Exercise Session 2

Data Structures and Algorithms, D-MATH, ETH Zurich

Program of today

- Feedback of last exercise
- C++ Container Library
- Templates Recap
- Repetition theory
  - Induction
- Use Case
  - Subarray Sum Problem

Landau Notation

- Give a correct definition of the set $\Theta(f)$ as compact as possible analogously to the definitions for sets $\mathcal{O}(f)$ and $\Omega(f)$.

Landau Notation

- Give a correct definition of the set $\Theta(f)$ as compact as possible analogously to the definitions for sets $\mathcal{O}(f)$ and $\Omega(f)$.
- $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists a > 0,\ b > 0,\ n_0 \in \mathbb{N} : a \cdot f(n) \leq g(n) \leq b \cdot f(n)\ \forall n \geq n_0\}$

Landau Notation

- Give a correct definition of the set $\Theta(f)$ as compact as possible analogously to the definitions for sets $\mathcal{O}(f)$ and $\Omega(f)$.
- $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists a > 0,\ b > 0,\ n_0 \in \mathbb{N} : a \cdot f(n) \leq g(n) \leq b \cdot f(n)\ \forall n \geq n_0\}$
- $\Theta(f) = \{g : \mathbb{N} \to \mathbb{R} \mid \exists c > 0,\ n_0 \in \mathbb{N} : \frac{1}{c} \cdot f(n) \leq g(n) \leq c \cdot f(n)\ \forall n \geq n_0\}$

Landau Notation

Prove or disprove the following statements, where $f, g : \mathbb{N} \to \mathbb{R}^+$.

(a) $f \in \mathcal{O}(g)$ if and only if $g \in \Omega(f)$.

(e) $\log_a(n) \in \Theta(\log_b(n))$ for all constants $a, b \in \mathbb{N} \setminus \{1\}$

(g) If $f_1, f_2 \in \mathcal{O}(g)$ and $f(n) := f_1(n) \cdot f_2(n)$, then $f \in \mathcal{O}(g)$.

Landau Notation

Sorting functions: if function f is left to function g, then $f \in \mathcal{O}(g)$.

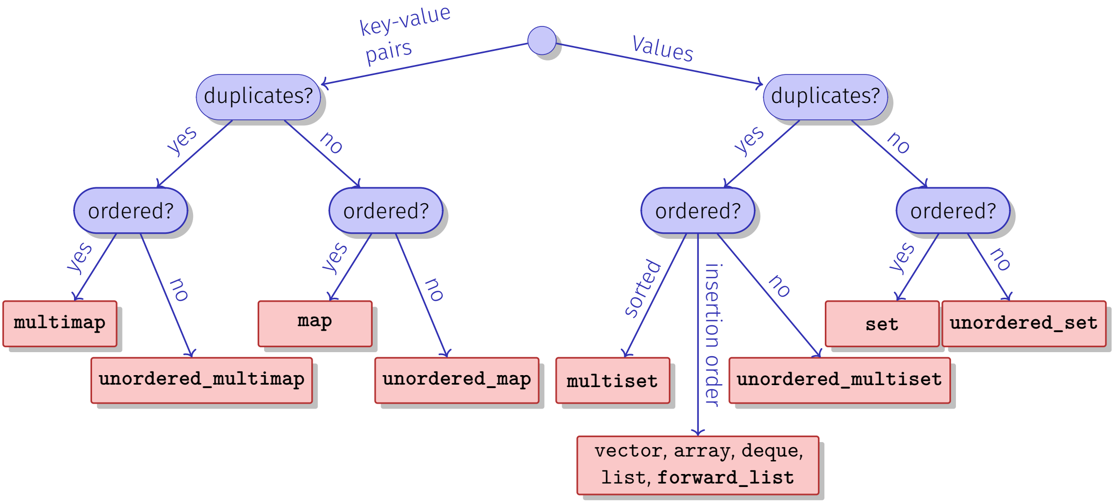
$2^{16},\ \log(n^4),\ \log^8(n),\ \sqrt{n},\ n\log n,\ \binom{n}{3},\ n^5+n,\ \frac{2^n}{n^2},\ n!,\ n^n.$

Sum of elements in two-dimensional array

Problems / Questions?

