Санкт-Петербургский
государственный
университет

Brute force solutions. Competitive Programming: Core Skills

Artur Riazanov

SPbSU

СПбГУ

Outline

- **Intuitive solutions**
- Search space
- Backtracking

Introduction

- Sometimes the first solution you come up with is the correct one.

Introduction

- Sometimes the first solution you come up with is the correct one.
- But sometimes it's not.

Introduction

- Sometimes the first solution you come up with is the correct one.
- But sometimes it's not.
- In this lesson we are going to develop a method for designing solutions which are **always correct.**

Introduction

- Sometimes the first solution you come up with is the correct one.
- But sometimes it's not.
- In this lesson we are going to develop a method for designing solutions which are **always correct.**
- The catch is they are going to be slow.

Digits ordering

Largest number

Input: list of digits.
Output: the largest number that can be made of the digits.

Digits ordering

Largest number

Input: list of digits.
Output: the largest number that can be made of the digits.

Sample

Input: 3, 7, 5
Output: 735

Digits ordering

Largest number

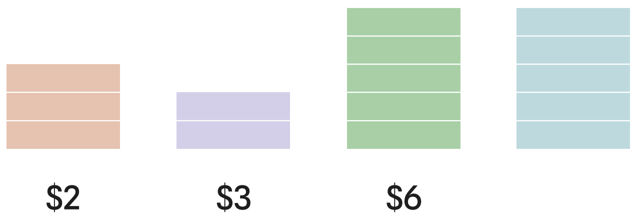
Input: list of digits.
Output: the largest number that can be made of the digits.

Sample

Input: 3, 7, 5
Output: 735

> The solution is to order the digits from the biggestone to the smallest one.

Robber's problem (aka knapsack problem)



Robber's problem

Input: a list of items with weights (kg) and costs ($) as well as the capacity of a bag (kg).

Output the maximum total cost of items that fit in the bag.

Robber’s problem: tempting approach

- It’s natural to process items in order of decreasing $ per kg.

СПбГУ

Robber's problem: tempting approach

- It's natural to process items in order of decreasing \$ per kg.
- Let's calculate utility $\frac{\text{cost}}{\text{weight}}$ for each item.

Robber’s problem: tempting approach

- It’s natural to process items in order of decreasing \$ per kg.
- Let’s calculate utility $\frac{\text{cost}}{\text{weight}}$ for each item.
- The better the utility the better the item.

СПбГУ

Robber's problem: tempting approach

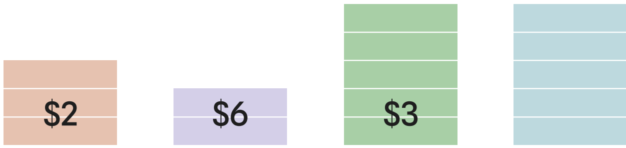
- It's natural to process items in order of decreasing \$ per kg.
- Let's calculate utility $\frac{\text{cost}}{\text{weight}}$ for each item.
- The better the utility the better the item.
- Therefore we should try to put items with maximum utility first.

СПбГУ

Robber's problem: tempting approach

- It's natural to process items in order of decreasing \$ per kg.
- Let's calculate utility $\frac{\text{cost}}{\text{weight}}$ for each item.
- The better the utility the better the item.
- Therefore we should try to put items with maximum utility first.
- Nice and easy. But, unfortunately, **wrong.**

Robber's problem: example



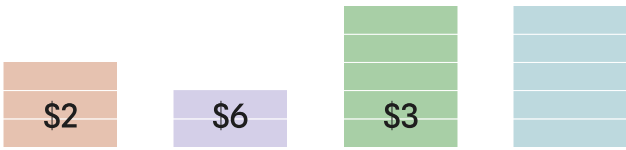
3 kg	2 kg	5 kg	The total cost is $0
$2	$3	$6	
$\frac{2}{3}$	$\frac{3}{2}$	$\frac{6}{5}$	

СПбГУ

Robber’s problem: example

СПбГУ

The Best



3 kg	2 kg	5 kg	The total cost is \$0
\$2	\$3	\$6	
$\frac{2}{3}$	$\frac{3}{2}$	$\frac{6}{5}$	

Robber's problem: example

3 kg	5 kg	The total cost is $0
$2	$6	
$\frac{2}{3}$	$\frac{6}{5}$	

Robber’s problem: example

The Best

3 kg	5 kg	The total cost is $0
$2	$6	
$\frac{2}{3}$	$\frac{6}{5}$	

Robber's problem: example

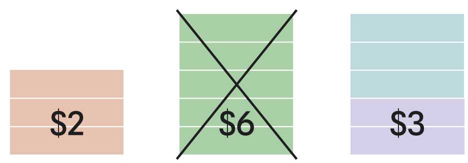
The Best

3 kg	5 kg	The total cost is \$0
\$2	\$6	
$\frac{2}{3}$	$\frac{6}{5}$	

But the third item doesn't fit to the knapsack.

Robber’s problem: example

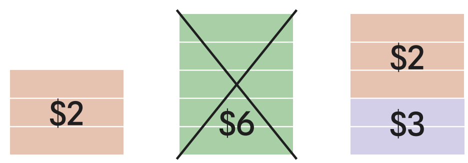
The Best



3 kg	5 kg	The total cost is $0
$2	$6	
$\frac{2}{3}$	$\frac{6}{5}$	

Robber's problem: example

The Best



3 kg	5 kg	The total cost is $0
$2	$6	
$\frac{2}{3}$	$\frac{6}{5}$	

СПбГУ

Robber's problem: example

- We got total cost \$5.

Robber’s problem: example

- We got total cost $5.
- But we could do better with the third item only:

СПбГУ

How to verify a solution?

- Thus, the initial intuitive idea turned out to be wrong.

How to verify a solution?

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?

How to verify a solution?

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.

How to verify a solution?

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.
- But what to do if your solution got “wrong answer” verdict from the grader?

How to verify a solution?

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.
- But what to do if your solution got “wrong answer” verdict from the grader?
- It’d be good to have a solution which is **always** conceptually correct.

How to verify a solution?

- Thus, the initial intuitive idea turned out to be wrong.
- Then how to convince yourself that your approach is correct?
- The simplest thing to do is to check your algorithm with pen and paper against sample tests.
- But what to do if your solution got “wrong answer” verdict from the grader?
- It’d be good to have a solution which is **always** conceptually correct.
- And that’s what we’ll do!

СПбГУ

Outline

- Intuitive solutions
- **Search space**
- Backtracking

Introduction

- In this video we will develop very powerful technique to solve problems, which works in almost all cases.

Introduction

- In this video we will develop very powerful technique to solve problems, which works in almost all cases.
- The approach yields slow solutions but it’s **conceptually correct by definition.**

Introduction

- In this video we will develop very powerful technique to solve problems, which works in almost all cases.
- The approach yields slow solutions but it's **conceptually correct by definition.**
- Therefore it could be used to verify correctness of faster solutions for the same problem.

