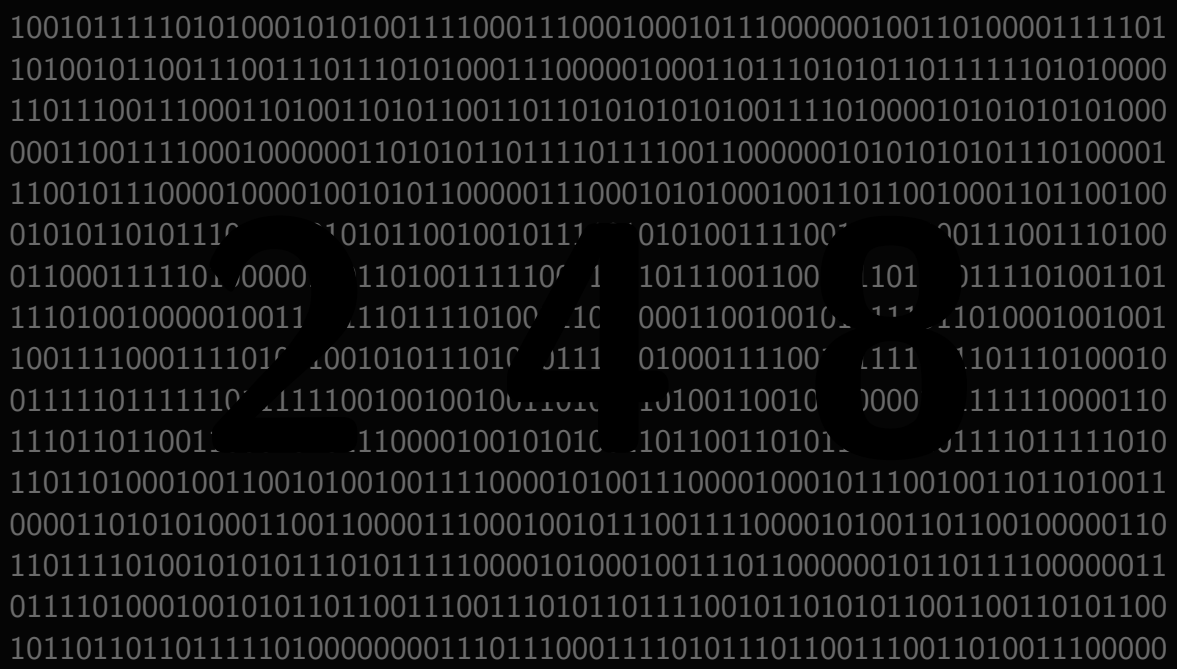
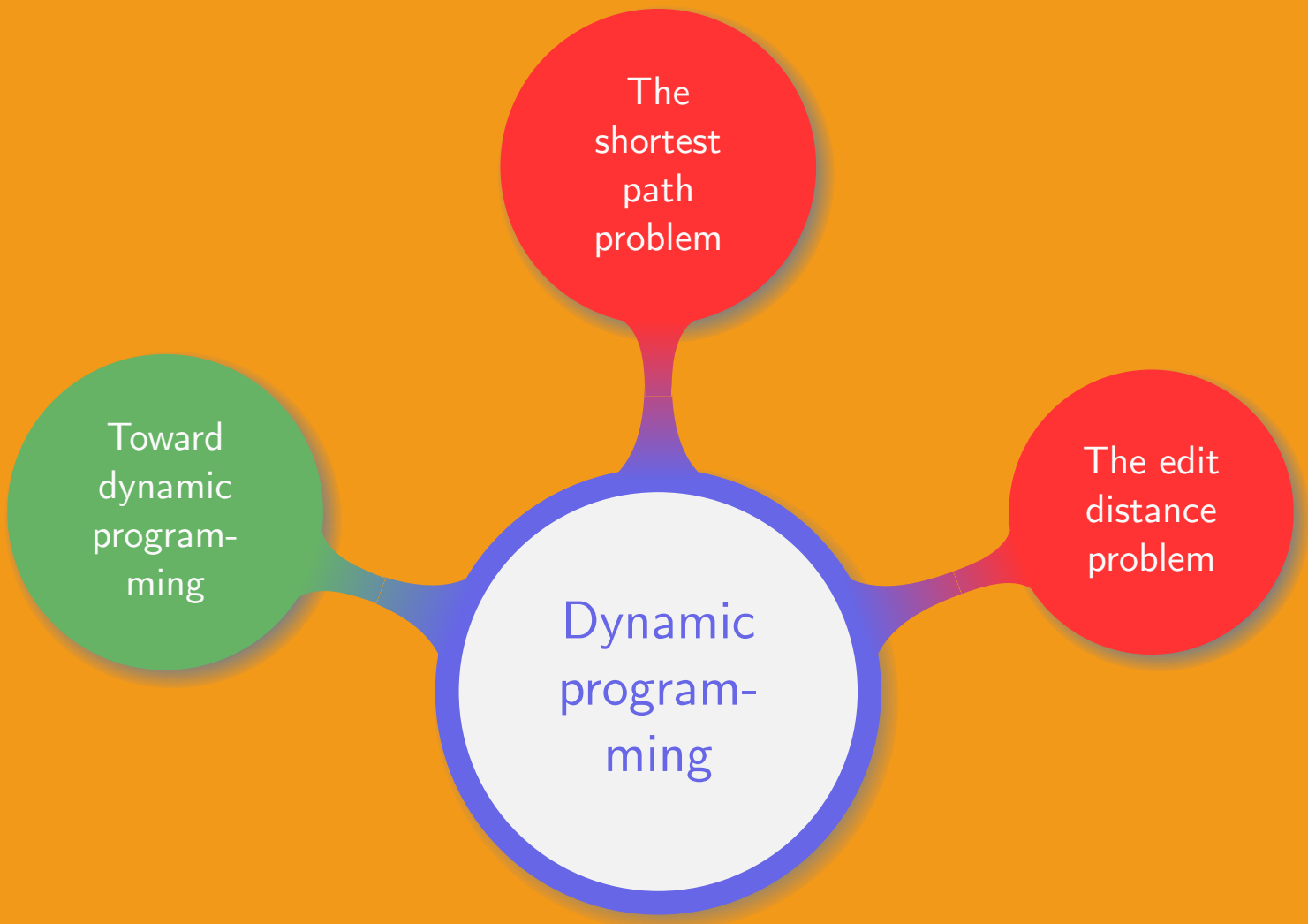


Introduction to Algorithms

3. Dynamic programming

Manuel – Fall 2020

Chapter organisation



Fibonacci Numbers

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.

Fibonacci Numbers

- 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,

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

A very naive approach

Algorithm.

