HPI Hasso Plattner Institut
Digital Engineering · Universität Potsdam

MOIA

Karlsruhe Institute of Technology

Algorithm Bootcamp

July 2024

Day 1: Introduction

Dr. Christopher Weyand
Optimization Expert
Fleet Optimization

David Stangl
Algorithm Engineer
Fleet Optimization

German Collegiate Programming Contest 2024

RANK	TEAM		SCORE		A	B	C	D	E	F	G	H	I	J	K	L	M
1	KIT	Participants Kindergarten Timelimit Karlsruher Institut für Technologie	10	828	5 1 try	13 1 try	121 1 try	78 2 tries	141 1 try		105 4 tries	220 1 try	30 1 try		24 1 try		11 1 try
2	KIT	Infinite Loopers Karlsruher Institut für Technologie	10	1062	6 1 try	20 1 try	114 1 try	58 1 try	107 1 try		294 4 tries		106 5 tries		19 1 try	187 1 try	11 1 try
3		addicted (òvó) Universität des Saarlandes	10	1325	7 1 try	12 1 try	283 1 try	51 1 try	111 1 try		194 3 tries		122 1 try	297 5 tries	58 1 try		70 1 try
4	HPI	Seems to be O(k!) Hasso-Plattner-Institut	9	757	13 1 try	16 1 try	253 1 try	164 3 tries	65 1 try		128 2 tries		34 1 try		19 1 try		5 1 try
5	KIT	Künstliche InkompeTenz Karlsruher Institut für Technologie	9	1116	15 1 try	22 1 try	232 1 try	166 3 tries	288 2 tries		104 1 try		112 1 try		49 1 try		68 1 try
6	RWTH	r/wth Rheinisch-Westfälische Technische Hochschule Aachen	8	580	11 1 try	25 1 try	81 1 try	113 1 try	236 1 try	1 try			56 1 try		40 1 try		18 1 try
7	TUM	MMR is just a number Technische Universität München	8	952	17 1 try	23 1 try	1 try	263 1 try	205 5 tries				132 2 tries		98 1 try	86 1 try	28 1 try
8	RWTH	Wrong Answer Rheinisch-Westfälische Technische Hochschule Aachen	7	575	14 1 try	26 1 try		131 1 try	215 2 tries		8 tries		103 1 try		44 1 try		22 1 try
9	HPI	First Time Participants ChillMorphiumUSER Hasso-Plattner-Institut	7	628	29 1 try	38 1 try	168 1 try		4 tries		246 2 tries		64 1 try		17 1 try		46 1 try
10	KIT	cracK IT Karlsruher Institut für Technologie	7	702	16 3 tries	12 1 try	265 1 try			5 tries	169 5 tries		60 1 try		28 1 try		32 1 try

We are the remnants of **Hasso's Race Mice**

Hello Barcelona 2018

GCPC2022
GCPC2021
GCPC2019
GCPC2018

2		**Hasso's Race Mice** Hasso-Plattner-Institut	11	1225
2		**Hasso's Race Mice** Out of Competition		11
8		Hasso's Race Mice Hasso-Plattner-Institut	8	970
17		Hasso's Race Mice HPI Potsdam	7	1091

Christopher Weyand

Master
2124

David Stangl

Grandmaster
2429

Discover Riga 2019

2. Hasso Platten I: Hexaflexagons (Weyand, Stangl)

DISCOVER the World 2020

41st Petrozavodsk Programming Camp Summer 2021

40th Petrozavodsk Programming Camp Winter 2021

13. (A/B:) Hasso Plattner I: Hanso Platten (Stangl, Weyand, Fischbeck)

Why Competitive Programming?

Fun!

Why Competitive Programming?

Fun!

Training of algorithmic thinking!

Why Competitive Programming?

Fun!

Training of algorithmic thinking!

Experience with Rust/C++

Why Competitive Programming?

Fun!

Training of algorithmic thinking!

Trains modelling/formalizing!

Experience with Rust/C++

Why Competitive Programming?

Fun!

Training of algorithmic thinking!

Trains modelling/formalizing!

Experience with Rust/C++

Trains systematic problem solving!

Why Competitive Programming?

Fun!

Training of algorithmic thinking!

Trains modelling/formalizing!

Experience with Rust/C++

Trains systematic problem solving!

Repeats paradigms like...

... Greedy Algos
... Dynamic Programming
... Divide & Conquer
...

Why Competitive Programming?

Fun!

Training of algorithmic thinking!

Trains modelling/formalizing!

Experience with Rust/C++

Trains systematic problem solving!

Repeats paradigms like...

... Greedy Algos
... Dynamic Programming
... Divide & Conquer
...

Prepares for interviews at Google,Microsoft & Co!

Why Competitive Programming?

Fun!

Training of algorithmic thinking!

Trains modelling/formalizing!

Experience with Rust/C++

Trains systematic problem solving!

Repeats paradigms like...

> Science is what we understand well enough to explain to a computer.
> Art is everything else we do.
>
> - Donald Knuth

... Greedy Algos
... Dynamic Programming
... Divide & Conquer
...