СПбГУ

Search space

Almost every combinatorial problem falls in one of the following categories.

1. Find an element of a set A satisfying some property (or a number of such elements).

Search space

Almost every combinatorial problem falls in one of the following categories.

1. Find an element of a set *A* satisfying some property (or a number of such elements).
2. Find an element of a set *A* such that some objective function is minimized/maximized.

Search space

Almost every combinatorial problem falls in one of the following categories.

1. Find an element of a set A satisfying some property (or a number of such elements).
2. Find an element of a set A such that some objective function is minimized/maximized.

We will call the set A **search space.**

Superstring

Superstring

Input: m strings $s_1, \ldots, s_m$ consisting of letters “a” and “b” only and an integer n.

Output: a string s of length n containing each s_i (for all $1 \le i \le m$) as a substring.

Superstring

Superstring

Input: m strings $s_1, \ldots, s_m$ consisting of letters “a” and “b” only and an integer n.

Output: a string s of length n containing each s_i (for all $1 \le i \le m$) as a substring.

Sample

Input: $m = 2$; $n = 3$; $s_1 = \text{ab}$, $s_2 = \text{ba}$

Output: aba (**ab**a, a**ba**)
(another valid output is bab).

СПбГУ

Superstring: solution

- One way to solve a problem is to simply go through all possible candidate solutions. For the superstring problem, the search space consists of all strings of length n over the alphabet $\{a; b\}$.
  For each such string, we check whether it is indeed a superstring of $s_1, \ldots, s_m$

СПбГУ

Superstring: solution

- One way to solve a problem is to simply go through all possible candidate solutions. For the superstring problem, the search space consists of all strings of length n over the alphabet $\{a; b\}$.
  For each such string, we check whether it is indeed a superstring of $s_1, \ldots, s_m$
- Let's consider another testcase: $n = 4$, $s_1 = \text{bab}$, $s_2 = \text{abb}$.

СПбГУ

Superstring: solution

- One way to solve a problem is to simply go through all possible candidate solutions. For the superstring problem, the search space consists of all strings of length n over the alphabet $\{a; b\}$.
  For each such string, we check whether it is indeed a superstring of $s_1, \ldots, s_m$
- Let's consider another testcase: $n = 4$, s_1 = bab, s_2 = abb.
- There are only $2^4 = 16$ strings of four letters "a" and "b".

Superstring: search space

Candidate	bab	abb	Candidate	bab	abb
aaaa	×	×	baaa	×	×
aaab	×	×	baab	×	×
aaba	×	×	baba	**bab**a	×
aabb	×	a**abb**	**babb**	**bab**b	b**abb**
abaa	×	×	bbaa	×	×
abab	a**bab**	×	bbab	b**bab**	×
abba	×	×	abba	×	×
abbb	×	**abb**b	bbbb	×	×

Maximum subarray problem

Maximum subarray problem

Input: an array $a_1, \ldots, a_n$.

Output the largest possible sum $a_l + a_{l+1} + \ldots + a_{r-1} + a_r$ for $1 \leq l \leq r \leq n$.
Note that a_i could be negative.

Sample

Input: $a = (4, 1, -2, 3, -10, 5)$

Output: the best subarray is $(\underbrace{4, 1, -2, 3}, -10, 5$ and the sum is $4 + 1 + (-2) + 3 = 6$.

Maximum subarray problem: search space

- Search space for the maximum subarray problem is the set of all subarrays of the array a.

Maximum subarray problem: search space

- Search space for the maximum subarray problem is the set of all subarrays of the array a.
- Subarray is determined by its first and last elements: l and r.

Maximum subarray problem: search space

- Search space for the maximum subarray problem is the set of all subarrays of the array a.
- Subarray is determined by its first and last elements: l and r.
- For this problem understanding what the search space is instantly provides us with the solution.

СПбГУ

Maximum subarray problem: search space

- Search space for the maximum subarray problem is the set of all subarrays of the array a.
- Subarray is determined by its first and last elements: l and r.
- For this problem understanding what the search space is instantly provides us with the solution.
- Enumerate all pairs (l, r) such that $l \leq r$, for each pair compute the sum $a_l + \ldots + a_r$, and take the maximum.

Robber's problem

Robber's problem

Input: n items with given weights $w_1, \ldots, w_n$ and costs $c_1, \ldots, c_n$.

Output the largest total cost of the set of items whose total weight does not exceed W.

Robber's problem

Robber's problem

Input: n items with given weights $w_1, \dots, w_n$ and costs $c_1, \dots, c_n$.

Output the largest total cost of the set of items whose total weight does not exceed W.

Sample

Input: $W = 5$; $n = 3$
$w_1 = 3 \ \ w_2 = 2 \ \ w_3 = 5$
$c_1 = 2 \ \ c_2 = 3 \ \ c_3 = 6$

Output: The best solution is to put the last item to the knapsack and get the total cost 6.

Robber's problem: search space

- What is the search space for the robber's problem?

Robber’s problem: search space

- What is the search space for the robber’s problem?
- It’s all sets of items.

Robber's problem: search space

- What is the search space for the robber's problem?
- It's all sets of items.
- For the given example, possible sets are the following: $\varnothing$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1; 2\}$, $\{1; 3\}$, $\{2; 3\}$, $\{1; 2; 3\}$.

Robber's problem: search space

- What is the search space for the robber's problem?
- It's all sets of items.
- For the given example, possible sets are the following: $\varnothing$, $\{1\}$, $\{2\}$, $\{3\}$, $\{1;2\}$, $\{1;3\}$, $\{2;3\}$, $\{1;2;3\}$.
- Not all of these sets fit into the backpack, but it's easy to check: compute the total weight of the set and check whether it exceeds the capacity of the backpack.

Robber's problem: search space

- What is the search space for the robber's problem?
- It's all sets of items.
- For the given example, possible sets are the following: $\varnothing$, {1}, {2}, {3}, {1; 2}, {1; 3}, {2; 3}, {1; 2; 3}.
- Not all of these sets fit into the backpack, but it's easy to check: compute the total weight of the set and check whether it exceeds the capacity of the backpack.

Search space: summary

Problem	Problem Search space
Superstring	strings consisting of letters “a”and “b”

СПбГУ

Search space: summary

Problem	Problem Search space
Superstring	strings consisting of letters “a”and “b”
Robber’s problem	all possible sets of items

Search space: summary

Problem	Problem Search space
Superstring	strings consisting of letters “a”and “b”
Robber’s problem	all possible sets of items
Maximum subarray	pairs (l, r) such that $l \le r$

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.
- We can try all possible pairs with two nested for cycles.

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.
- We can try all possible pairs with two nested for cycles.
- For the substring problem we want to try all possible strings of n symbols.

СПбГУ

Exploring the search space

- For the maximum subarray problem the search space gives us the solution instantly.
- We can try all possible pairs with two nested for cycles.
- For the substring problem we want to try all possible strings of n symbols.
- It’d be good to have n nested for cycles iterating through letters “a” and “b”.