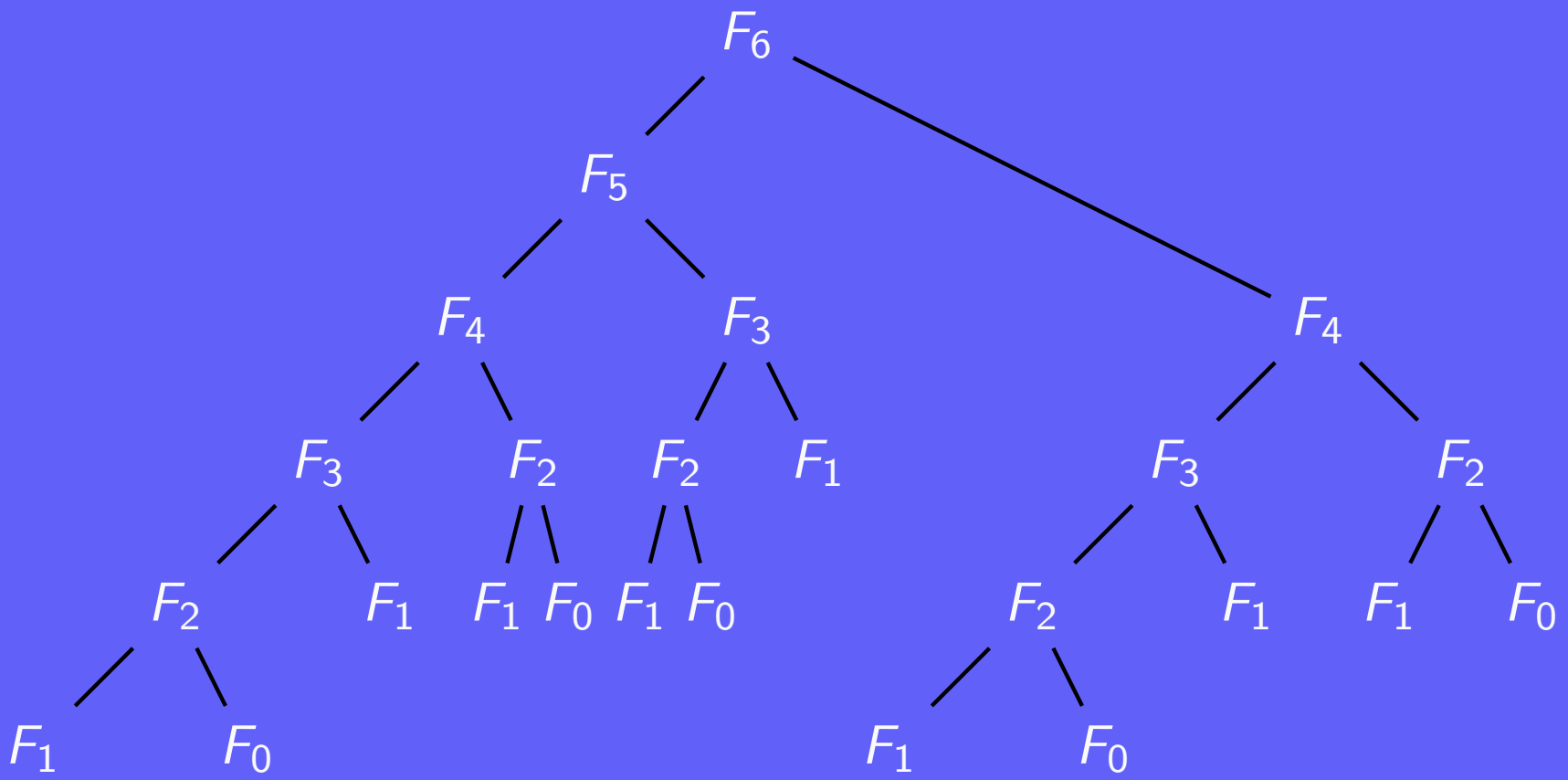
Input : An integer n
Output: F_n

```
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.

A very naive approach



A naive approach

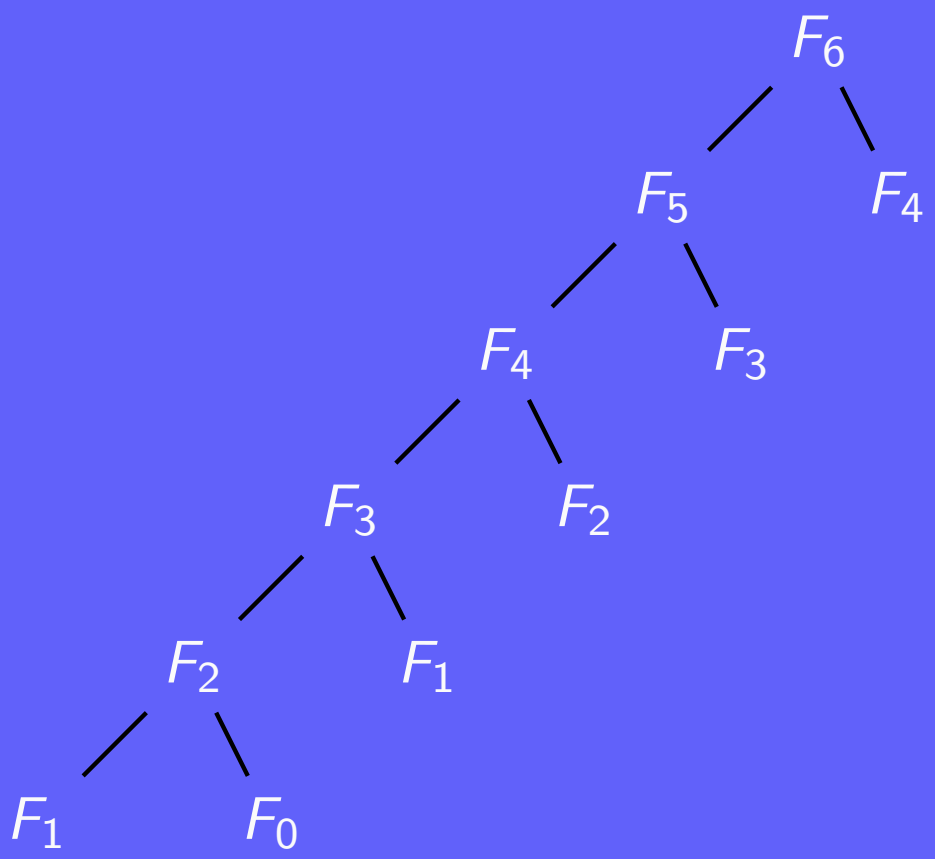
Algorithm.

Input : An integer n
Output: F_n

```
1  Function Fib_n(n):
2  |   if F_n = −1 then F_n ← Fib_n(n-1) + Fib_n(n-2);
3  |   return F_n
4  end
5  Function Fib_nm(n):
6  |   F_0 ← 0;
7  |   F_1 ← 1;
8  |   for i ← 2 to n do F_i ← −1;
9  |   return Fib_n(n)
10 end
```

Both the time and space complexities are linear in n

A naive approach



A proper approach

Algorithm.

Input : An integer n
Output: F_n

