Saint Petersburg
State
University

Worst cases

Kirill Simonov

Leading Researcher

Introduction

- Time limits are tight.

Introduction

- Time limits are tight.
- The whole point — to do the thing quickly.

Introduction

- Time limits are tight.
- The whole point — to do the thing quickly.
- Measure quickness.

Introduction

- Time limits are tight.
- The whole point — to do the thing quickly.
- Measure quickness.
- Predict before implementing.

Introduction

- Time limits are tight.
- The whole point — to do the thing quickly.
- Measure quickness.
- Predict before implementing.
- Make solutions faster.

Example

Substring problem

Given two strings s and t check if s is a substring of t.

Example

Substring problem

Given two strings *s* and *t* check if *s* is a substring of *t*.

Input: *s* = abac; *t* = abacabad
Output: Yes: abac**abad**

Example

Substring problem

Given two strings *s* and *t* check if *s* is a substring of *t*.

Input: *s* = abac; *t* = abacabad
Output: Yes: abac**abad**

Input: *s* = cac; *t* = abacabad
Output: No

Example

Substring problem

Given two strings s and t check if s is a substring of t.

Input: s = abac; t = abacabad
Output: Yes: abac**abad**

Input: s = cac; t = abacabad
Output: No

Input: s = abab; t = abacabab
Output: Yes: abac**abab**

Algorithm

$n := \mathbf{length}(s)$

$m := \mathbf{length}(t)$

For all substrings of ***t*** of length ***n***:

- Compare characters of ***s*** and this substring one by one.
- If there is a mismatch, move on to the next substring.
- If all characters are equal, return Yes.
- If none of substrings matches, return No.

Examples

s = **abac**; t = **abacabad**;