Implementation

Your pseudo-Python code will look like this for the superstring problem:

```
for x[0] in ['a', 'b']:
    for x[1] in ['a', 'b']:
        # . . .
        for x[n−1] in ['a', 'b']:
            # check if x contains
            # all strings
            # s[1], . . .s[m]
```

СПбГУ

Outline

- Intuitive solutions
- Search space
- **Backtracking**

СПбГУ

Introduction

- In this video we will finally understand how to write basic solution for combinatorial problems with backtracking.

Introduction

- In this video we will finally understand how to write basic solution for combinatorial problems with backtracking.
- Backtracking is roughly the way how to write n nested for cycles.

Recursion

Enumerating all strings x over *{a, b}* of length *n*:

```
for x[0] in ['a', 'b']:
    for x[1] in ['a', 'b']:
        # ...
        for x[n-1] in ['a', 'b']:
            # do some thing with x
```

Recursion

Enumerating all strings x over *{a, b}* of length *n*:

```
for x[0] in ['a', 'b']:
    for x[1] in ['a', 'b']:
        # . . .
        for x[n-1] in ['a', 'b']:
            # do some thing with x
```

The simplest possible way to simulate this “code” with an actual code is via recursion.

Recursion

The key idea is to look at n nested for cycles like this:

```
for x[0] in ['a', 'b']:
    # remaining n−1 for cycles
```

Recursion

The key idea is to look at n nested for cycles like this:

```
for x[0] in ['a', 'b']:
    # remaining n−1 for cycles
```

So we can implement the function recursively.

Three for cycles

Let's first do it for three nested for cycles.

```
def threeFors (n, x):
    for x [0] in ['a', 'b']:
        twoFors (n, x)
```

Three for cycles

Let’s first do it for three nested for cycles.

```
def threeFors (n, x):
    for x [0] in ['a', 'b']:
        twoFors (n, x)
def twoFors (n, x):
    for x [1] in ['a', 'b']:
        oneFors (n, x)
```

СПбГУ

Three for cycles

Let’s first do it for three nested for cycles.

```
def threeFors (n, x):
    for x [0] in ['a', 'b']:
        twoFors (n, x)
def twoFors (n, x):
    for x [1] in ['a', 'b']:
        oneFors (n, x)
def oneFors (n, x):
    for x [2] in ['a', 'b']:
        print (n, x)
```

Three for cycles

Let's first do it for three nested for cycles.

```
def threeFors (n, x):
    for x [0] in ['a', 'b']:
        twoFors (n, x)
def twoFors (n, x):
    for x [1] in ['a', 'b']:
        oneFors (n, x)
def oneFors (n, x):
    for x [2] in ['a', 'b']:
        print (n, x)
```

similar

Three for cycles

Let’s first do it for three nested for cycles.

```
def threeFors (n, x):
    for x [0] in ['a', 'b']:
        twoFors (n, x)
def twoFors (n, x):
    for x [1] in ['a', 'b']:
        oneFors (n, x)
def oneFors (n, x):
    for x [2] in ['a', 'b']:
        print (n, x)
```

similar

Outer call: threeFors (n, ['']*3).

Recursion

We will write the function nestedFors with additional parameter firstFor and it’ll behave like:

firstFor	Behaviour
0	threeFors
1	twoFors
2	oneFor
3	print (x)

Recursion

We will write the function nestedFors with additional parameter firstFor and it'll behave like:

firstFor	Behaviour
0	threeFors
1	twoFors
2	oneFor
3	print (x)

```
def nestedFors (n, firstFor, x):
    if firstFor < n:
        for x [firstFor] in ['a', 'b']:
            nestedFors (n, firstFor + 1, x)
    else:
        print (x)
```

Recursion

We will write the function nestedFors with additional parameter firstFor and it'll behave like:

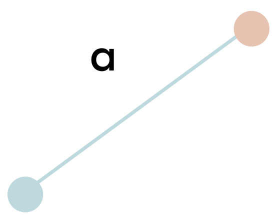
firstFor	Behaviour
0	threeFors
1	twoFors
2	oneFor
3	print (x)

```
def nestedFors (n, firstFor, x):
    if firstFor < n:
        for x [firstFor] in ['a', 'b']:
            nestedFors (n, firstFor + 1, x)
    else:
        print (x)
```

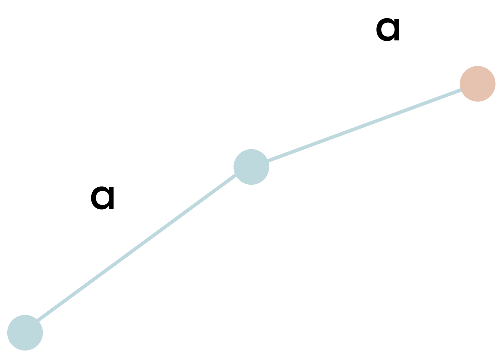
This function simulates for cycles with numbers firstFor, firstFor + 1, ..., n − 1.