```
1 Function Fib(n):
2 |   F_old2 ← 0; F_old1 ← 1;
3 |   if n = 0 then return 0;
4 |   for i ← 2 to n do
5 |   |   F ← F_old1 + F_old2; F_old2 ← F_old1; F_old1 ← F;
6 |   end for
7 |   return F_old1 + F_old2
8 end
```

The time is linear in n and the storage constant

Dynamic programming

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

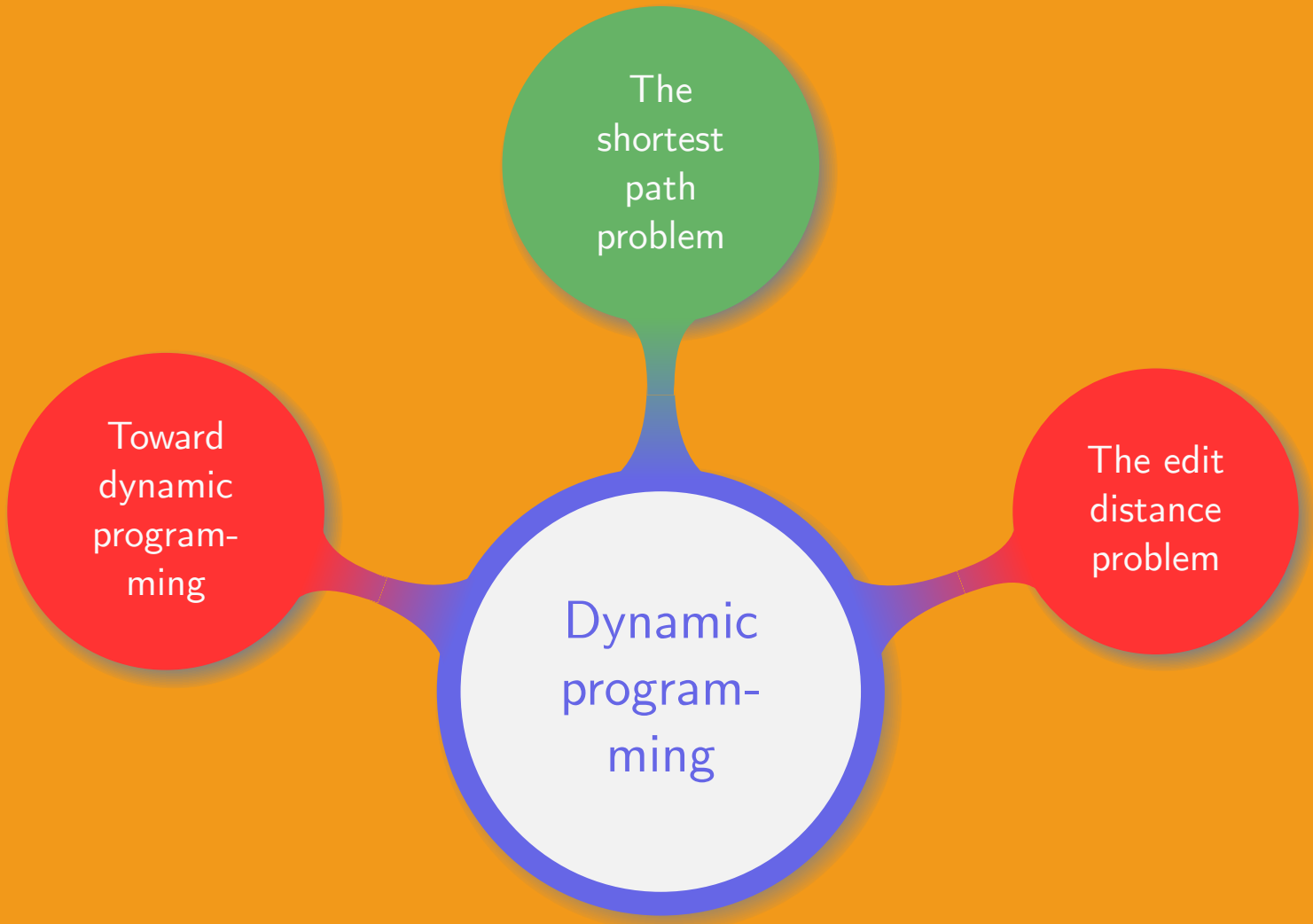
Dynamic programming

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

Chapter organisation



Shortest paths in weighted graphs

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.

Dijkstra's Algorithm

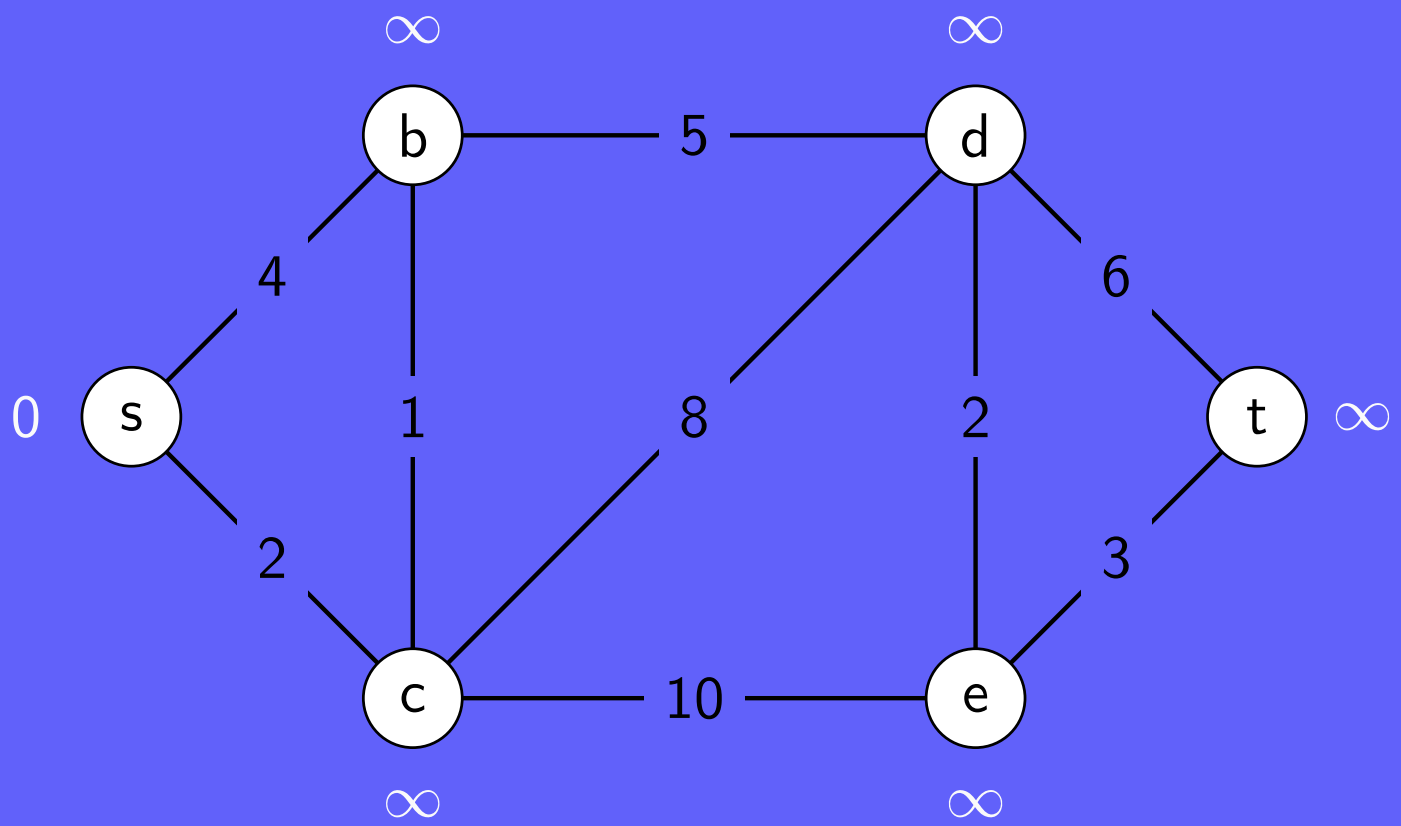
Algorithm. (*Dijkstra*)

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 ← 0; s.prev ← NULL; S ← ∅;
2  foreach vertex v ∈ G.V do
3  |   if v ≠ s then v.dist ← ∞; v.prev ← NULL;
4  |   add v to S;
5  end foreach
6  v ← s;
7  repeat
8  |   foreach neighbor u of v do
9  |   |   tmp ← v.dist+weight(v, u);
10 |   |   if tmp < u.dist then u.dist ← tmp; u.prev ← v;
11 |   end foreach
12 |   remove v from S; v ← vertex with minimal distance in S;
13 until v = t;
14 return t, t.prev, ··· , s
```