2. C++ Container Library

C++ Containers



Sequence-Container

`vector`	`array`	`deque`	`list`	`forward_list`
contiguous dynamic memory	contiguous static memory	Non-cont. dyn. memory	Non-cont. dyn. memory	Non-cont. dyn. memory
random access	random access	random access		
fast push/pop back		fast push/pop front/back	fast push/pop front/back	fast push/pop front
bid. iteration	bid. iteration	bid. iteration	bid. iteration	forward iteration

Sets and Multisets

- `std::set<E>` contains unique elements
- `std::multiset<E>` allows duplicate elements
    - Iteration yields all elements in decreasing order (in non-deterministic order if `unordered_multiset`)
    - `std::multiset<E>::count(elem)` returns the number of occurences of a given element

Sets and Multisets

- `std::set<E>` contains unique elements
- `std::multiset<E>` allows duplicate elements
  - Iteration yields all elements in decreasing order (in non-deterministic order if `unordered_multiset`)
  - `std::multiset<E>::count(elem)` returns the number of occurences of a given element

Example of `std::multiset`

```
Content: Xanten Xenon Xenon Xenon Xerografie Xerophil Xylose
count("Xenon") = 3
count("Xylose") = 1
```

Maps and Multimaps

- `std::map<K,V>` contains pairs (key, value), where a key maps to at most one value
- `std::multimap<K,V>` allows duplicate pairs
  - Iteration yields all pairs in descending key order (in non-deterministic order, if `unordered_multimap`)
  - `std::multimap<K,V>::count(key)` returns the number of occurrences of a given key
  - `std::multimap<K,V>::equal_range(key)` returns all values (in non-det. order) for a given key

Maps and Multimaps

- **`std::map<K,V>`** contains pairs (key, value), where a key maps to at most one value
- **`std::multimap<K,V>`** allows duplicate pairs
  - Iteration yields all pairs in descending key order (in non-deterministic order, if **`unordered_multimap`**)
  - **`std::multimap<K,V>::count(key)`** returns the number of occurrences of a given key
  - **`std::multimap<K,V>::equal_range(key)`** returns all values (in non-det. order) for a given key

Example of `std::multimap<K,V>`

```
Content: {2, er} {2, du} {2, es} {3, Axt} {3, sie} {4, Igel}
count(2) = 3
Values for key 2: er du es
```

3. Templates Recap

Motivation

Goal: generic binary tree without duplicating code

```
class Node { ... }; // Node of a binary search tree
auto n1 = Node<int>(5);
auto n2 = Node<std::string>("Zürich");
n1.insert(1);
n2.contains(2); // Compiler error
```

Motivation

Goal: generic binary tree without duplicating code

```
class Node { ... }; // Node of a binary search tree
auto n1 = Node<int>(5);
auto n2 = Node<std::string>("Zürich");
n1.insert(1);
n2.contains(2); // Compiler error
```

Idea:

- Make classes and functions parametric in types (= template parameters)

  ...

- ... just as they are already parametric in values (= function parameters)

Types as Template Parameters

1. In the concrete implementation of a class replace the type that should become generic (e.g. **`int`**) by a representative element, e.g. **`T`**.
2. Put in front of the class the construct **`template<typename T>`** Replace **`T`** by the representative name).

The construct **`template<typename T>`** can be understood as **“for all types** `T`”.