Recursion: visualization

Recursion: visualization

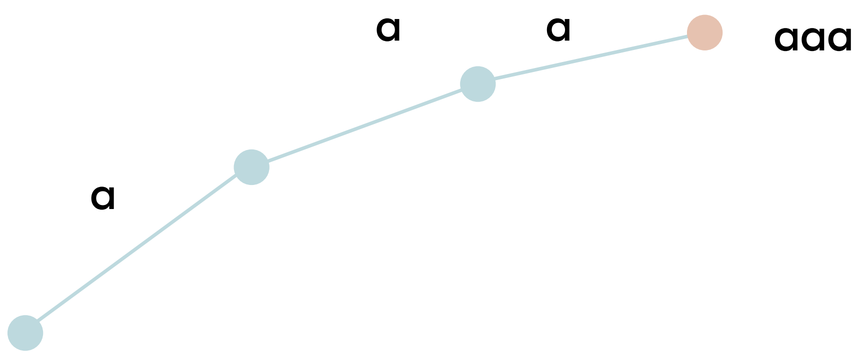


Recursion: visualization

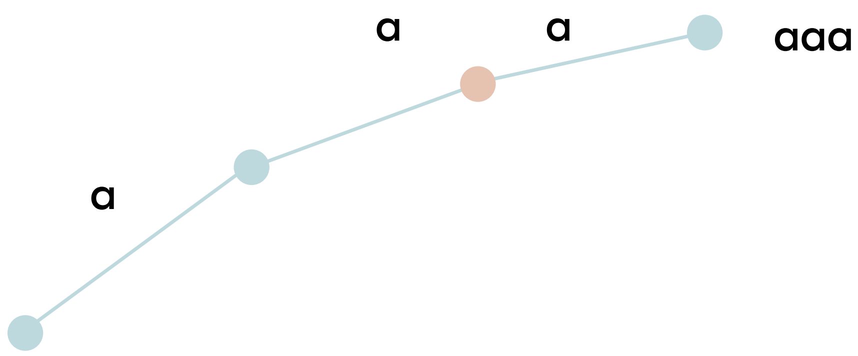


СПбГУ

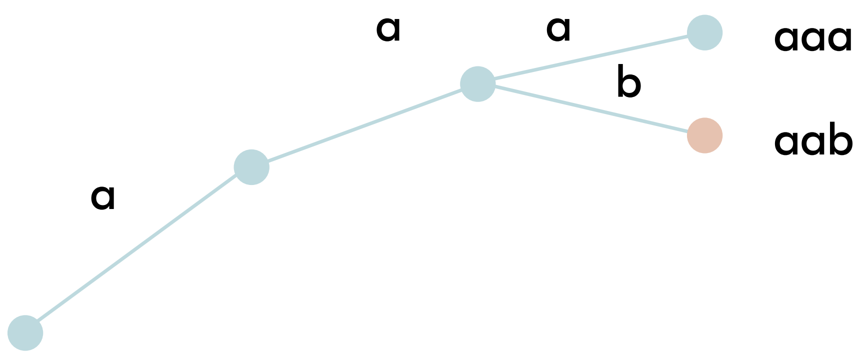
Recursion: visualization



Recursion: visualization

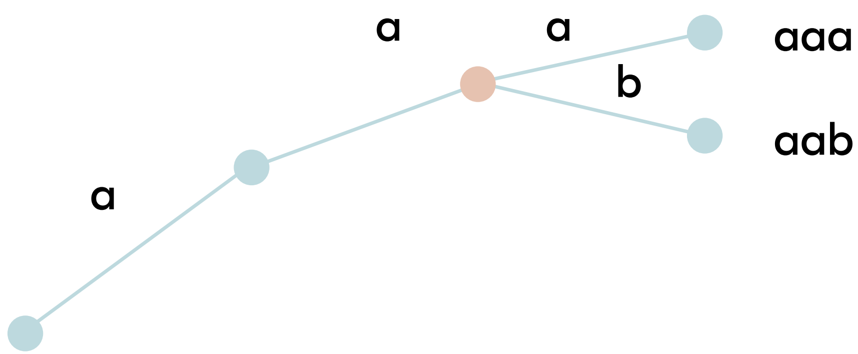


Recursion: visualization



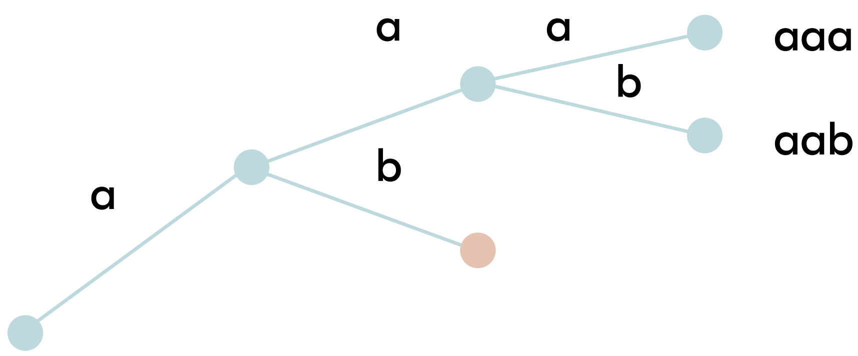
СПбГУ

Recursion: visualization

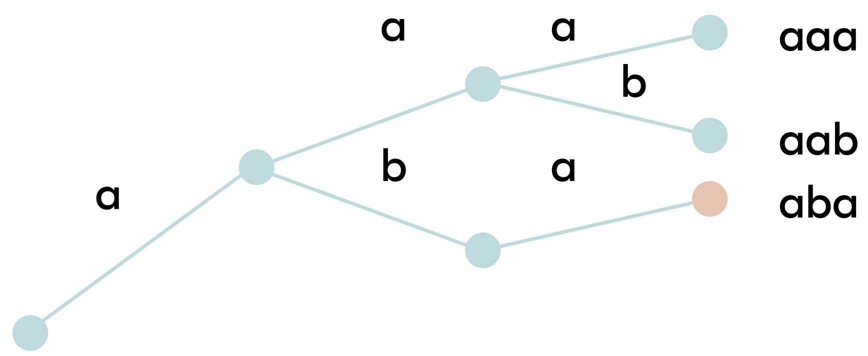


СПбГУ

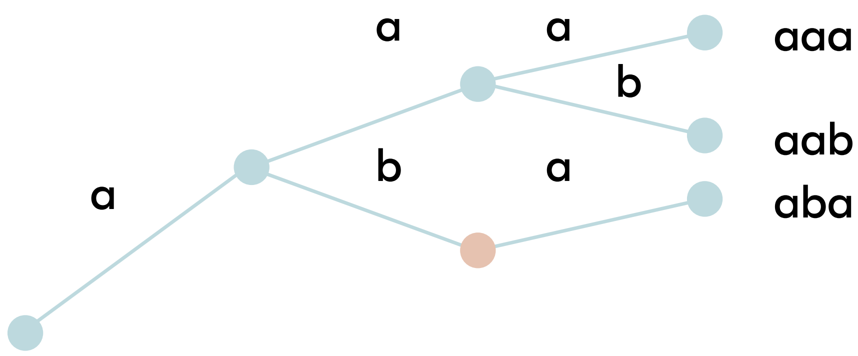
Recursion: visualization



Recursion: visualization

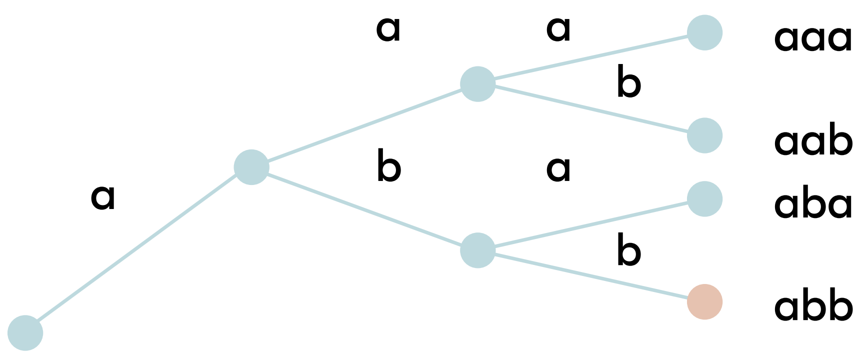


Recursion: visualization

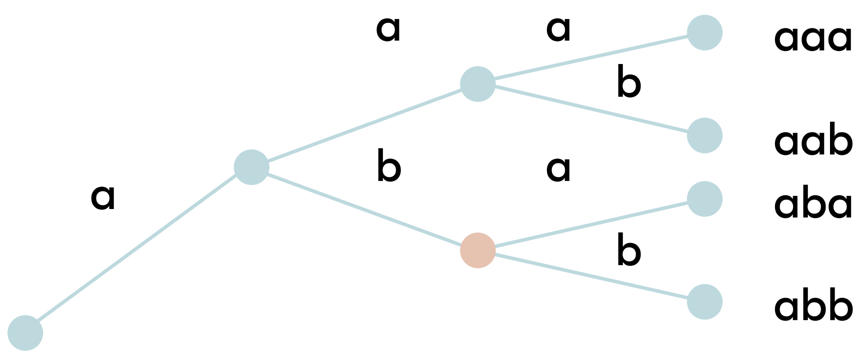


СПбГУ

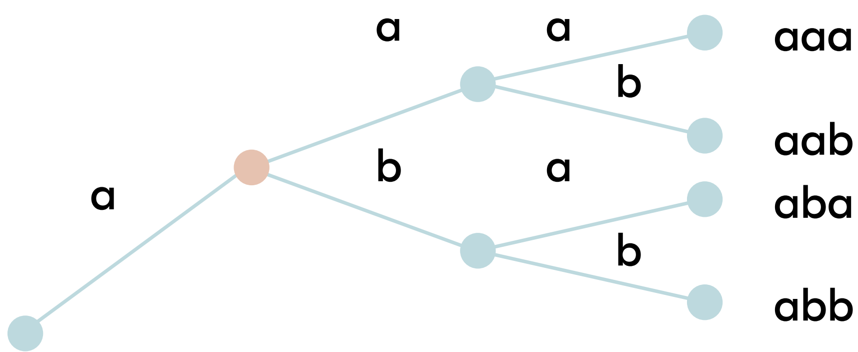