Examples

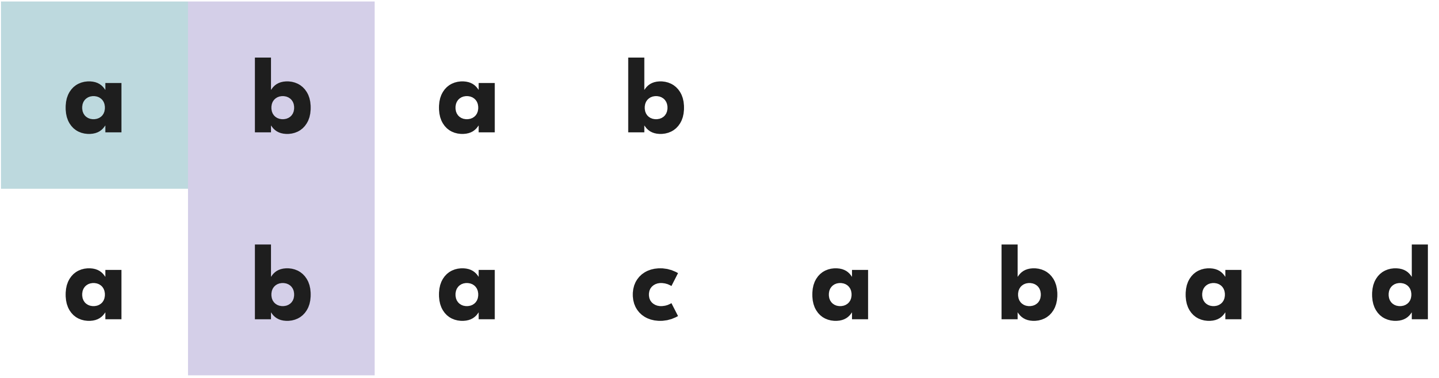
s = **abac;** *t* = **abacabad;**

a	**b**	**a**	**b**				
a	**b**	**a**	**c**	**a**	**b**	**a**	**d**

0 operations

Examples

s = **abac;** *t* = **abacabad;**

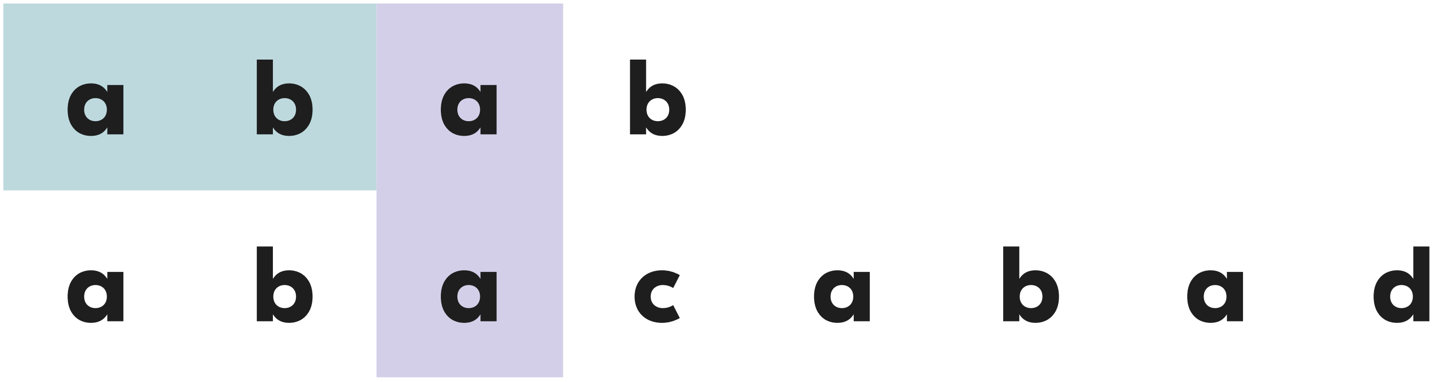


a	**b**	**a**	**b**				
a	**b**	**a**	**c**	**a**	**b**	**a**	**d**

1 operations

Examples

s **= abac;** ***t*** **= abacabad;**



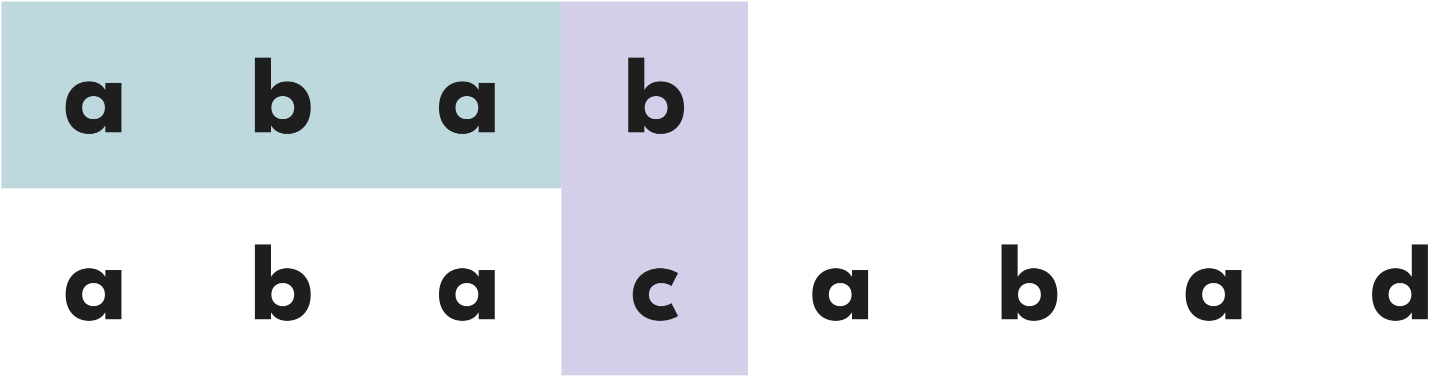
a b a b

a b a c a b a d

2 operations

Examples

s = **abac**; *t* = **abacabad**;

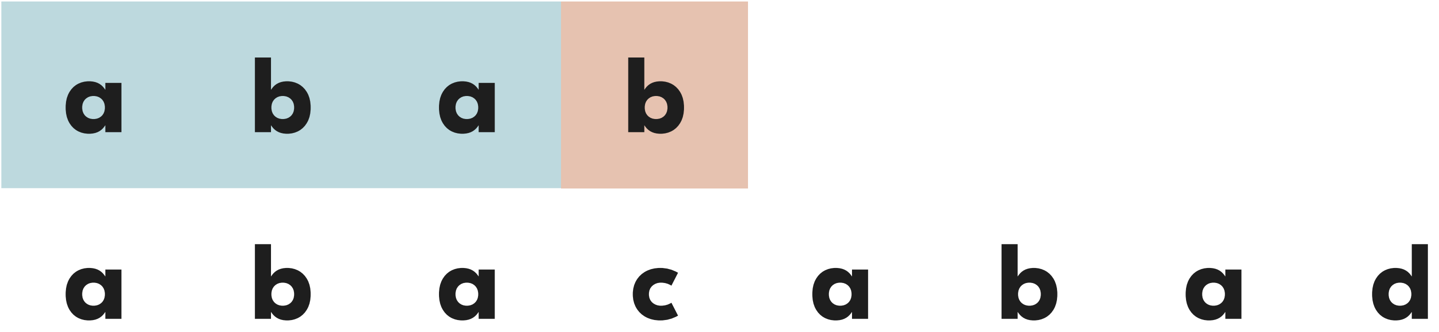


a	**b**	**a**	**b**				
a	**b**	**a**	**c**	**a**	**b**	**a**	**d**

3 operations

Examples

s = abac; *t* = abacabad;



a b a b

a b a c a b a d

4 = 4

Examples

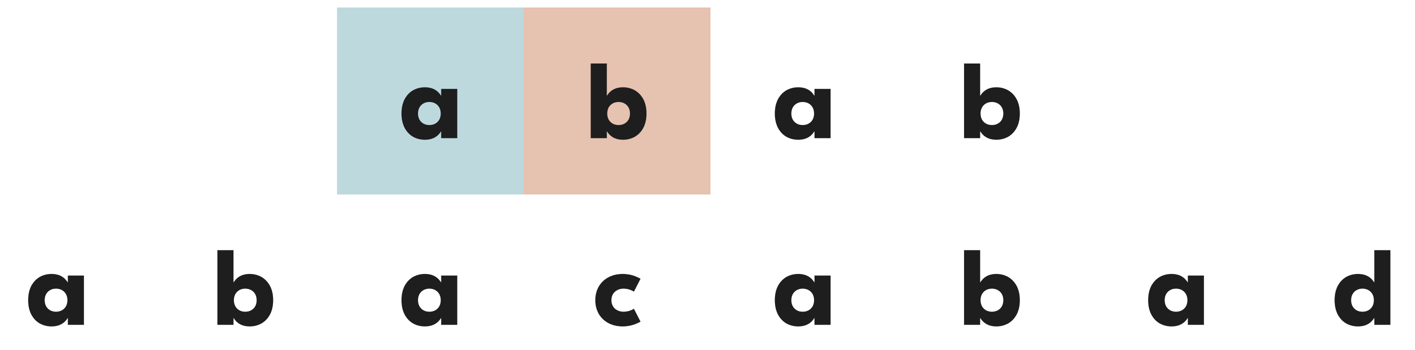
s = abac; *t* = abacabad;