Class template

```
template <typename K>
class Node {
  K key;
  Node* left, right;
public:
  Node(K k, Node* l, Node* r): key(k), left(l), right(r) {}

  bool contains(K search_key) const {
    return search_key != key
      || left != nullptr && left->contains(search_key)
      || right != nullptr && right->contains(search_key)
  }
  ...
};
```

Function Template: Analogous Approach

1. To make a concrete implementation generic, replace the specific type (e.g. int) with a name, e.g. **`T`**,
2. Put in front of the function the construct **`template<typename T>`** (Replace **`T`** by the chosen name)

Examples

- For free functions

```
template <typename T>
void swap(T& x, T& y) {
  T temp = x;
  x = y;
  y = temp;
}
```

```
template <typename Iter>
void is_sorted(Iter begin, Iter end){
  ...
}
```

- For operators

```
template <typename T>
ostream& operator<<(ostream& out, const Node<T> root) {
  ...
}
```

Semantics (Code-Generation)

For each template instance, the compiler creates a corresponding instantiated class (or function) → static code generation

```
Node<int> n1 = ...;
Node<std::string> n2 = ...;
Node<Student> n3 = ...;
```

n1 n2 n3

```
class Node_int {
  int key;
  ...
  bool contains(int k) {...}
  int max() {...}
};
```

```
class Node_string {
  std::string key;
  ...
};
```

```
class Node_Student {
  Student key;
  ...
};
```

Semantics (Code-Generation)

For each template instance, the compiler creates a corresponding instantiated class (or function) $\rightarrow$ static code generation

Question: what does this imply for seperate compilation?

- Should templates go into .h (declarations) or .cpp (definitions) files?
- Is it possible to ship the compiled implementation (binary file compiled from .cpp) alongside the header file?

Type testing

- Templates: syntactic checks
- Instances: checks as usual

```
template <typename T>
T abs(T v) {
  return 0 <= v ? v : -v;
}
// main
foo(8); // OK
```

```
emplate <typename T>
T abs(T v) {
  return 0 <= v ? v : -v; // Error
}
// main
foo("hi"); // Error
```

```
template <typename T>
void swap(T& x, T& y) {
  ...
}
// main
double a = 1.0;
double b = 7;
swap(a, b); // OK
```

```
template <typename T>
void swap(T& x, T& y) {
  ...
}
// main
double a = 1.0;
string b = "seven";
swap(a, b); // Error
```

Other Languages

All languages try to foster code reuse but chose different solutions.

- C++, Rust:
  - static code generation
  - no runtime overhead
  - difficult to integrate into OOP
- C#, Scala (, Java)
  - type parameters are turned into runtime values
  - well-suited for OOP
  - minor runtime overhead
- Python, JavaScript:
  - dynamic typing (duck typing)
  - no syntactic overhead
  - potentially significant runtime overhead

3.1 auto vs templates

auto

- Placeholder type specifier
  - Must be uniquely determined by direct context: initialiser code, or returns
  - User could write type themself, but leave it to the compiler

```
std::vector<int> vec = ...;
auto it = vec.cbegin();
// placeholder for td::vector<int>::const_iterator
```

- Failing examples:

```
auto x; // x has no initializer
x = 0.0;
auto first_or_else(std::vector<int> data, unsigned int or_else) {
  if (data.size() == 0) return or_else;
  else return data[0];
}
```

Templates

- Parameters are unknown until instantiated

```
template <typename N>
char sign(N v) {
  if (0 <= v) return '+';
  else return '-';
}

template <typename T1, typename T2>
struct Pair {
  T1 fst;
  T2 snd;
};
```

- Instantiation may happen anywhere

```
Pair<int, double> p1 = Pair{1, 0.1};
auto p2 = Pair<std::string, bool>{"Brazil", true};
```

Combining templates and auto

auto auto inside template must be determined after instantiation

```
template <typename C>
void print(C container) {
  for (auto& e : container)
  std::cout << e << ' ';
}

std::vector<int> numbers = {1, 2, 3};
print(numbers); // now auto can be determined

std::vector<std::string> airports = {"LAX", "LDN", "ZHR"};
print(airports); // now auto can be determined
```

Combining templates and auto

`auto` auto inside template must be determined after instantiation

```
template <typename C>
void print(C container) {
  for (auto& e : container)
  std::cout << e << ' ';
}
```

Question: Is it possible to not use `auto` here?