Recursion: visualization



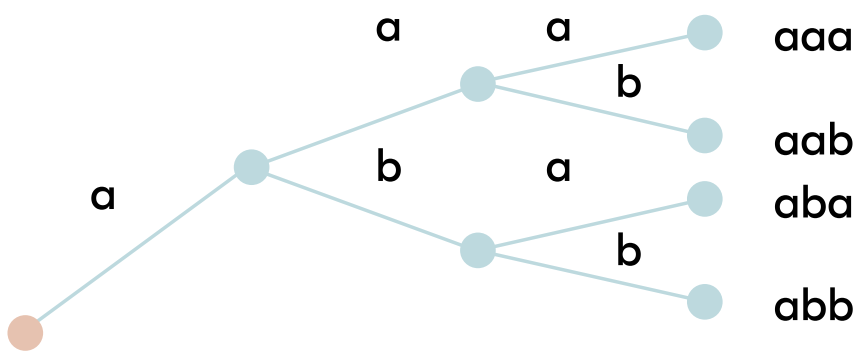
Recursion: visualization



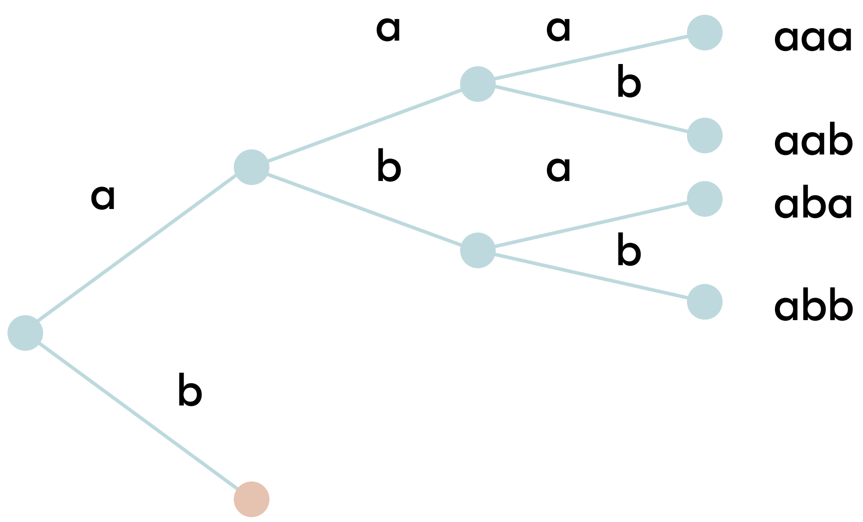
Recursion: visualization



Recursion: visualization

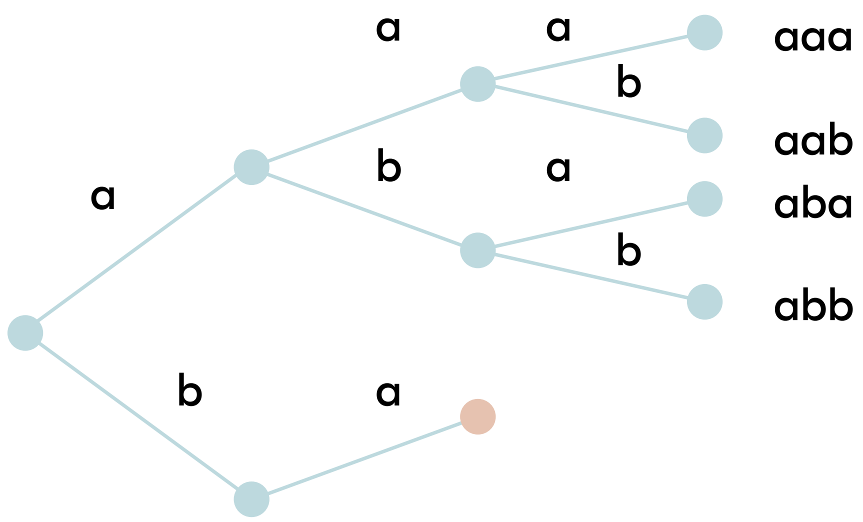


Recursion: visualization

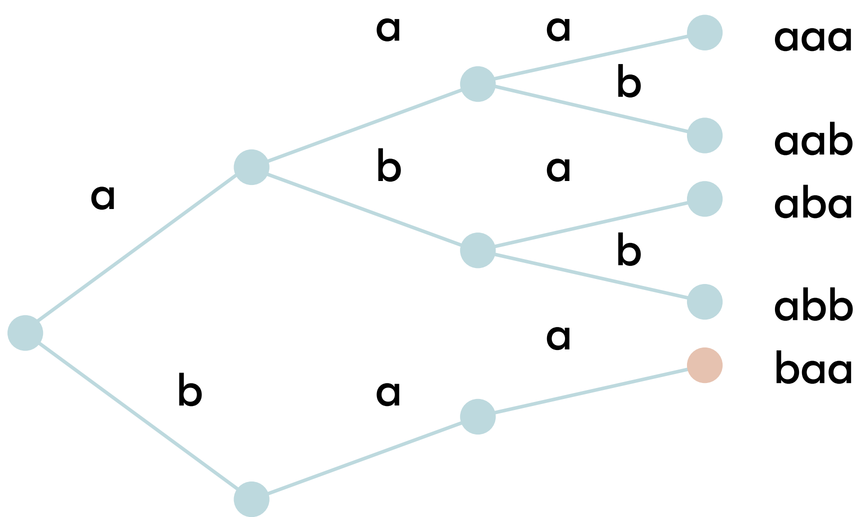


СПбГУ

Recursion: visualization

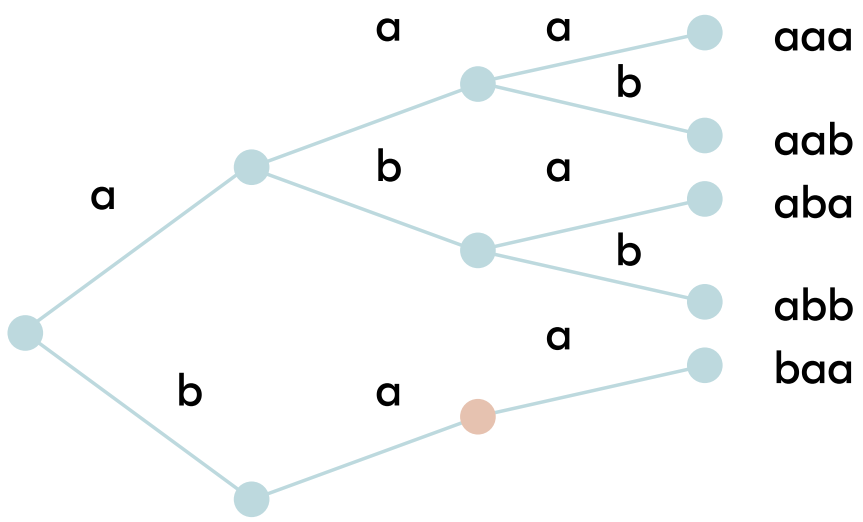


Recursion: visualization



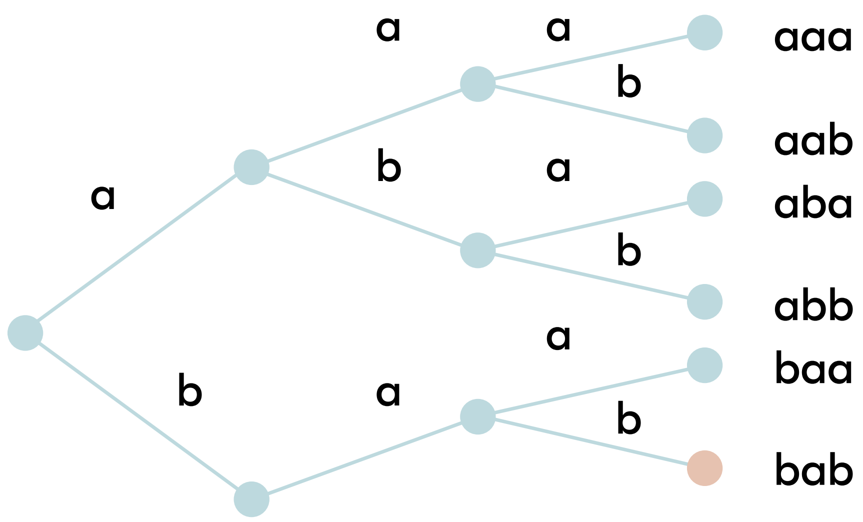
СПбГУ

Recursion: visualization

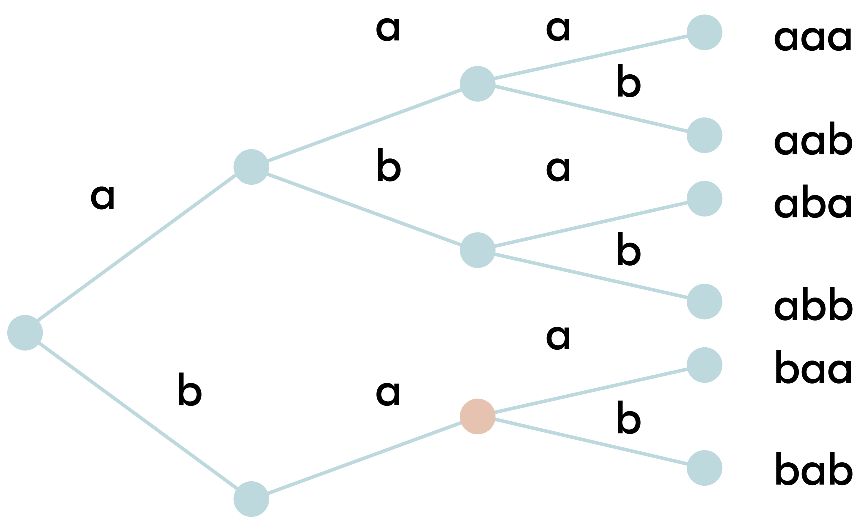


СПбГУ

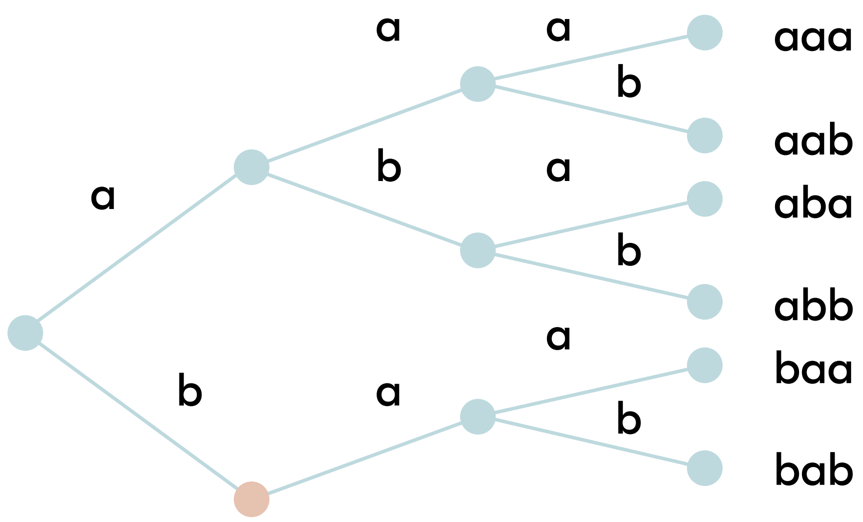
Recursion: visualization



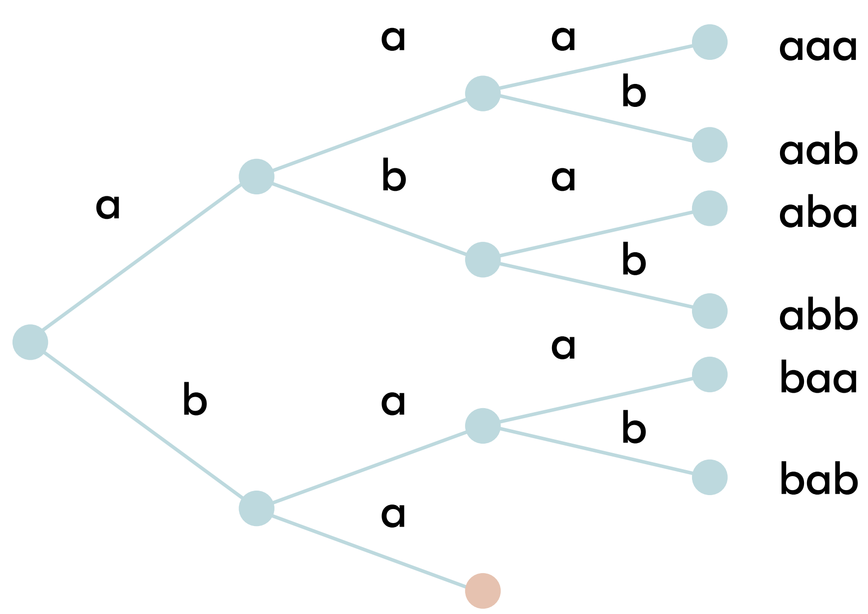
Recursion: visualization



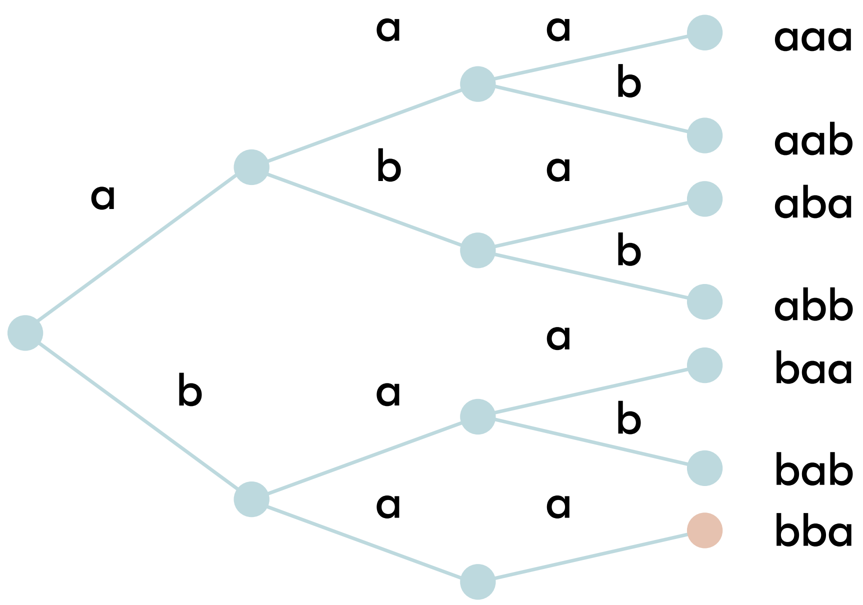
Recursion: visualization