	a	b	a	b			
a	b	a	c	a	b	a	d

4 + 1 = 5

Examples

s = abac; *t* = abacabad;



		a	b	a	b		
a	b	a	c	a	b	a	d

4 + 1 + 2 = 7

Examples

s = abac; *t* = abacabad;

			a	**b**	**a**	**b**	
a	**b**	**a**	**c**	**a**	**b**	**a**	**d**

4 + **1** + **2** + **1** = **8**

Examples

s = **abac**; *t* = **abacabad;**

				a	**b**	**a**	**b**
a	**b**	**a**	**c**	**a**	**b**	**a**	**d**

4 + 4 + 4 + 4 = 16

Examples

s = **abac**; t = **abacabad**;

Examples

s **= abac;** ***t*** **= abacabad;**

a	**b**	**a**	**c**				
a	**b**	**a**	**c**	**a**	**b**	**a**	**d**

4 = 4

We instantly got the match!

Time could vary

- The number of operations could be different.

Time could vary

- The number of operations could be different.
- If your program is fast on the samples or even on some custom tests, that doesn’t mean it’ll ***always*** be this way.

Time could vary

- The number of operations could be different.
- If your program is fast on the samples or even on some custom tests, that doesn't mean it'll ***always*** be this way.
- Your program should work quickly on the ***worst*** possible test.

Time could vary

- The number of operations could be different.
- If your program is fast on the samples or even on some custom tests, that doesn’t mean it’ll ***always*** be this way.
- Your program should work quickly on the ***worst*** possible test.
- The worst possible test for our previous algorithm:

Time could vary

- The number of operations could be different.
- If your program is fast on the samples or even on some custom tests, that doesn’t mean it’ll ***always*** be this way.
- Your program should work quickly on the ***worst*** possible test.
- The worst possible test for our previous algorithm:

— the answer is “No” — we will check every substring;

Time could vary

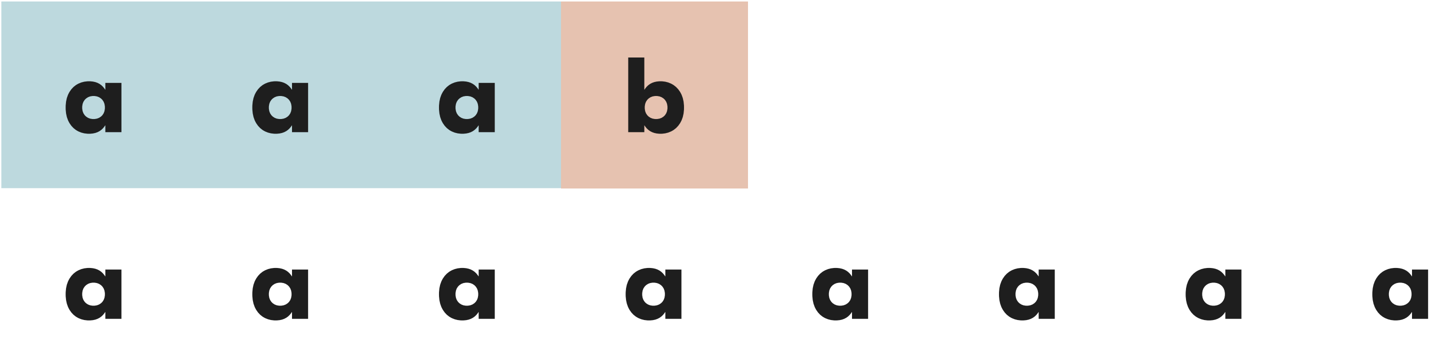
- The number of operations could be different.
- If your program is fast on the samples or even on some custom tests, that doesn't mean it'll ***always*** be this way.
- Your program should work quickly on the ***worst*** possible test.
- The worst possible test for our previous algorithm:
  - the answer is "No" — we will check every substring;
  - on every substring we will compare characters until the last.

Examples

s = **aaab**; t = **aaaaaaaa**;

Examples

s = aaab; *t* = aaaaaaaa;