Prepares for interviews at Google,Microsoft & Co!

No one reads the title

Structure

- morning lecture
- coding for the rest of the day
- discuss solutions the next morning before the lecture

No one reads the title

Structure

- morning lecture
- coding for the rest of the day
- discuss solutions the next morning before the lecture

Topics

- concepts
- graphs
- dynamic programming
- data structures
- numbers

No one reads the title

Structure

- morning lecture
- coding for the rest of the day
- discuss solutions the next morning before the lecture

Topics

- concepts
- graphs
- dynamic programming
- data structures
- numbers

Programming Tasks

- 6-7 per day
  expect you to solve 3-5
- cover topics discussed in lecture
- domjudge judging plattform
- use Python, C++, Rust
  it’s totally possible to learn C++ or Rust now

No one reads the title

Structure

- morning lecture
- coding for the rest of the day
- discuss solutions the next morning before the lecture

Topics

- concepts
- graphs
- dynamic programming
- data structures
- numbers

Programming Tasks

- 6-7 per day
  expect you to solve 3-5
- cover topics discussed in lecture
- domjudge judging plattform
- use Python, C++, Rust
  it’s totally possible to learn C++ or Rust now

Rules for Coding

✓ Use any software, IDE, language
you will be judged, though :)

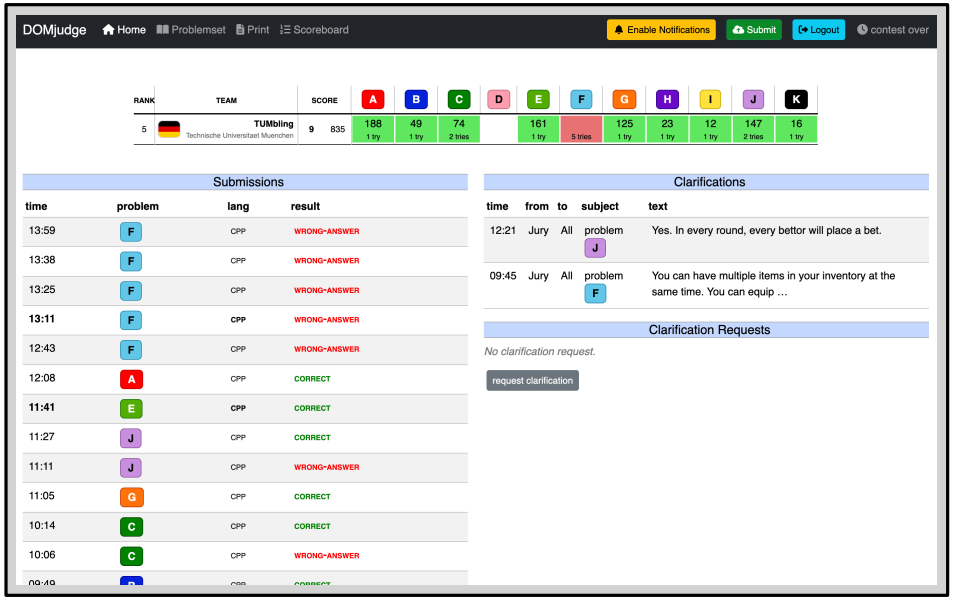
✓ Ask the teaching team for clarifications
... and hints!

✓ Use slides and code from the course

✓ Use your **own** code from previous problems

✗ try not to copy code from the Internet
when stuck, ask us instead

Judge Demo



https://judge.playground.moia-group.io/

https://github.com/chistopher/CPT

https://github.com/Felerius/rust-compprog-template

Greedy Algorithms

Greedy Algorithms

- always do/choose the best option at this moment

Greedy Algorithms

- always do/choose the best option at this moment

Never trust a greedy algorithm!

Greedy Algorithms

- always do/choose the best option at this moment

Never trust a greedy algorithm!

- greedy algorithms always look temptingly correct

Greedy Algorithms

- always do/choose the best option at this moment

Never trust a greedy algorithm!

- greedy algorithms always look temptingly correct
- correctness can usually be proven easily (e.g., with exchange argument)

Greedy Algorithms

- always do/choose the best option at this moment

Never trust a greedy algorithm!

- greedy algorithms always look temptingly correct
- correctness can usually be proven easily (e.g., with exchange argument)
- often easy to implement

Greedy Algorithms

- always do/choose the best option at this moment

Never trust a greedy algorithm!

- greedy algorithms always look temptingly correct
- correctness can usually be proven easily (e.g., with exchange argument)
- often easy to implement
- often very efficient

Greedy Algorithms

- always do/choose the best option at this moment

Never trust a greedy algorithm!

- greedy algorithms always look temptingly correct
- correctness can usually be proven easily (e.g., with exchange argument)
- often easy to implement
- often very efficient

Types of Greedy Algorithms

Greedy Construction

Scanline

Greedy Algorithms

- always do/choose the best option at this moment

Never trust a greedy algorithm!