Combining templates and auto

`auto` auto inside template must be determined after instantiation

```
template <typename C>
void print(C container) {
  for (auto& e : container)
  std::cout << e << ' ';
}
```

Question: Is it possible to not use **`auto`** here?

Answer: Yes, for exampl by replacing auto with an additional template parameter **`E`**

From auto to templates

- Before **C++20** aufo function parameters are forbidden

```
void print(auto x) {...} // Compiler error
```

From auto to templates

- Before **C++20** aufo function parameters are forbidden

```
void print(auto x) {...} // Compiler error
```

Question: Why do you think that is?

From auto to templates

- Before `C++20` aufo function parameters are forbidden

  ```
  void print(auto x) {...} // Compiler error
  ```

  Question: Why do you think that is?
  Answer: Cannot determine type from context

From auto to templates

- Before **C++20** aufo function parameters are forbidden

  ```
  void print(auto x) {...} // Compiler error
  ```

  Question: Why do you think that is?
  Answer: Cannot determine type from context
- Since **C++20** aufo function parameters are allowed

  ```
  void print(auto x) {...} // ok
  ```

  Clearly, it is still not possible to determine what auto stands for.

From auto to templates

- Before **C++20** aufo function parameters are forbidden

  ```
  void print(auto x) {...} // Compiler error
  ```

  Question: Why do you think that is?
  Answer: Cannot determine type from context

- Since **C++20** aufo function parameters are allowed

  ```
  void print(auto x) {...} // ok
  ```

  Clearly, it is still not possible to determine what auto stands for.
  Question: What could be the meaning of auto in this case??

From auto to templates

- Before **C++20** aufo function parameters are forbidden

```
void print(auto x) {...} // Compiler error
```

Question: Why do you think that is?
Answer: Cannot determine type from context

- Since **C++20** aufo function parameters are allowed

```
void print(auto x) {...} // ok
```

Clearly, it is still not possible to determine what auto stands for.
Question: What could be the meaning of auto in this case??
Answer: It is just a shorthand for a template parameter

```
template <typename T>
void Print(T x){ ... }
```

4. Repetition theory

Induction: what is required?

- Prove statements, for example $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Induction: what is required?

- Prove statements, for example $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Base clause:
  - The given (in)equality holds for one or more base cases.
  - e.g. $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$.

Induction: what is required?

- Prove statements, for example $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Base clause:
  - The given (in)equality holds for one or more base cases.
  - e.g. $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$.
- Induction hypothesis: we assume that the statement holds for some n

Induction: what is required?

- Prove statements, for example $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- Base clause:
  - The given (in)equality holds for one or more base cases.
  - e.g. $\sum_{i=1}^{1} i = 1 = \frac{1(1+1)}{2}$.
- Induction hypothesis: we assume that the statement holds for some n
- Induction step ($n \to n+1$):
  - From the validity of the statement for n (induction hypothesis) it follows the one for $n+1$.
  - e.g.: $\sum_{i=1}^{n+1} i = n + 1 + \sum_{i=1}^{n} i = n + 1 + \frac{n(n+1)}{2} = \frac{(n+2)(n+1)}{2}$.

Induction: Example

- Show $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$.

Induction: Example

- Show $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$.
- Base clause:
  $n = 0$: $\sum_{i=0}^{0} r^i = 1 = \frac{1-r^1}{1-r}$.

Induction: Example

- Show $\sum_{i=0}^{n} r^i = \frac{1-r^{n+1}}{1-r}$.
- Base clause:
  $n = 0$: $\sum_{i=0}^{0} r^i = 1 = \frac{1-r^1}{1-r}$.
- Induction step $(n \to n+1)$:

$$\begin{aligned}\sum_{i=0}^{n+1} r^i &= r^{n+1} + \sum_{i=0}^{n} r^i \\ &= r^{n+1} + \frac{1-r^{n+1}}{1-r} = \frac{r^{n+1} - r^{n+2} + 1 - r^{n+1}}{1-r} = \frac{1-r^{n+2}}{1-r}.\end{aligned}$$

[Besides..]