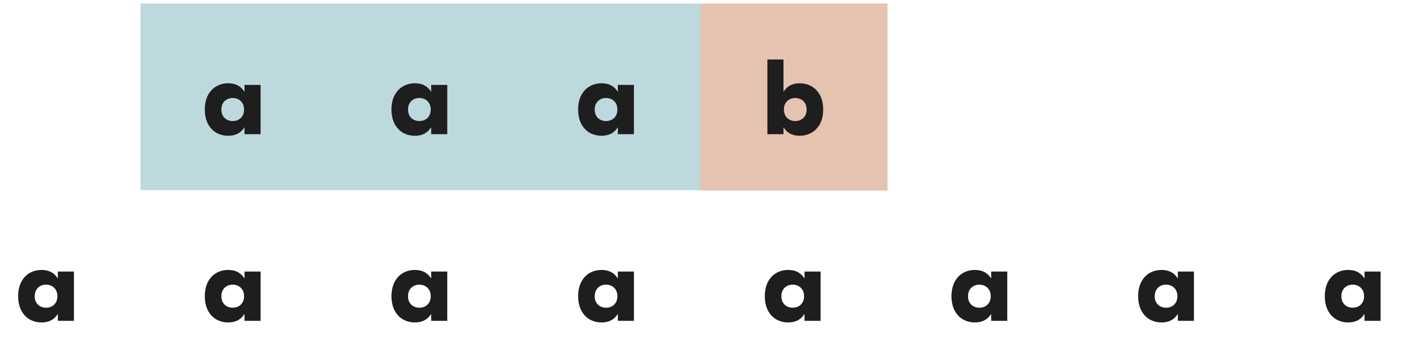


a	a	a	b				
a	a	a	a	a	a	a	a

4 = 4

Examples

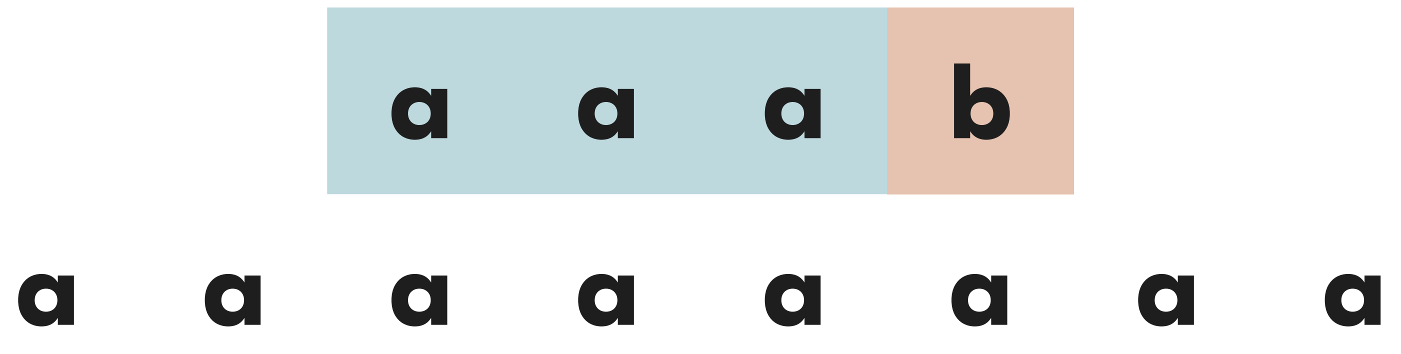
s = aaab; t = aaaaaaaa;



4 + 4 = 8

Examples

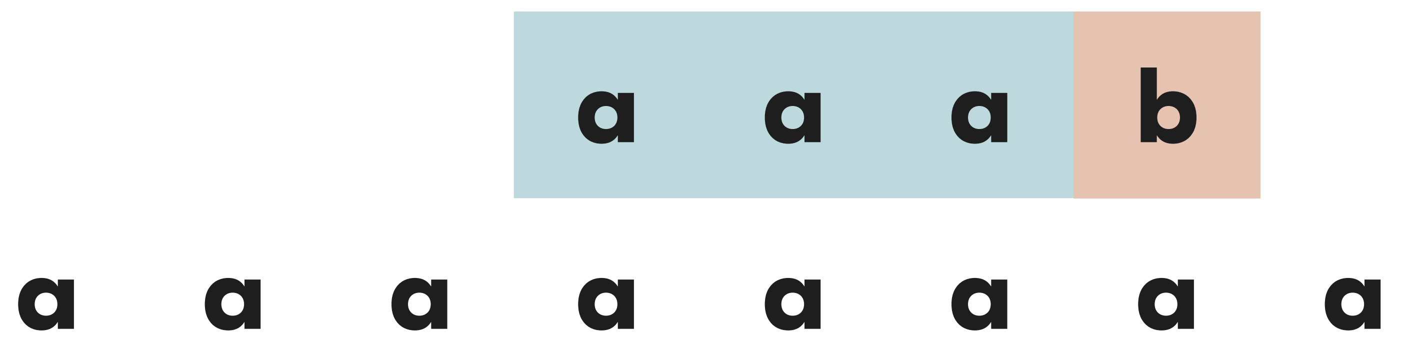
s = aaab; *t* = aaaaaaaa;



4 + 4 + 4 = 12

Examples

s = aaab; *t* = aaaaaaaa;



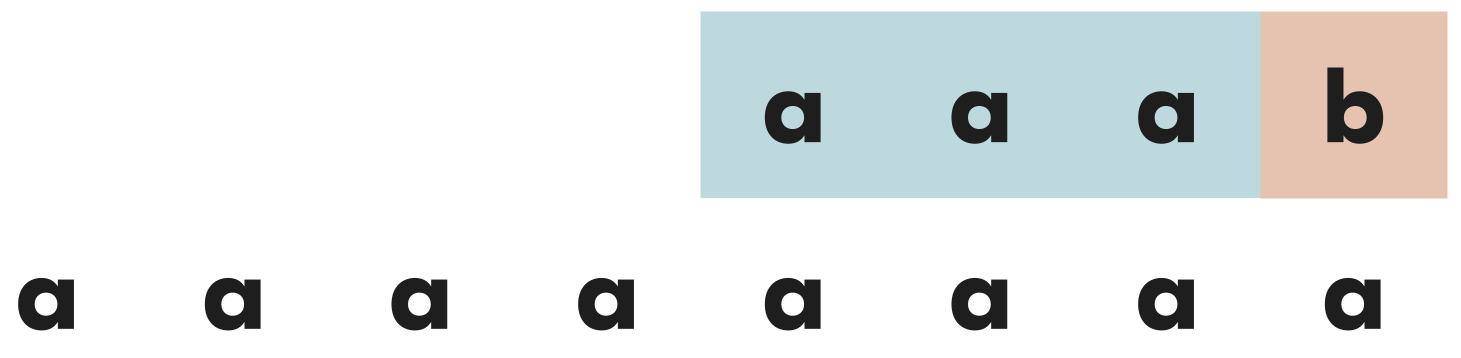
a a a b

a a a a a a a a

4 + 4 + 4 + 4 = 16

Examples

s = aaab; *t* = aaaaaaaa;



4 + 4 + 4 + 4 + 4 = 20

Conclusion

- The worst test is not just any big enough test.