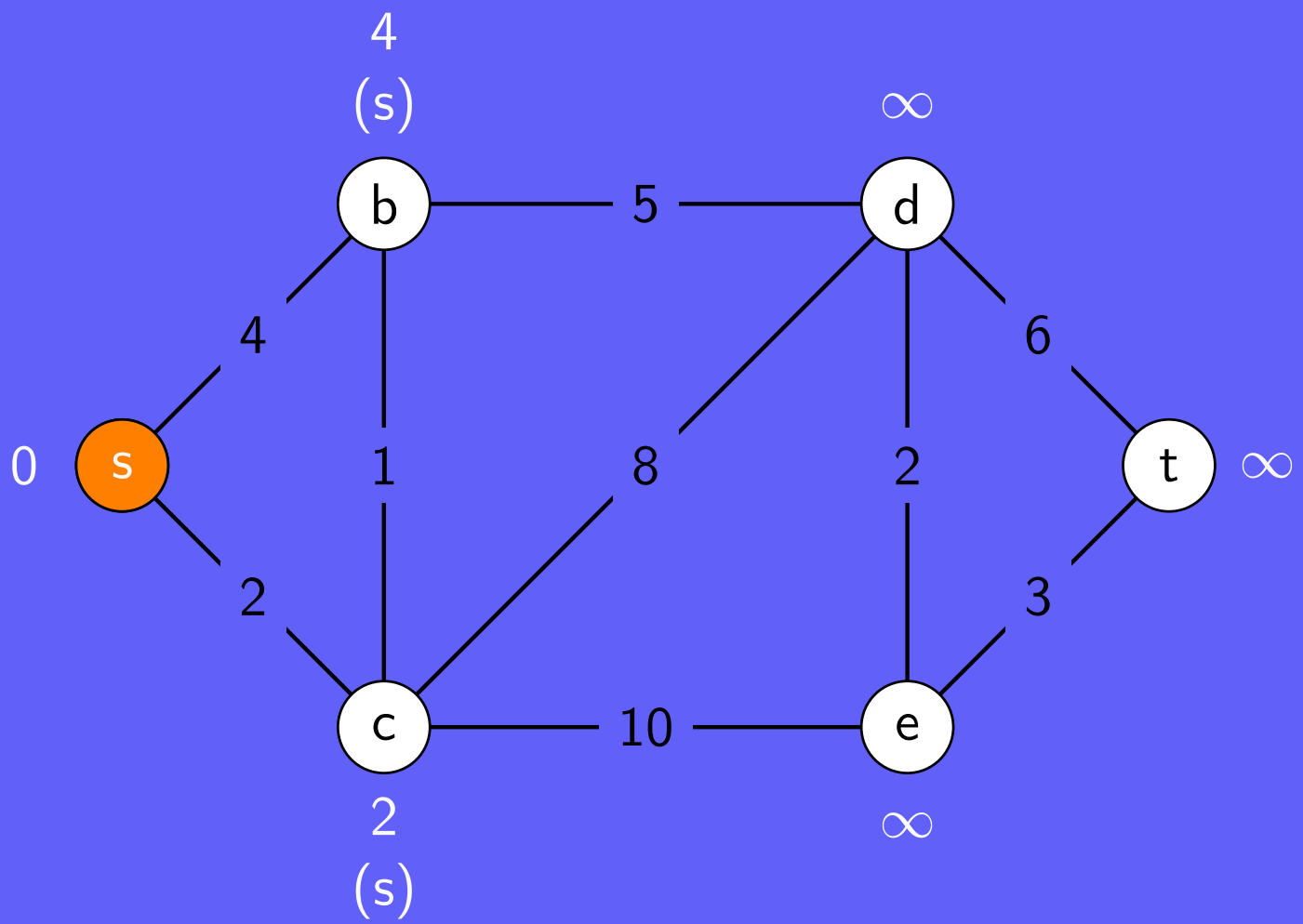
Example

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



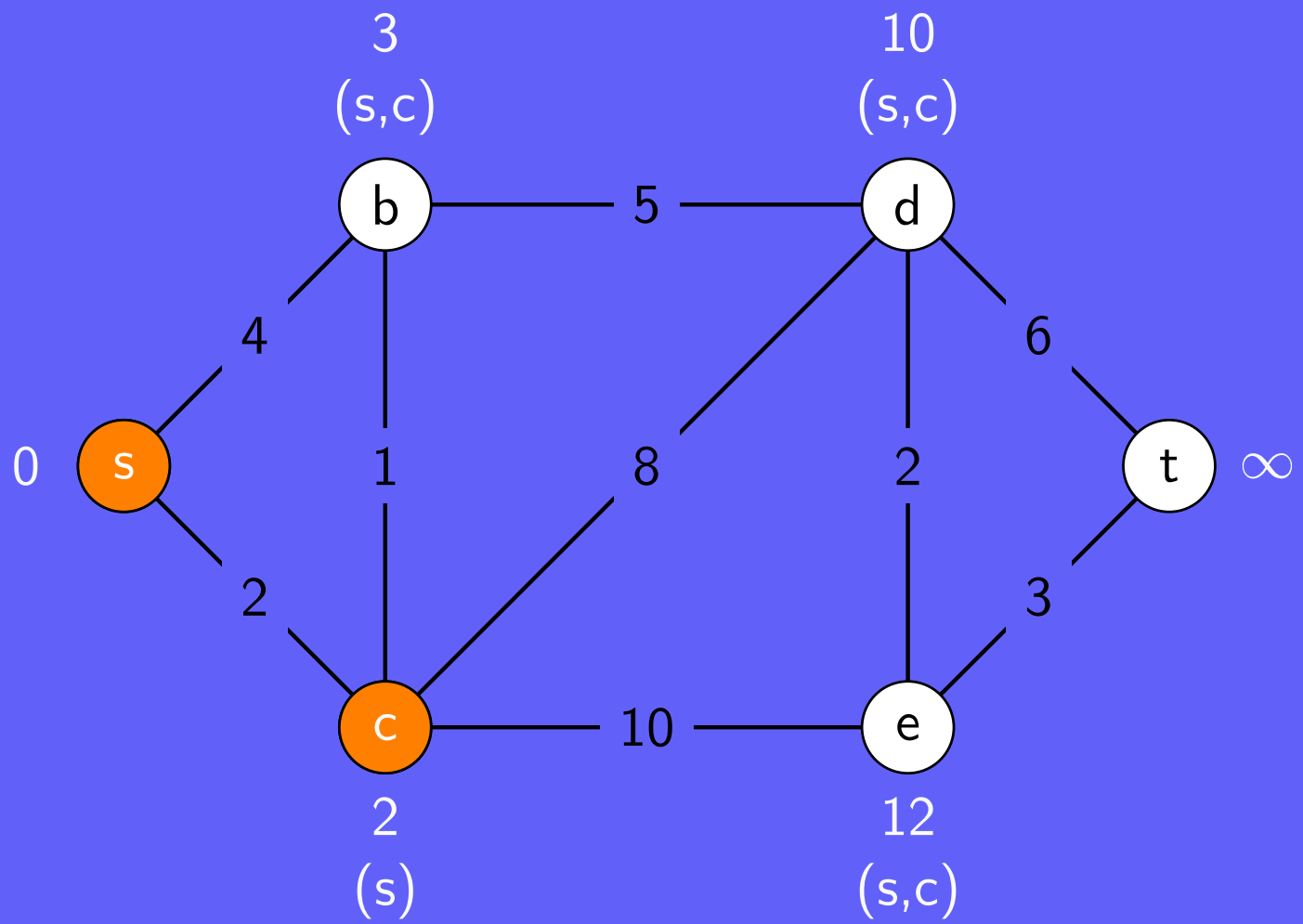
Example

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



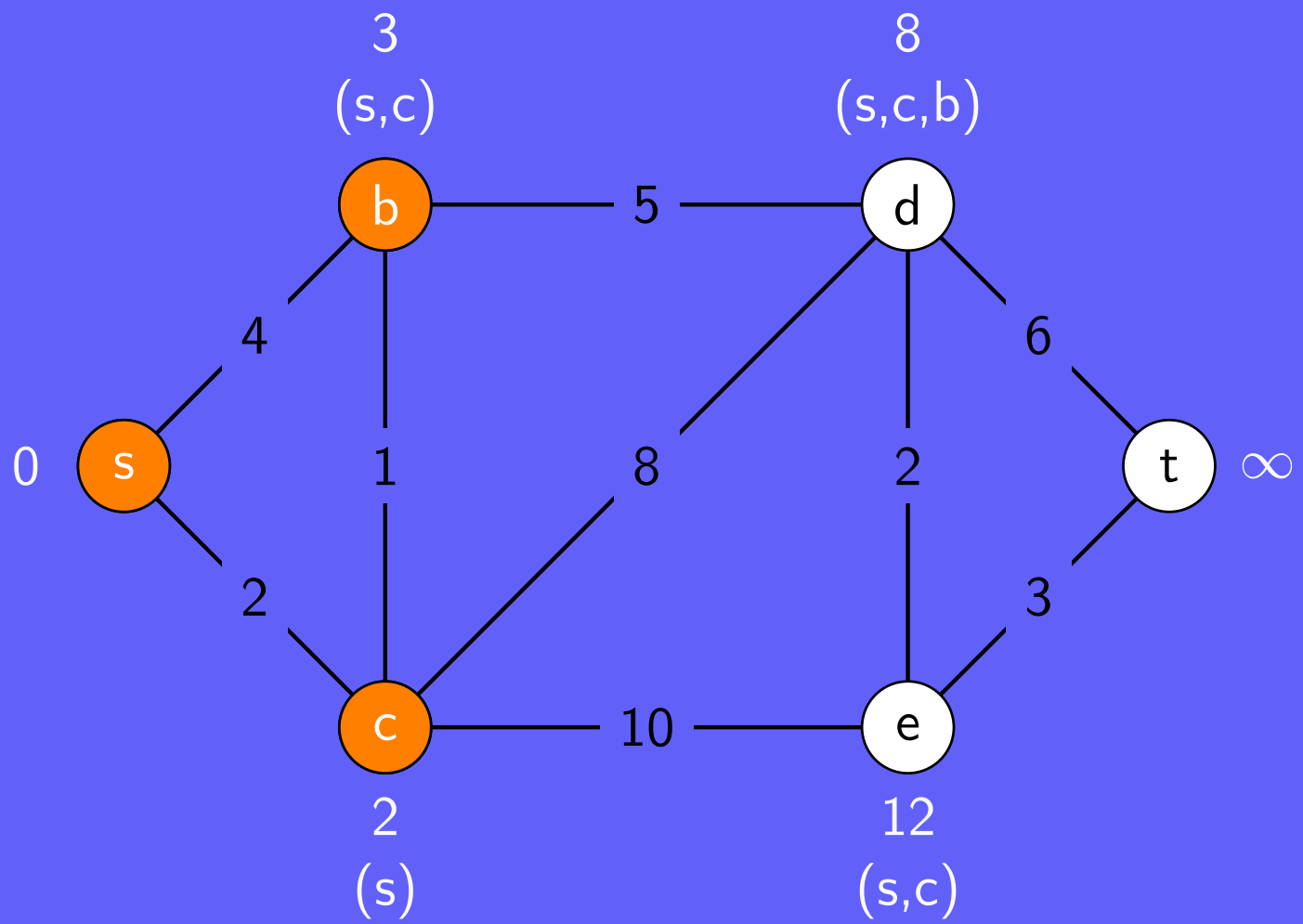
Example