It can be shown easily in a direct manner

$$
\begin{aligned}
\frac{r^{n+1}-1}{r-1} &\stackrel{!}{=} \sum_{i=0}^{n} r^i \\
(r-1)\cdot\sum_{i=0}^{n} r^i &= \sum_{i=0}^{n} r^{i+1} - \sum_{i=0}^{n} r^i \\
&= \sum_{i=1}^{n+1} r^i - \sum_{i=0}^{n} r^i = \sum_{i=0}^{n+1} r^i - 1 - \sum_{i=0}^{n} r^i \\
&= r^{n+1} - 1
\end{aligned}
$$

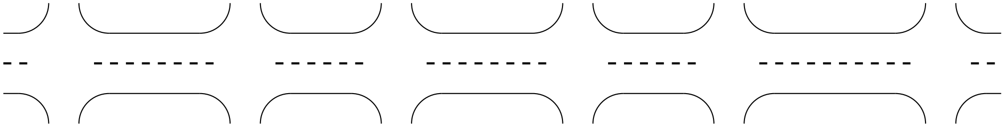
5. Use Case

5.1 Subarray Sum Problem

Naïve Solution, prefix sums, binary search, Sliding Window

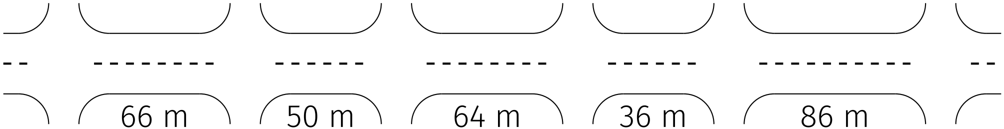
Street section of a given length

Street section of a given length



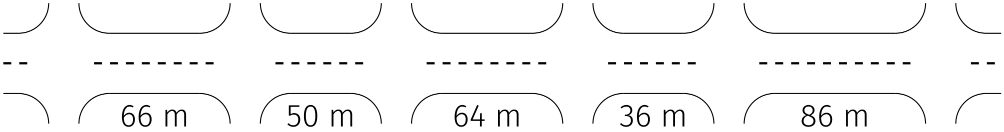
Street section of a given length

Given: distances between all crossroads on a street



Street section of a given length

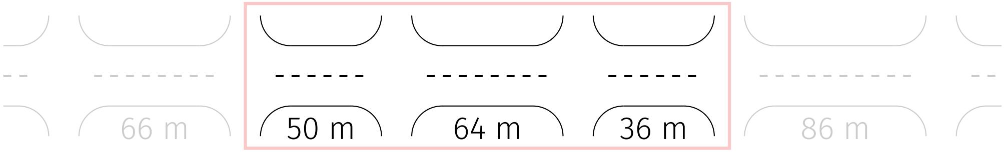
Given: distances between all crossroads on a street



Wanted: street section of length 150 meters between crossroads

Street section of a given length

Given: distances between all crossroads on a street



Wanted: street section of length 150 meters between crossroads

Subarray Sum Problem

Subarray Sum Problem

Given: a sequence $a[0], \ldots, a[n-1]$ of non-negative integers

Wanted: a subsequence with sum k:

pair (l, r) with $0 \leq l \leq r \leq n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Subarray Sum Problem

Given: a sequence $a[0], \ldots, a[n-1]$ of non-negative integers

Wanted: a subsequence with sum k:

pair (l, r) with $0 \leq l \leq r \leq n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Example: $n = 9, k = 7$

1	2	2	3	1	4	2	2	3
0	1	2	3	4	5	6	7	8

Subarray Sum Problem

Given: a sequence $a[0], \dots, a[n-1]$ of non-negative integers

Wanted: a subsequence with sum k:

pair (l, r) with $0 \le l \le r \le n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Example: $n = 9, k = 7$ **Solution:** $l = 1, r = 3$.