Conclusion

- The worst test is not just any big enough test.
- It could be hard to construct it.

Conclusion

- The worst test is not just any big enough test.
- It could be hard to construct it.
- Goal — to estimate the number of operations on any test without finding the worst possible.

Saint Petersburg
State
University

Big-O notation

Kirill Simonov

Leading Researcher

Which operations are unit?

We’ll count operations taking some small fixed amount of time:

Which operations are unit?

We'll count operations taking some small fixed amount of time:

- number operations (+, –, *, /, %, <, >, =);
- logical operations (or, and, not, xor);
- accessing a value from an array;
- defining a new variable.

Which operations are not unit?

Some operations take more time:

СПбГУ

Which operations are not unit?

Some operations take more time:

- comparing strings or lists;
- defining a string or a list with many elements;
- concatenating two strings.

Which operations are not unit?

Some operations take more time:

- comparing strings or lists;
- defining a string or a list with many elements;
- concatenating two strings.

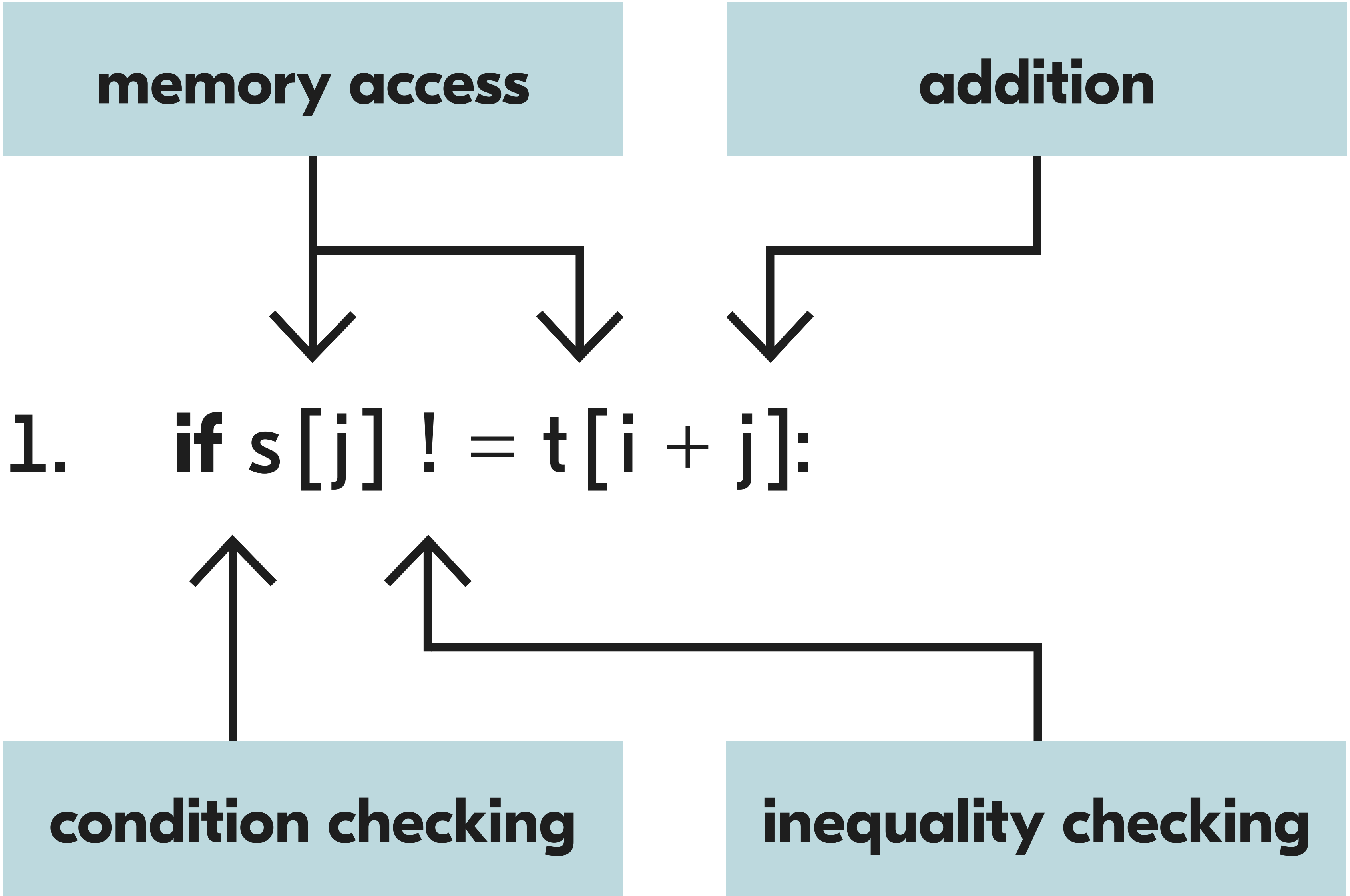
Strings and lists consist of small elements.
The operations are applied to each element.
So if there are many of them, it could take much time.

