | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| X | 2 | 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| P | 3 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| O | 4 | 3 | 2 | 3 | 4 | 5 | 5 | 6 | 7 | 8 | 9 |
| N | 5 | 4 | 3 | 3 | 4 | 4 | 5 | 6 | 7 | 8 | 9 |
| E | 6 | 5 | 4 | 4 | 4 | 5 | 5 | 6 | 7 | 8 | 9 |
| N | 7 | 6 | 5 | 5 | 5 | 4 | 5 | 6 | 7 | 8 | 9 |
| T | 8 | 7 | 6 | 6 | 6 | 5 | 5 | 6 | 7 | 8 | 9 |
| I | 9 | 8 | 7 | 7 | 7 | 6 | 6 | 6 | 6 | 7 | 8 |
| A | 10 | 9 | 8 | 8 | 8 | 7 | 7 | 7 | 7 | 6 | 7 |
| L | 11 | 10 | 9 | 8 | 9 | 8 | 8 | 8 | 8 | 7 | 6 |

Theorem

Let S and T be two strings of length n and m , respectively. Algorithm 3.26 solves edit distance in time $\mathcal{O}(nm)$.

Proof. Algorithm 3.26 is simply a table look up version of the recursive algorithm described on slide 3.24. As all the cases are considered in the initial recursive approach they are also all covered in Algorithm 3.26.

Looking at the description of the algorithm it is clear that most of the work is performed in the nested for loops. This results in the creation of a $n \times m$ table. All the other operations are only reading from this lookup table. Therefore the complexity is $\mathcal{O}(nm)$. \square

Dynamic programming:

- Simple idea dramatically improving space-time complexity
- Cover all the possibilities at the subproblems level
- Never recompute anything that is already known
- Efficient as long as the number of subproblems remains polynomial

1001011111010100010101001111000111000100010111000000100110100001111101
1010010110011100111011101010001110000010001101110101011011111101010000
1101110011100011010011010110011011010101010100111101000010101010101000
00011001111000100000011010101101111011110011000000101010101110100001
1100101110000100001001010110000011100010101000100110110010001101100100
01010110101110000010101100100101111000101001111000001100111001110100
01100011111010000011010011111000110111001100110111101001101
11101001000001001111011110100110000110011001001001111010001001001
100111100011111010010101110100111101000111100111101110100010
011111011111100111100100100100110100110010010001111110000110
111011011001111000010010101011011001101011111011111011111010
1101101000100110010100100111100001010011100001000101110010011011010011
0000110101010001100110000111000100101110011110000101001101100100000110
110111101001010101110101111100001010001001110110000001011011110000011
0111101000100101011011001110011101011011110010110101011001100110101100
10110110110111110100000000111011100011110101110110011100111010011100000

Thank you!