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



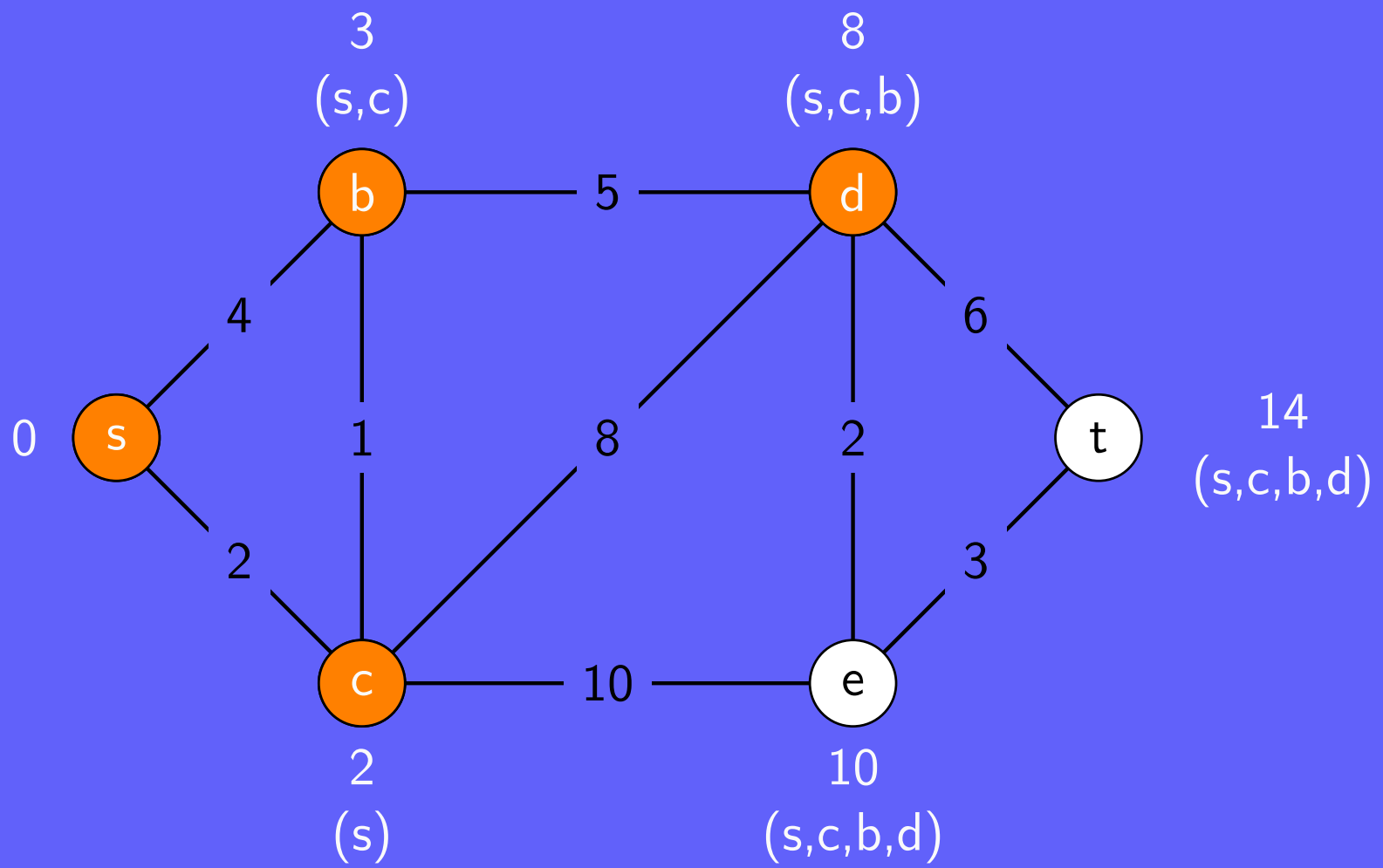
Example

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



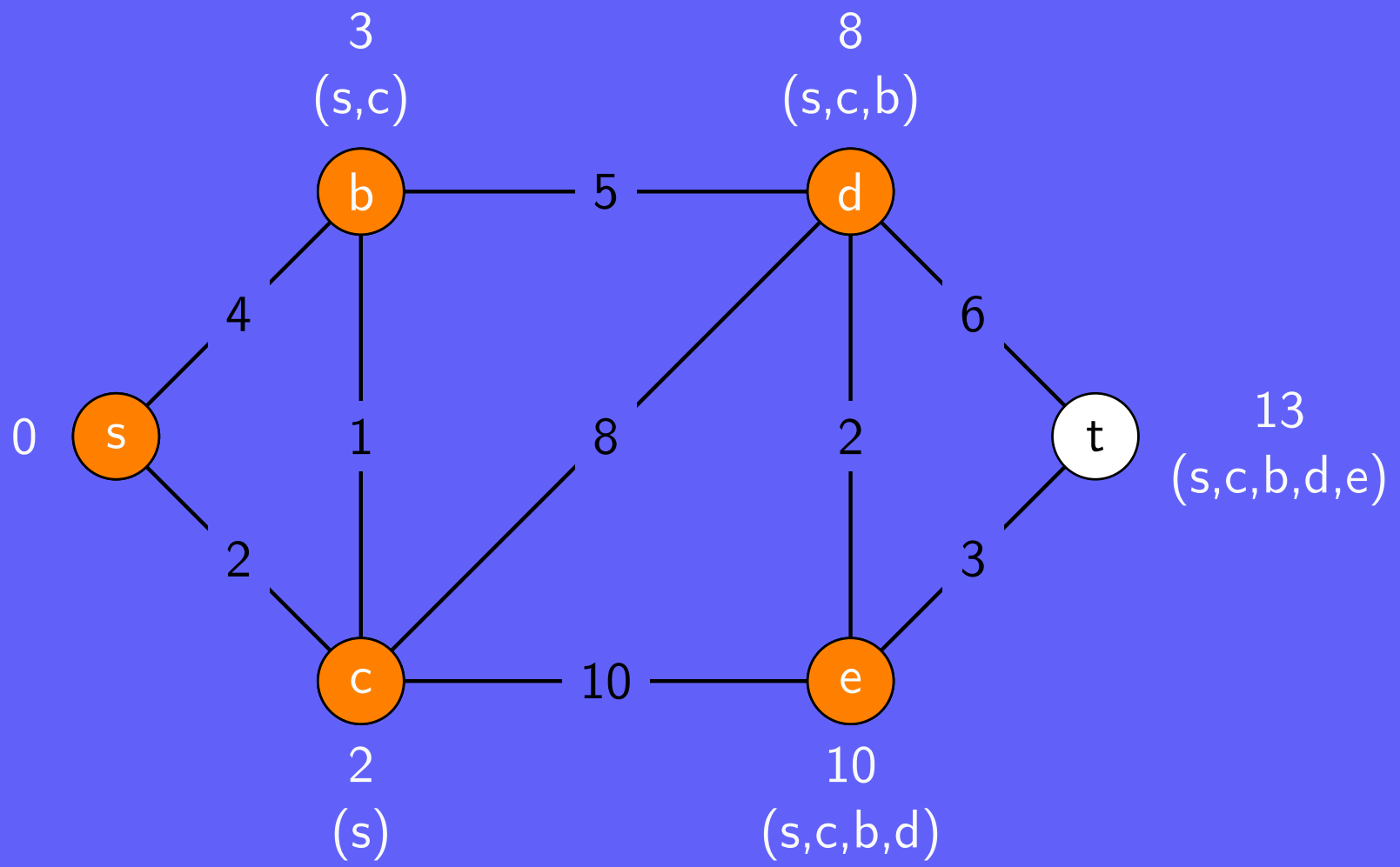
Example

$S = \{e, t\}$



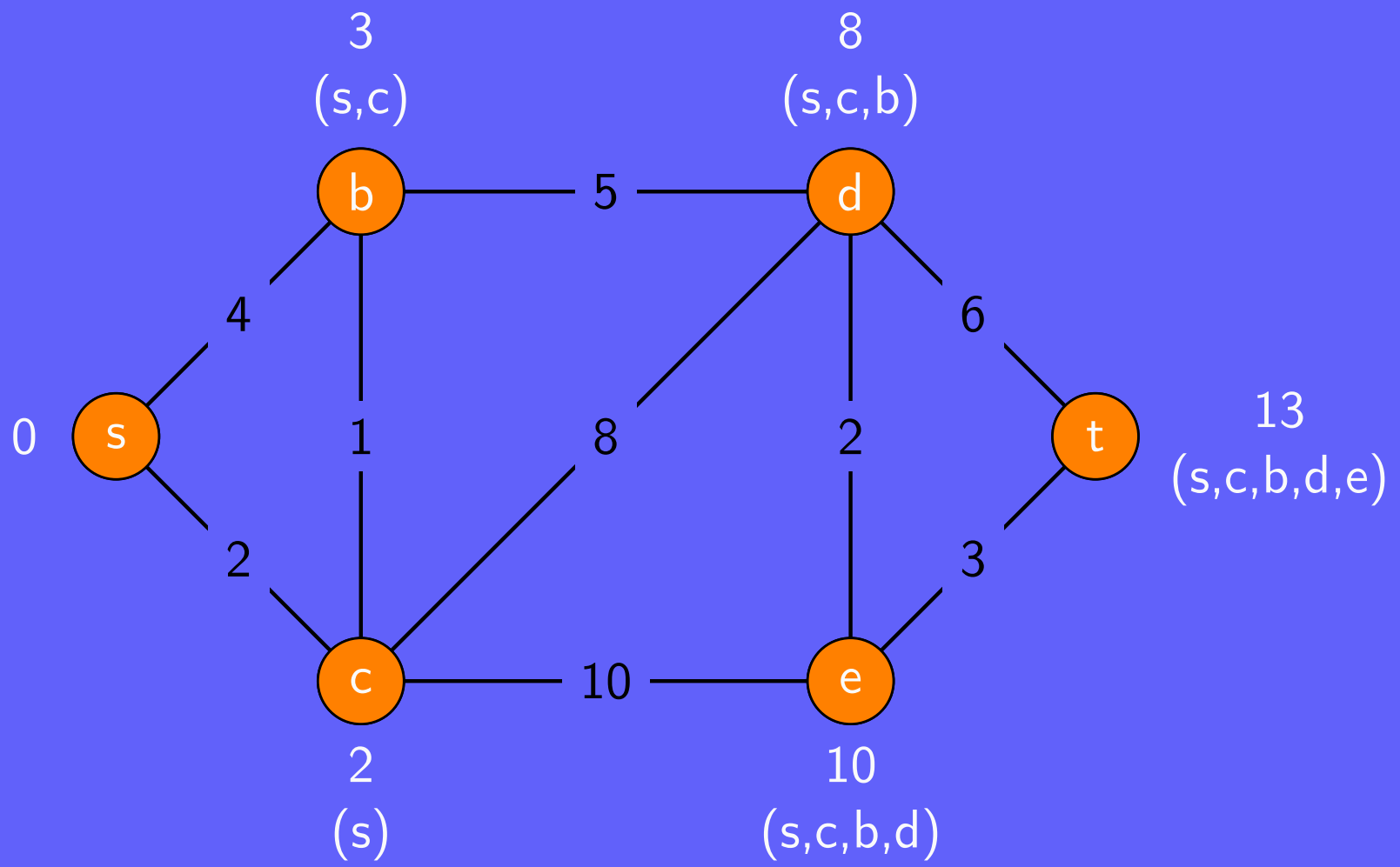
Example

$S = \{t\}$



Example

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



Shortest path with negative edges

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. □

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 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.