Substring problem

n = **length** (s); m = **length** (t)

```
for i in range(m − n + 1): (0, 1, … , m − n)
    match = True
    for j in range(n):
        if s[j] != t[i + j]: mismatch!
            match = False
            break already not equal
    if match:
        break
```

A condition



memory access

addition

```
1.    if s[j] != t[i + j]:
```

condition checking

inequality checking

Dropping constants

- Tedious to count all operations.

Dropping constants

- Tedious to count all operations.
- The number of operations in that line is ***independent*** of the input.

Dropping constants

- Tedious to count all operations.
- The number of operations in that line is ***independent*** of the input.
- A constant number of operations — no need to count explicitly.

Substring problem

```
for i in range (m − n + 1):            ← no more than m times
    match = True
    for j in range (n):                ← n times
             ⎧ if s [j] ! = t [i + j]:
    constant ⎨     match = False
             ⎩     break
    if match:
        break
```

Substring problem

```
for i in range (m − n + 1):            ← no more than m times
    match = True
    for j in range (n):                ← n times
           ⎧ if s [j] ! = t [i + j]:
 constant  ⎨     match = False
           ⎩     break
    if match:
        break
```

The algorithm does no more than $m \cdot n \cdot$ **constant** operations.
Without checking particular tests!

Big-O notation

- **Upper** bounds up to a ***constant*** multiplier.

Big-O notation

- **Upper** bounds up to a ***constant*** multiplier.
- Constants are similar for different solutions, but what matters is the dependence on input.

Big-O notation

- **Upper** bounds up to a ***constant*** multiplier.
- Constants are similar for different solutions,
  but what matters is the dependence on input.
- $\boldsymbol{n}$ — some input parameter, and $\boldsymbol{f(n)}$ — some function of $\boldsymbol{n}$.

Big-O notation

- **Upper** bounds up to a ***constant*** multiplier.
- Constants are similar for different solutions,
  but what matters is the dependence on input.
- n — some input parameter, and $f(n)$ — some function of n.
- An algorithm has asymptotic time complexity of $O(f(n))$
  if it does no more than $C \cdot f(n)$ operations on any input,
  where C is some constant number.

Big-O notation

- **Upper** bounds up to a ***constant*** multiplier.
- Constants are similar for different solutions, but what matters is the dependence on input.
- n — some input parameter, and $f(n)$ — some function of n.
- An algorithm has asymptotic time complexity of $O(f(n))$ if it does no more than $C \cdot f(n)$ operations on any input, where C is some constant number.
- Could be several parameters.
  Our superstring algorithm is $O(m \cdot n)$.

Properties of *O*

Properties of O

- Upper bounds: $O(...)$ if $\leq$ constant $\cdot \ldots$ operations.

Properties of O

- Upper bounds: $O(...)$ if ≤ constant · … operations.
- Could be less!
  May be $O(n^2)$, but also $O(n)$.
  If $O(n)$, then $O(n^2)$, as $n \leq n^2$.

Properties of O

- Upper bounds: $O(\ldots)$ if $\leq$ constant $\cdot \ldots$ operations.
- Could be less!
  May be $O(n^2)$, but also $O(n)$.
  If $O(n)$, then $O(n^2)$, as $n \leq n^2$.
- Optimal bounds may be very non-trivial.

Properties of O

- Upper bounds: $O(\ldots)$ if $\leq$ constant $\cdot \ldots$ operations.
- Could be less!
  May be $O(n^2)$, but also $O(n)$.
  If $O(n)$, then $O(n^2)$, as $n \leq n^2$.
- Optimal bounds may be very non-trivial.
- But we could get some simple bounds.

Single statement

- Unit operations — $O(1)$.

Single statement

- Unit operations — ***O*** **(1)**.
- Built-in functions/structures need to know in advance.
  Comparing strings — ***O*** **(size)**.
  Requires passing through elements — at least size operations.

Single statement

- Unit operations — $O(1)$.
- Built-in functions/structures need to know in advance.
  Comparing strings — $O(\text{size})$.
  Requires passing through elements — at least size operations.
- Own function — bound separately.

Recursion

1. **for** i **in** **range**(n):

$O(f(n))$

- Inside part $O(f(n))$ on each iteration.
- $O(n \cdot f(n) + n)$ in total.
- Iterating is constant $\cdot\, n$ by itself.

Recursion

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

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

- ***n*** nested for loops, each runs over 2 letters.
- So $\mathbf{2 \cdot 2 \cdots 2 = 2n}$ iterations in total print **(*x*)** outputs every element of ***x***, length is ***n***, so it's $\mathbf{O(n)}$ by itself.
- Overall, $\mathbf{O(n \cdot 2^n)}$.

Saint Petersburg
State
University

From theory to practice

Kirill Simonov
Leading Researcher

Solving a problem

1. Invent a solution.

Solving a problem

1. Invent a solution.
2. Check if it’s correct.

Solving a problem

1. Invent a solution.
2. Check if it’s correct.
3. Get $O(\ldots)$ bound — could be done without implementing!

Solving a problem

1. Invent a solution.
2. Check if it’s correct.
3. Get $O(\ldots)$ bound — could be done without implementing!
4. Check if it’s fast enough.