Recursion: visualization

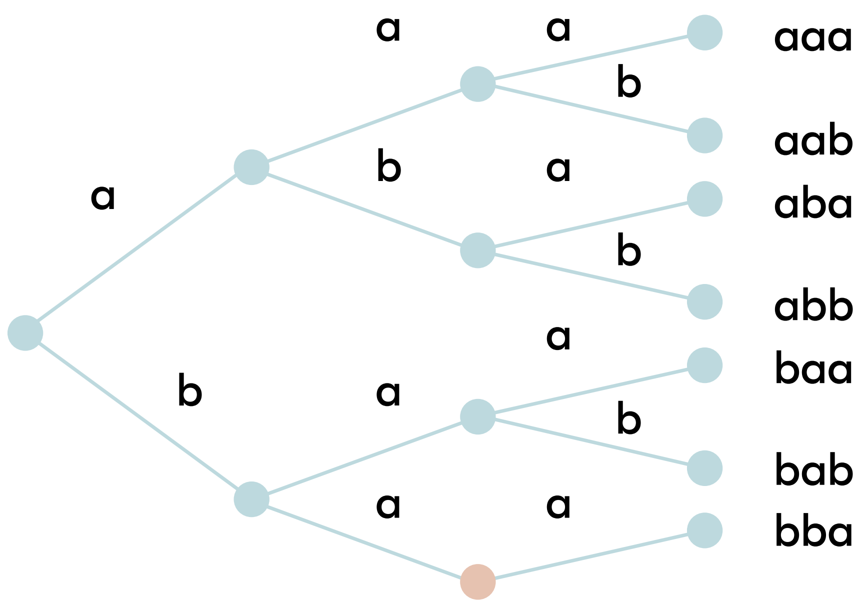


Recursion: visualization

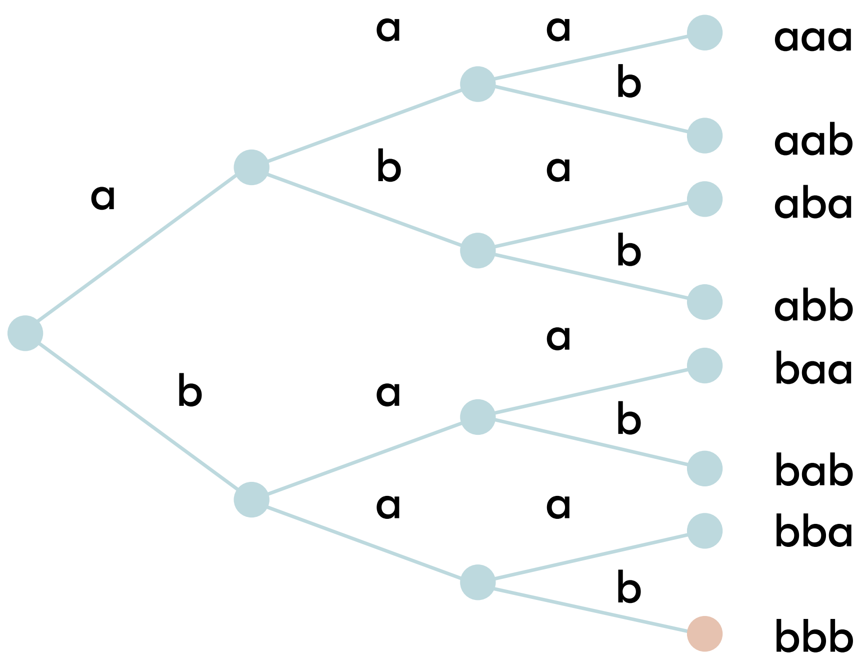


СПбГУ

Recursion: visualization

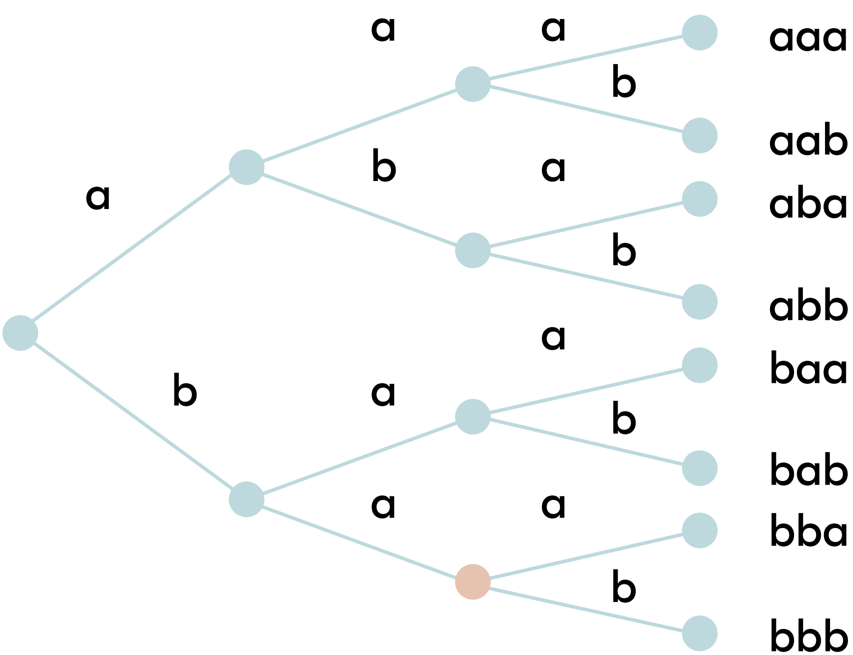


Recursion: visualization



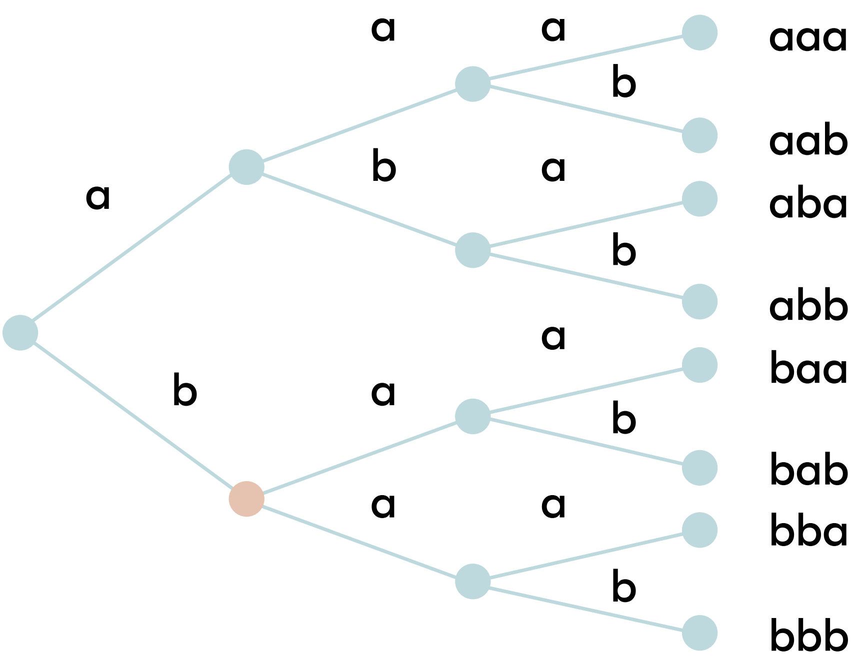
СПбГУ

Recursion: visualization

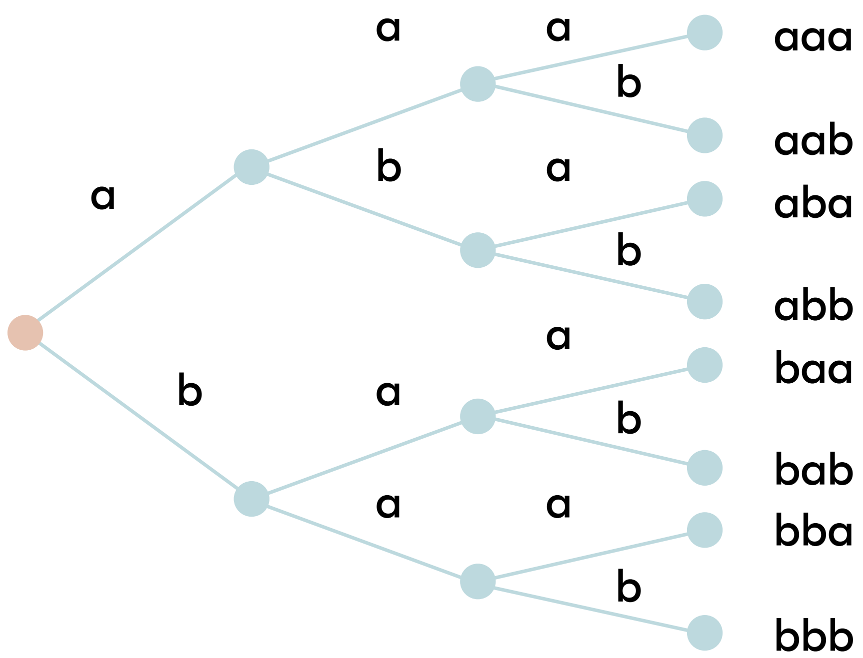


СПбГУ

Recursion: visualization



Recursion: visualization



Robber’s problem

- Search space for the robber’s problem is **the set of all sets of items.**

СПбГУ

Robber's problem

- Search space for the robber's problem is **the set of all sets of items.**
- How to enumerate all sets of n items?

Robber's problem