Shortest path with negative edges

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

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

□

Bellman-Ford algorithm

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 ∈ G.V do
2  |   if v ≠ s then v.dist ← ∞; v.prev = NULL;
3  |   else v.dist ← 0;
4  end foreach
5  for i ← 1 to |G.V| − 1 do
6  |   foreach (u, v) ∈ G.E do
7  |   |   tmp ← u.dist + weight(u, v);
8  |   |   if tmp < v.dist then v.dist ← tmp; v.prev = u;
9  |   end foreach
10 end for
11 return t, t.prev, ··· , s
```

Bellman-Ford algorithm

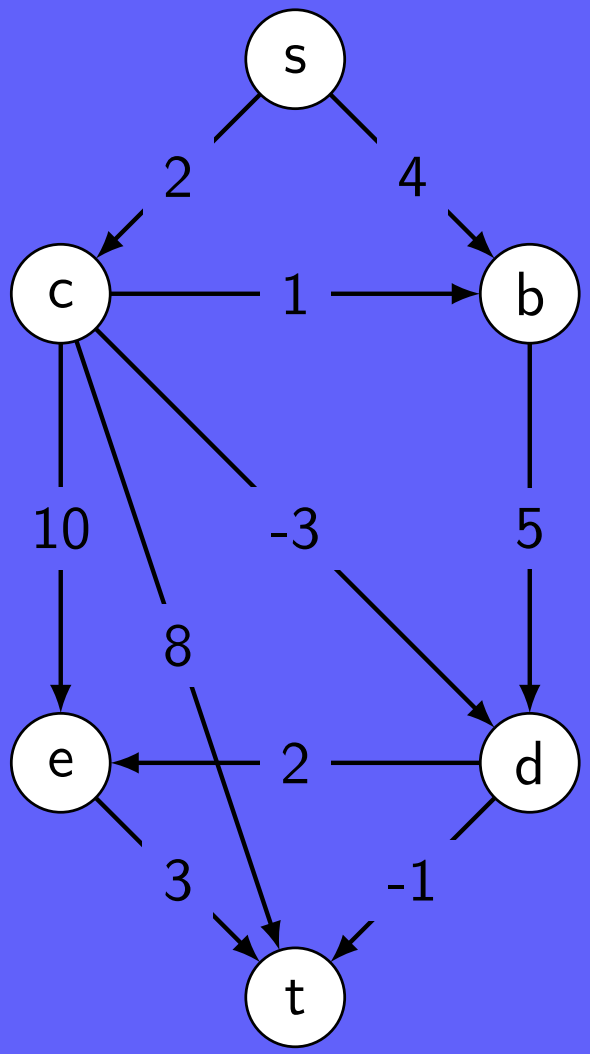
Theorem

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.

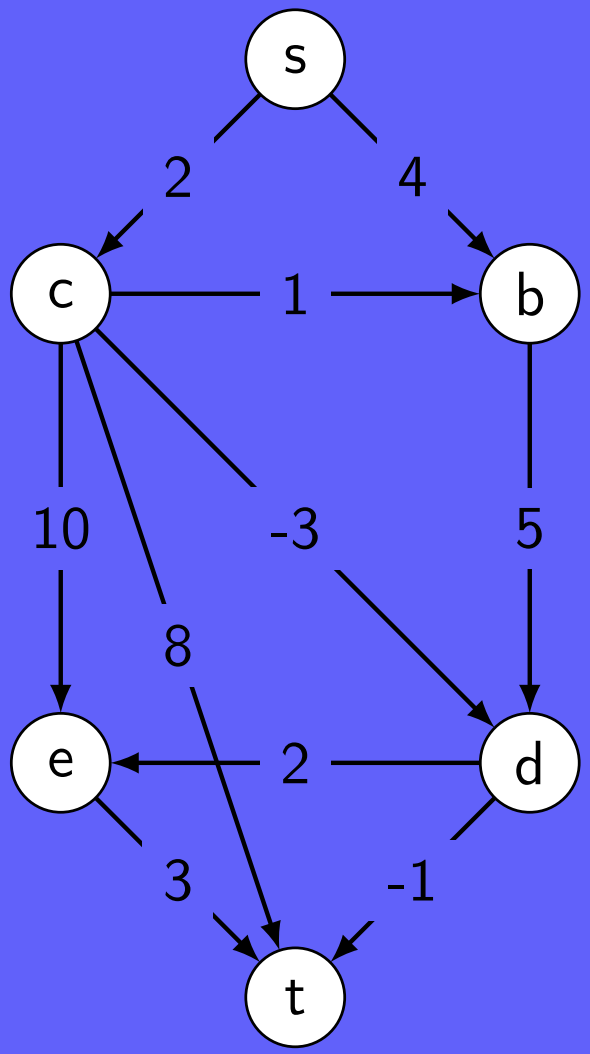
From lines 5 to 9 of Bellman-Ford algorithm (3.17) the loop on all the m edges is applied n times. □

Example



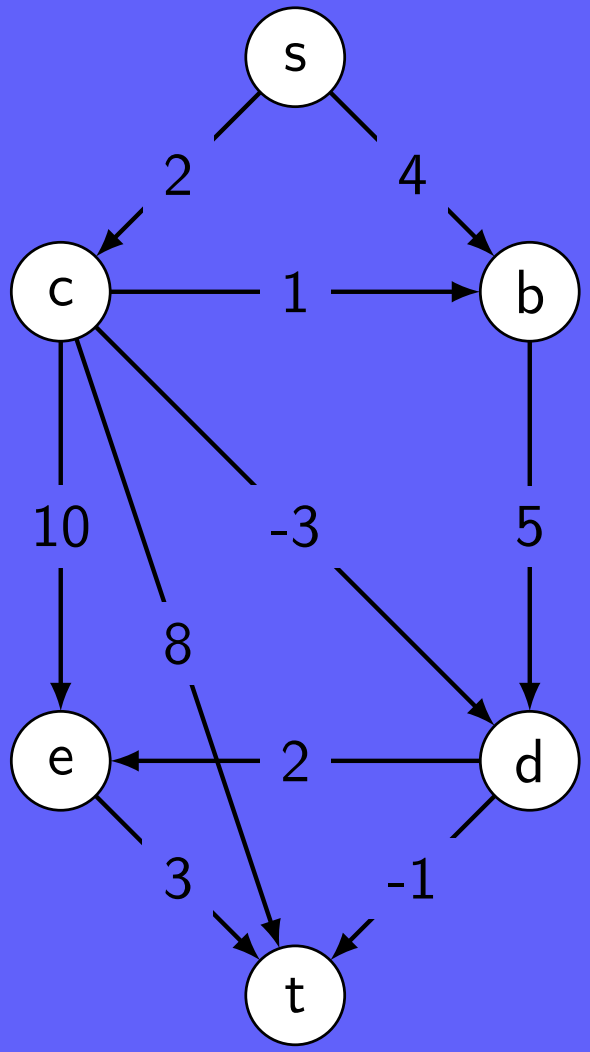
	0	1	2	3	4	5
s	0					
b	∞					
c	∞					
d	∞					
e	∞					
t	∞					

Example



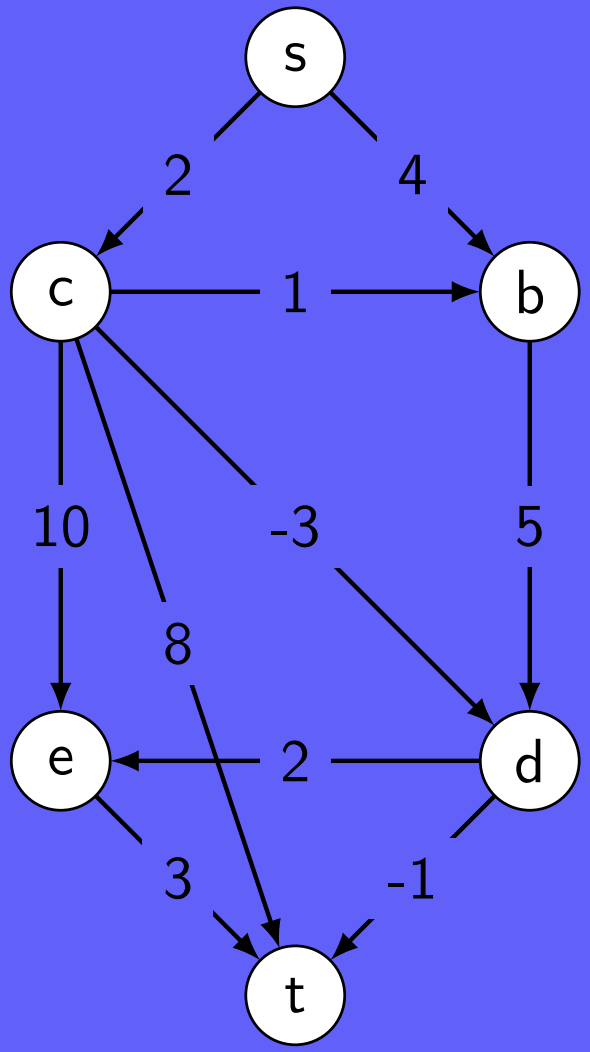
	0	1	2	3	4	5
s	0	0				
b	∞	4				
c	∞	2				
d	∞	∞				
e	∞	∞				
t	∞	∞				

Example



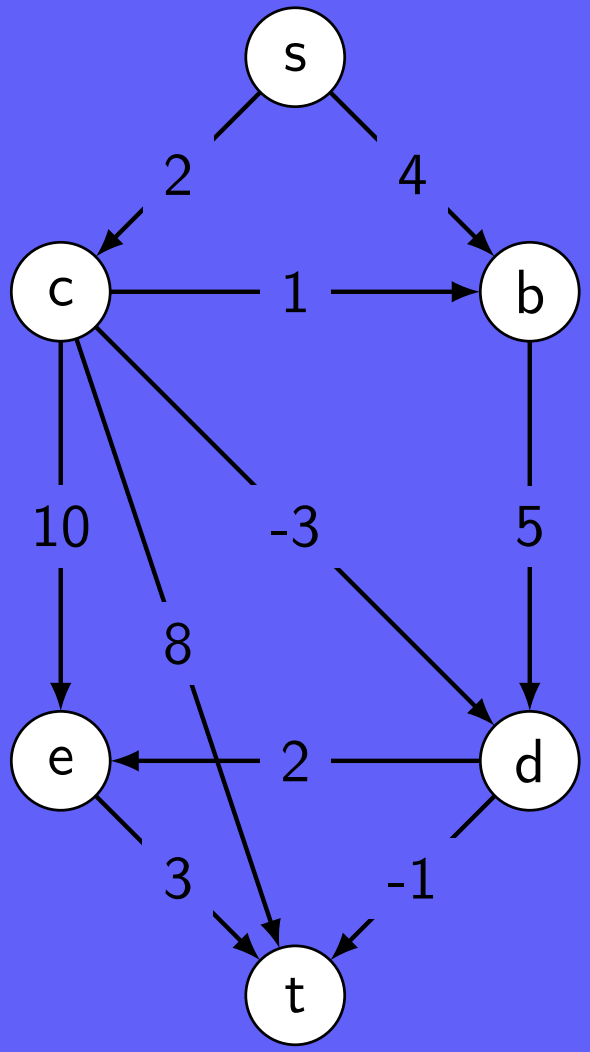
	0	1	2	3	4	5
s	0	0	0			
b	∞	4	3			
c	∞	2	2			
d	∞	∞	-1			
e	∞	∞	12			
t	∞	∞	10			

Example



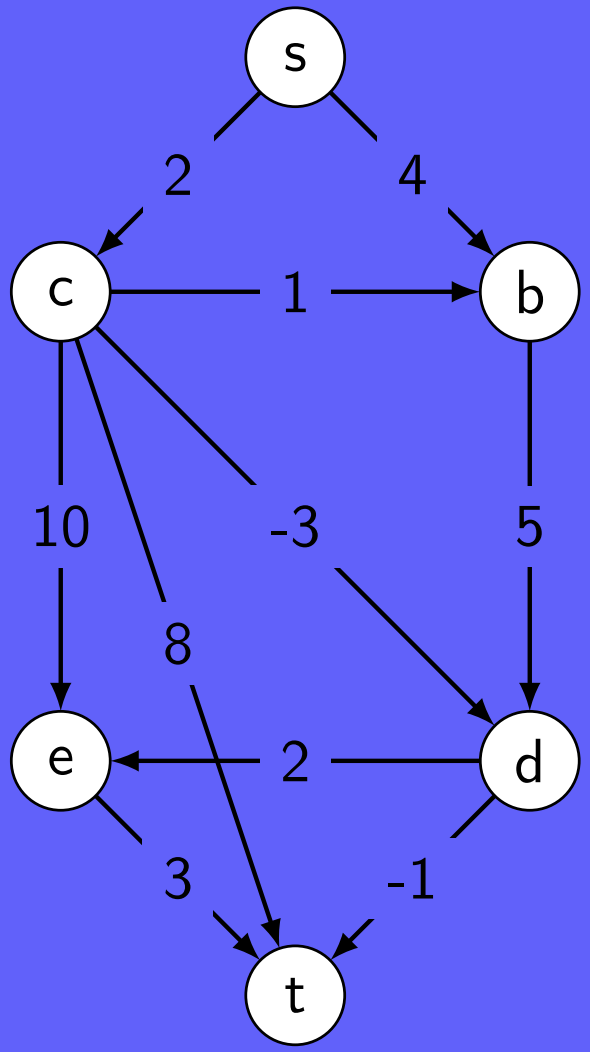
	0	1	2	3	4	5
s	0	0	0	0		
b	∞	4	3	3		
c	∞	2	2	2		
d	∞	∞	-1	-1		
e	∞	∞	12	1		
t	∞	∞	10	-2		

Example



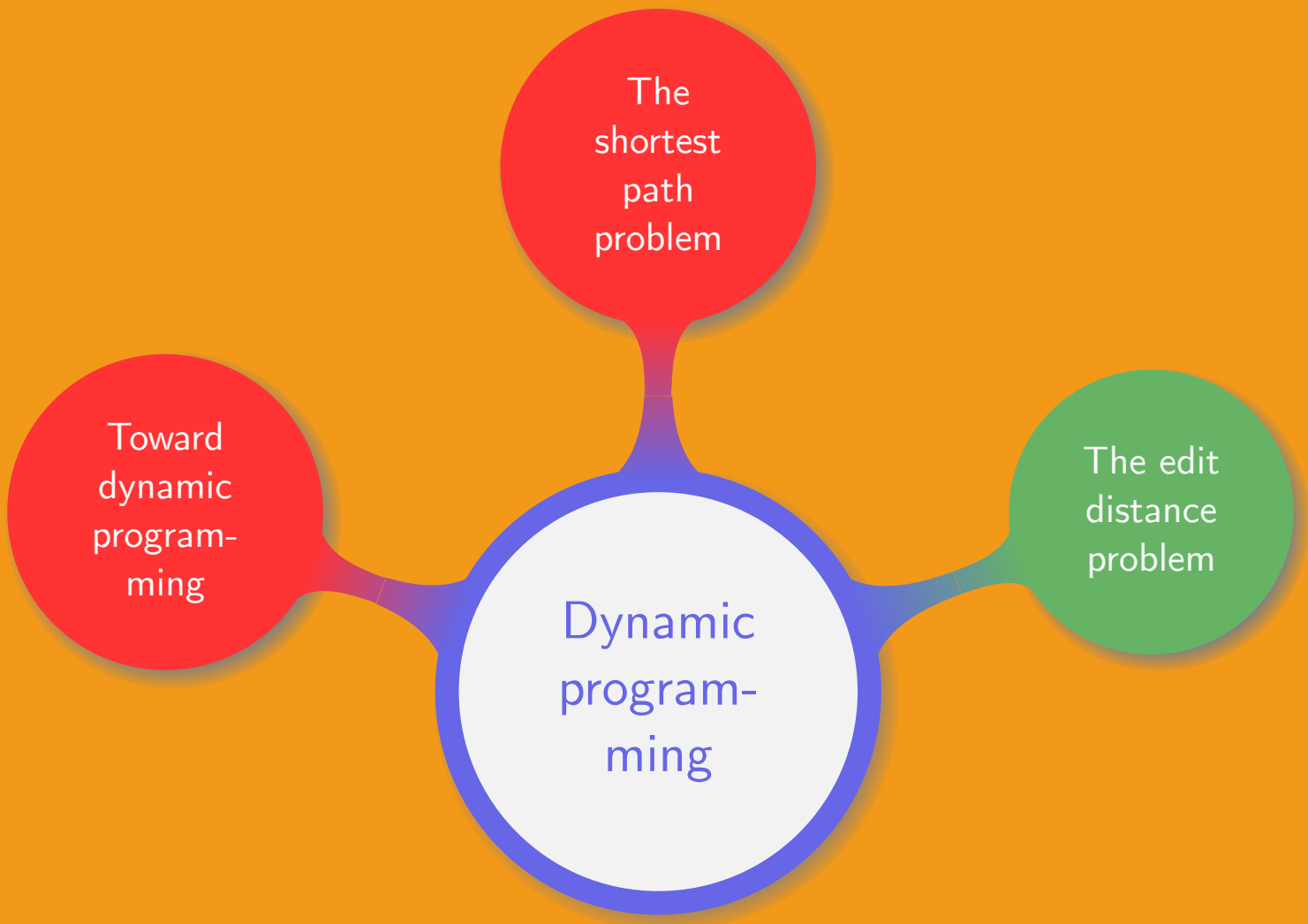
	0	1	2	3	4	5
s	0	0	0	0	0	
b	∞	4	3	3	3	
c	∞	2	2	2	2	
d	∞	∞	-1	-1	-1	
e	∞	∞	12	1	1	
t	∞	∞	10	-2	-2	

Example



	0	1	2	3	4	5
s	0	0	0	0	0	0
b	∞	4	3	3	3	3
c	∞	2	2	2	2	2
d	∞	∞	-1	-1	-1	-1
e	∞	∞	12	1	1	1