Solving a problem

1. Invent a solution.
2. Check if it's correct.
3. Get $O(\ldots)$ bound — could be done without implementing!
4. Check if it's fast enough.
5. If not, invent another or get a better bound.

Will it pass?

- $O(\ldots)$ operations — some function of input variables like $O(n^3)$ or $O(n \cdot m)$.

Will it pass?

- $O(\ldots)$ operations — some function of input variables like $O(n^3)$ or $O(n \cdot m)$.
- These values are bound by the statement — plug the limits in your O estimate.

Will it pass?

- $O(\ldots)$ operations — some function of input variables like $O(n^3)$ or $O(n \cdot m)$.
- These values are bound by the statement — plug the limits in your O estimate.
- $O(n^3)$, $n \le 100 : 100^3 = 10^6$.
  $O(n \cdot m)$, $n \le 10^4$, $m \le 10^6 : 10^4 \cdot 10^6 = 10^{10}$.

Will it pass?

- $O(\ldots)$ operations — some function of input variables like $O(n^3)$ or $O(n \cdot m)$.
- These values are bound by the statement — plug the limits in your O estimate.
- $O(n^3)$, $n \le 100 : 100^3 = 10^6$.
  $O(n \cdot m)$, $n \le 10^4$, $m \le 10^6 : 10^4 \cdot 10^6 = 10^{10}$.
- Compare with how many operations could be done in a second.
  Expected to be 10^8–10^9 simple operations, in C++ or Java.

Will it pass?

- $O(\ldots)$ operations — some function of input variables like $O(n^3)$ or $O(n \cdot m)$.
- These values are bound by the statement — plug the limits in your O estimate.
- $O(n^3)$, $n \le 100 : 100^3 = 10^6$.
  $O(n \cdot m)$, $n \le 10^4$, $m \le 10^6 : 10^4 \cdot 10^6 = 10^{10}$.
- Compare with how many operations could be done in a second.
  Expected to be 10^8—10^9 simple operations, in C++ or Java.
- Less for Python, about 10^7.

Will it pass?

- $O(\dots)$ operations — some function of input variables like $O(n^3)$ or $O(n \cdot m)$.
- These values are bound by the statement — plug the limits in your O estimate.
- $O(n^3)$, $n \leq 100 : 100^3 = 10^6$.
  $O(n \cdot m)$, $n \leq 10^4$, $m \leq 10^6 : 10^4 \cdot 10^6 = 10^{10}$.
- Compare with how many operations could be done in a second.
  Expected to be 10^8—10^9 simple operations, in C++ or Java.
- Less for Python, about 10^7.
- 10^6 — will pass even with quite big constant.
  10^{10} — won't pass.

Constants matter

- What if we’re somewhere in between.

СПбГУ

Constants matter

- What if we're somewhere in between.
- $O(n)$ is better than $O(n^2)$.
  And if it's $10^6 \cdot n$ vs $10 \cdot n^2$ and $n \leq 100$?

Constants matter

- What if we're somewhere in between.
- $O(n)$ is better than $O(n^2)$.
  And if it's $10^6 \cdot n$ vs $10 \cdot n^2$ and $n \le 100$?
- Multiply by large factors even when formally constants.

```
for i in range (n):
    for c in 'a' .. 'z':
        some thing in O(1)
```

- Formally $O(n)$ — second doesn't depend on input.
- But when estimating operations, use $26 \cdot n$ instead of just n.

Operations differ

Light:

- +, −
- logical
- *

Heavy:

- %
- appending to strings/lists
- recursion
- math functions like sqrt
- I/O

Considering constants

- When you’ve got the number of operations under ***O*** and still in doubt.

Considering constants

- When you’ve got the number of operations under ***O*** and still in doubt.
- Think about what constant will it be multiplied by.

Considering constants

- When you’ve got the number of operations under ***O*** and still in doubt.
- Think about what constant will it be multiplied by.
- Few light operations per one — larger bound is still fine.

Considering constants

- When you’ve got the number of operations under ***O*** and still in doubt.
- Think about what constant will it be multiplied by.
- Few light operations per one — larger bound is still fine.
- Many and/or heavy — smaller, like 10^7, could also TL.

Considering constants

- When you’ve got the number of operations under ***O*** and still in doubt.
- Think about what constant will it be multiplied by.
- Few light operations per one — larger bound is still fine.
- Many and/or heavy — smaller, like 10^7, could also TL.
- You should consider only frequent operations sqrt is heavier than + but if you have 1 of sqrt and 10^6 of +, it doesn’t matter.

Saint Petersburg
State
University

Making a solution faster

Kirill Simonov
Leading Researcher

Locally

Locally

```
for i in range(n):
    …
    for j in range(m):
        …
        doSomething()
        …
    …
```

- Overall number of operations is $O(\dots)$.

Locally

```
for i in range(n):
    …
    for j in range(m):
        …
        doSomething()
        …
    …
```

- Overall number of operations is $\boldsymbol{O}(\ldots)$.
- Our contribution: $\boldsymbol{O}(\boldsymbol{n} \cdot \boldsymbol{m} \cdot \textbf{time}(\boldsymbol{doSomething}))$.