- Search space for the robber's problem is **the set of all sets of items.**
- How to enumerate all sets of n items?
- Basically, it is the same as enumerating all strings over {0, 1} of length *n*!

Set to string

firstFor	Items 1 2 3
∅	0 0 0
{1}	1 0 0
{2}	0 1 0
{1, 2}	1 1 0
{3}	0 0 1
{1, 3}	1 0 1
{2, 3}	0 1 1
{1, 2, 3}	1 1 1

Robber’s problem: search space

Items 1 2 3	Set	Weight	Cost
0 0 0	∅	0	0
1 0 0	{1}	3	2
0 1 0	{2}	2	3
1 1 0	{1, 2}	2 + 3 = 5	2 + 3 = 5
0 0 1	{3}	5	6
1 0 1	{1, 3}	3 + 5 = 8	2 + 6 = 8
0 1 1	{2, 3}	2 + 5 = 7	3 + 6 = 8
1 1 1	{1, 2, 3}	2 + 3 + 5 = 10	3 + 2 + 6 = 11

Recall our example: $n = 3$; $W = 5$ and

$w_1 = 3 \quad w_2 = 2 \quad w_3 = 5$

$c_1 = 2 \quad c_2 = 3 \quad c_3 = 6$

Robber's problem: solution

Therefore we reduced robber's problem to the same n nested for cycles!

Search space

Designing a brute force solution:

Search space

Designing a brute force solution:

1. Identify the search space.

Search space

Designing a brute force solution:

1. Identify the search space.
2. Design a way of enumerating all its elements.

Search space

Designing a brute force solution:

1. Identify the search space.
2. Design a way of enumerating all its elements.
3. Turn it into a solution.

Search space

Designing a brute force solution:

1. Identify the search space.
2. Design a way of enumerating all its elements.
3. Turn it into a solution.

> The resulting solution is usually slow,
> but it is clearlycorrect and can be used for debugging.