t	∞	∞	10	-2	-2	-2

Chapter organisation



Intuition of the problem

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.

Intuition of the problem

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
- Changes in language usage
- DNA sequencing
- Plagiarism

The edit distance problem

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.

Tackling the problem

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?

Tackling the problem

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.

A recursive approach

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.

Limitations of the recursive approach

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?

Limitations of the recursive approach

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?

Limitations of the recursive approach

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

Solving edit distance

Algorithm.

Input : Two strings S and T
Output: The edit distance between S and T

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

Example

		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	6	7	8	9	10
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**

Solving edit distance

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 $\mathcal{O}(nm)$. □

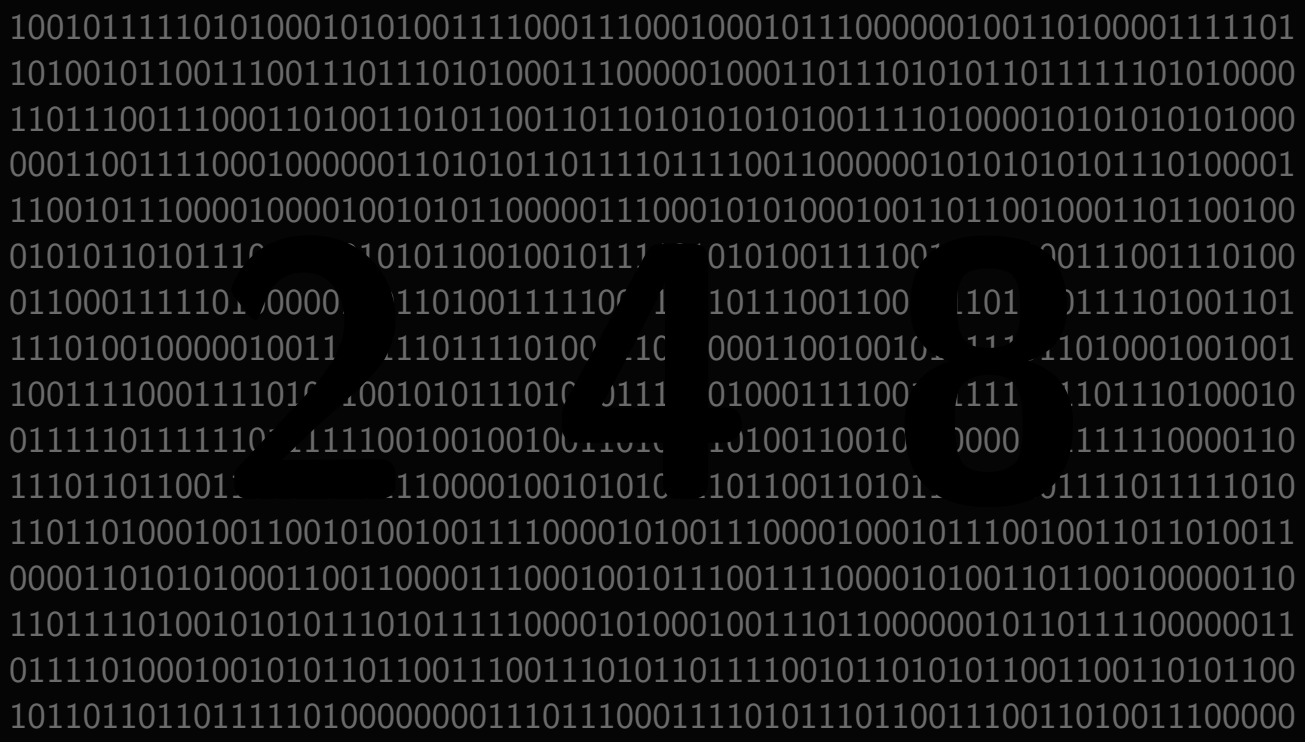
A few more remarks

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

Key points

- Explain dynamic programming in three words
- When to use Dijkstra and Bellman-Ford algorithms?
- Describe the edit distance problem?
- Why is recursion not a good strategy to solve edit distance?



Thank you!