1	2	2	3	1	4	2	2	3
0	1	2	3	4	5	6	7	8
	l		**r**					

Strategies?

Given: a sequence $a[0], \ldots, a[n-1]$ of non-negative integers
Wanted: a subsequence with sum k:
pair (l, r) with $0 \leq l \leq r \leq n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Strategies

$\Theta(n^3)$	Three loops
$\Theta(n^2)$	?
$\Theta(n \log n)$	?
$\Theta(n)$	?

Strategies?

Given: a sequence $a[0], \ldots, a[n-1]$ of non-negative integers
Wanted: a subsequence with sum k:
pair (l, r) with $0 \leq l \leq r \leq n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Strategies

$\Theta(n^3)$	Three loops
$\Theta(n^2)$	Prefix Sums
$\Theta(n \log n)$	?
$\Theta(n)$	?

Strategies?

Given: a sequence $a[0], \ldots, a[n-1]$ of non-negative integers
Wanted: a subsequence with sum k:
pair (l, r) with $0 \leq l \leq r \leq n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Strategies

$\Theta(n^3)$	Three loops
$\Theta(n^2)$	Prefix Sums
$\Theta(n \log n)$	Binary Search
$\Theta(n)$	?

Strategies?

Given: a sequence $a[0], \dots, a[n-1]$ of non-negative integers
Wanted: a subsequence with sum k:
pair (l, r) with $0 \le l \le r \le n-1$ such that $\sum_{i=l}^{r} a[i] = k$

Strategies

$\Theta(n^3)$	Three loops
$\Theta(n^2)$	Prefix Sums
$\Theta(n \log n)$	Binary Search
$\Theta(n)$	Sliding Window

Subarray Sum Problem: Sliding Window

Sliding Window Idea

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
    - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
    - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
    - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9

$\uparrow$

l, r

window sum $=$ **2**

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
$\uparrow$	$\uparrow$								
l	**r**								

window sum $=$ **5**

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
↑ **l**		↑ **r**							

window sum $=$ **6**

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
$\uparrow$			$\uparrow$						
l			**r**						

window sum $=$ **8**

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
	↑		↑						
	l		**r**						

window sum $=$ **6**

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
	$\uparrow$			$\uparrow$					
	l			**r**					

window sum $=$ **8**

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
		↑		↑					
		l		**r**					

window sum $=$ **5**

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
		↑			↑				
		l			**r**				

window sum $=$ **8**

Subarray Sum Problem: Sliding Window

Sliding Window Idea

- start with left and right pointer at 0
- repeat until the end of the sequence:
  - window **too small** (sum $< k$) $\Rightarrow$ increment right pointer
  - window **too large** (sum $> k$) $\Rightarrow$ increment left pointer
  - window **as desired** (sum $= k$) $\Rightarrow$ done!

Example: $k = 7$

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
			$\uparrow$		$\uparrow$				
			l		**r**				

window sum $=$ **7**

Subarray Sum Problem: Sliding Window Analysis

Subarray Sum Problem: Sliding Window Analysis

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9

Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9

Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9

Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start

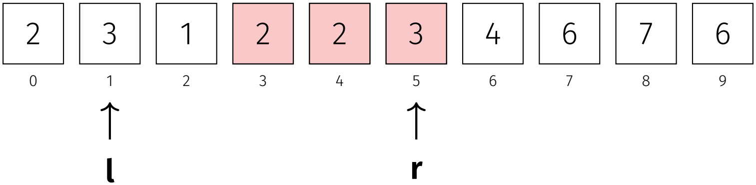
2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
	↑				↑				
	l				**r**				

Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large

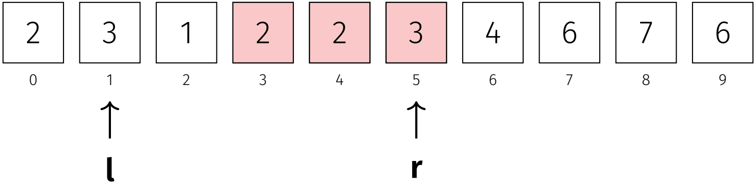


Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window

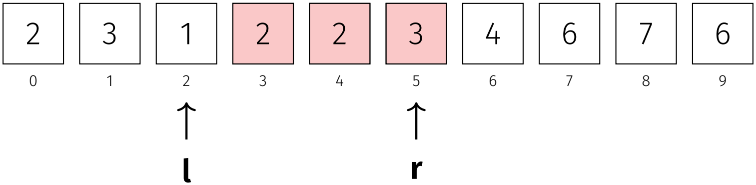


Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window

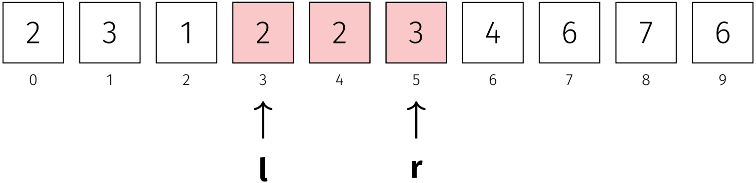


Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window

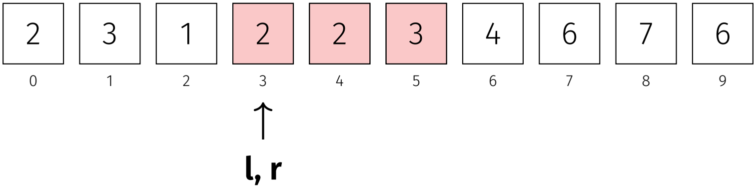


Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window
- if l reaches the start before r reaches the end

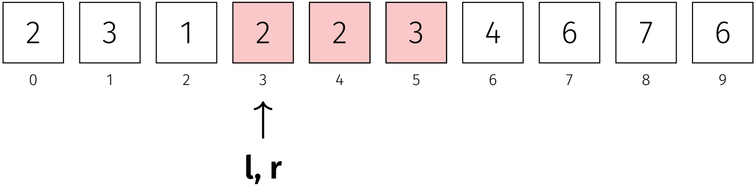


Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window
- if l reaches the start before r reaches the end
  $\Rightarrow$ sum too small



Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window
- if l reaches the start before r reaches the end
  $\Rightarrow$ sum too small $\Rightarrow$ r is increased until it reaches the end of the window

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9

↑

l, r

Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window
- if l reaches the start before r reaches the end
  $\Rightarrow$ sum too small $\Rightarrow$ r is increased until it reaches the end of the window

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
			↑	↑					
			l	**r**					

Subarray Sum Problem: Sliding Window Analysis

- in each step: either l or r is increased
  $\Rightarrow$ algorithm terminates after a maximum of $2n$ steps

target window: lexicographically smallest (left-most) window with sum k

- if r reaches the end before l reaches the start
  $\Rightarrow$ sum too large $\Rightarrow$ l is increased until it reaches the start of the window
- if l reaches the start before r reaches the end
  $\Rightarrow$ sum too small $\Rightarrow$ r is increased until it reaches the end of the window

2	3	1	2	2	3	4	6	7	6
0	1	2	3	4	5	6	7	8	9
			↑		↑				
			l		**r**				

Analysis

We consider the lexicographically smallest (left-most) window with sum k, called *target window*

- In each step of the algorithm either l or r is increased. The algorithm terminates after a maximum of $2n$ steps.
- Assume r reaches the end of the target window before l reaches the start of the target window, then l keeps increasing until it reaches the start of the window.
- Assume l reaches the start of the target window before r reaches the end of the target window, then r keeps increasing until it reaches the end of the window.

Exercise: window with sum closest to k

Questions or Suggestions?