Locally

- May be a bottleneck: if overall $\boldsymbol{O}(\boldsymbol{n}^2 \cdot \boldsymbol{m})$
  and **time**(***doSomething***) = $\boldsymbol{O}(\boldsymbol{n})$,
  it contributes $\boldsymbol{O}(\boldsymbol{n} \cdot \boldsymbol{m} \cdot \boldsymbol{n}) = \boldsymbol{O}(\boldsymbol{n}^2 \cdot \boldsymbol{m})$. So up to a constant
  this line has as much operations, as the entire program.
  If you want faster solution, you need to optimize that.

Locally

- May be a bottleneck: if overall $O(n^2 \cdot m)$ and $\mathbf{time}(doSomething) = O(n)$, it contributes $O(n \cdot m \cdot n) = O(n^2 \cdot m)$. So up to a constant this line has as much operations, as the entire program. If you want faster solution, you need to optimize that.
- Or not: if overall $O(n^3 \cdot m)$, then no sense making doSomething faster it alredy contributes only $O(n^2 \cdot m)$ — n times smaller than something else.

Making a solution faster

- Your solution is too slow.

Making a solution faster

- Your solution is too slow.
- First, try to improve asymptotically.

Making a solution faster

- Your solution is too slow.
- First, try to improve asymptotically.
- Only in bottleneck parts.

Making a solution faster

- Your solution is too slow.
- First, try to improve asymptotically.
- Only in bottleneck parts.
- If you couldn’t get better asymptotically and your solution is just above the TL, try to optimize constants, but only **then**.

Making a solution faster

- Your solution is too slow.
- First, try to improve asymptotically.
- Only in bottleneck parts.
- If you couldn’t get better asymptotically and your solution is just above the TL, try to optimize constants, but only **then**.
- Get rid of heavy operations.

Making a solution faster

- Your solution is too slow.
- First, try to improve asymptotically.
- Only in bottleneck parts.
- If you couldn’t get better asymptotically and your solution is just above the TL, try to optimize constants, but only **then**.
- Get rid of heavy operations.
- Especially of large debug output.

Making a solution faster

- Your solution is too slow.
- First, try to improve asymptotically.
- Only in bottleneck parts.
- If you couldn’t get better asymptotically and your solution is just above the TL, try to optimize constants, but only **then**.
- Get rid of heavy operations.
- Especially of large debug output.
- Do not recompute.

Measure actual time

- $O(\ldots)$ — theoretical bounds.

Measure actual time

- $O(\ldots)$ — theoretical bounds.
- If you have a program — you could measure actual time.

Measure actual time

- $O(\ldots)$ — theoretical bounds.
- If you have a program — you could measure actual time.
- Remotely submit to a testing system. Could be a remote run interface, like in Codeforces.

Measure actual time

- $O(\ldots)$ — theoretical bounds.
- If you have a program — you could measure actual time.
- Remotely submit to a testing system. Could be a remote run interface, like in Codeforces.
- Locally — need max test, could be different. But could measure different parts and do not waste attempts.

Measure actual time

- $O(\ldots)$ — theoretical bounds.
- If you have a program — you could measure actual time.
- Remotely submit to a testing system. Could be a remote run interface, like in Codeforces.
- Locally — need max test, could be different. But could measure different parts and do not waste attempts.
- How many times a function is called:

```
def someFunction():
    counter + = 1
    …
```

Measure locally

- Whole program
  time [command] — UNIX-like systems.

Measure locally

- Whole program
  time [command] — UNIX-like systems.
- See how much time has elapsed inside the program:

```
1. start = getTime( )
2. …
3. print(getTime( ) − start)
```

Could measure the whole program, or just some parts, and see how much do they actually contribute.

Measure locally

- Whole program
  time [command] — UNIX-like systems.
- See how much time has elapsed inside the program:

```
1. start = getTime( )
2. …
3. print(getTime( ) − start)
```

Could measure the whole program, or just some parts, and see how much do they actually contribute.

- Profilers measure running time and number of calls for each function. Only a structured code benefits!

Memory

- Aside from time, your program should also fit in the memory limit.

Memory

- Aside from time, your program should also fit in the memory limit.
- But it’s usually weaker than TL. Too much appends to lists nearly always TL, not ML.

Memory

- Aside from time, your program should also fit in the memory limit.
- But it’s usually weaker than TL. Too much appends to lists nearly always TL, not ML.
- The most common cause of ML — large arrays.
  But their size is easy to calculate explicitly.
  Only need to know sizes of variables.

Summary

- Your program is expected to work fast on worst-case inputs.

Summary

- Your program is expected to work fast on worst-case inputs.
- You should always get O bound before implementing.

СПбГУ

Summary

- Your program is expected to work fast on worst-case inputs.
- You should always get ***O*** bound before implementing.
- To check, plug limits in the bound and compare
  with possible number of operations.

Summary

- Your program is expected to work fast on worst-case inputs.
- You should always get ***O*** bound before implementing.
- To check, plug limits in the bound and compare with possible number of operations.
- Speed up only in bottlenecks.

Summary

- Your program is expected to work fast on worst-case inputs.
- You should always get O bound before implementing.
- To check, plug limits in the bound and compare with possible number of operations.
- Speed up only in bottlenecks.
- First optimize asymptotically. Only if this fails and you need very little optimize constants.

Summary

- Your program is expected to work fast on worst-case inputs.
- You should always get O bound before implementing.
- To check, plug limits in the bound and compare with possible number of operations.
- Speed up only in bottlenecks.
- First optimize asymptotically. Only if this fails and you need very little optimize constants.
- Could be useful to measure actual time.