- greedy algorithms always look temptingly correct
- correctness can usually be proven easily (e.g., with exchange argument)
- often easy to implement
- often very efficient

Types of Greedy Algorithms

Greedy Construction

- create desired structure step by step, always consider best extension of current structure

Scanline

Greedy Algorithms

- always do/choose the best option at this moment

Never trust a greedy algorithm!

- greedy algorithms always look temptingly correct
- correctness can usually be proven easily (e.g., with exchange argument)
- often easy to implement
- often very efficient

Types of Greedy Algorithms

Greedy Construction

- create desired structure step by step, always consider best extension of current structure

Scanline

- scan linearly over the input, handle all “events” appropriately

Greedy Algorithms

Problem: Given an array of pairwise different integers. Create and alternating array with these integers!

alternating:  or 

Greedy Algorithms

Problem: Given an array of pairwise different integers. Create and alternating array with these integers!

alternating:  or 

Solution idea:

Greedy Algorithms

Problem: Given an array of pairwise different integers. Create and alternating array with these integers!

alternating:  or 

Solution idea:

Scanline!

Greedy Algorithms

Problem: You are given the results of an all-vs-all tennis tournament. Find an order of the players such that the first won against the second, the second against the third, . . . , the second-to-last against the last.

Greedy Algorithms

Problem: You are given the results of an all-vs-all tennis tournament. Find an order of the players such that the first won against the second, the second against the third, . . . , the second-to-last against the last.

Solution idea:

Greedy Algorithms

Problem: You are given the results of an all-vs-all tennis tournament. Find an order of the players such that the first won against the second, the second against the third, . . . , the second-to-last against the last.

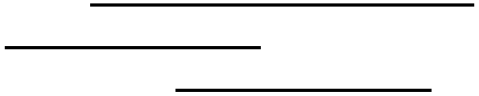
Solution idea:

Greedy construction!

(This is a hamilton path in a tournament graph)

Greedy Algorithms

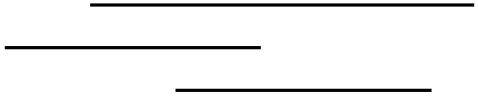
Problem: Given a collection of time intervals. Find the maximum number of intervals that are active simultaneously.



Greedy Algorithms

Problem: Given a collection of time intervals. Find the maximum number of intervals that are active simultaneously.

Scanline!

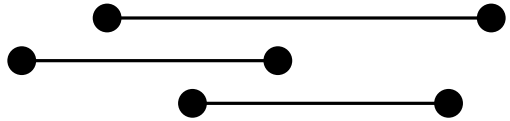


Greedy Algorithms

Problem: Given a collection of time intervals. Find the maximum number of intervals that are active simultaneously.

Scanline!

- split each interval in two events: start, end

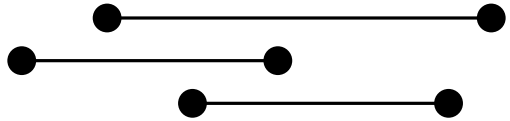


Greedy Algorithms

Problem: Given a collection of time intervals. Find the maximum number of intervals that are active simultaneously.

Scanline!

- split each interval in two events: start, end
- sort all events by time

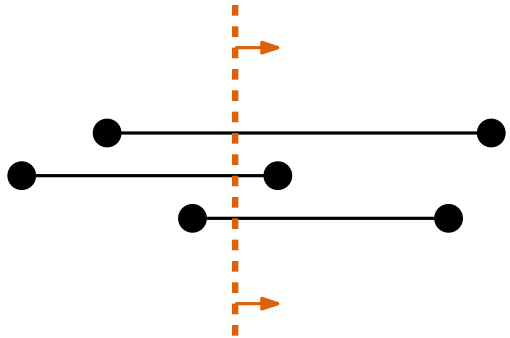


Greedy Algorithms

Problem: Given a collection of time intervals. Find the maximum number of intervals that are active simultaneously.

Scanline!

- split each interval in two events: start, end
- sort all events by time
- scan over events

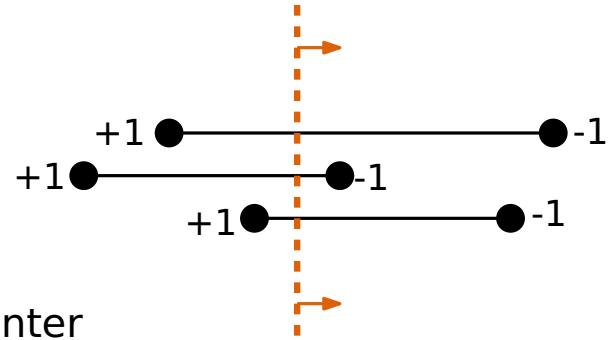


Greedy Algorithms

Problem: Given a collection of time intervals. Find the maximum number of intervals that are active simultaneously.

Scanline!

- split each interval in two events: start, end
- sort all events by time
- scan over events
- maintain the number of active intervals as a counter

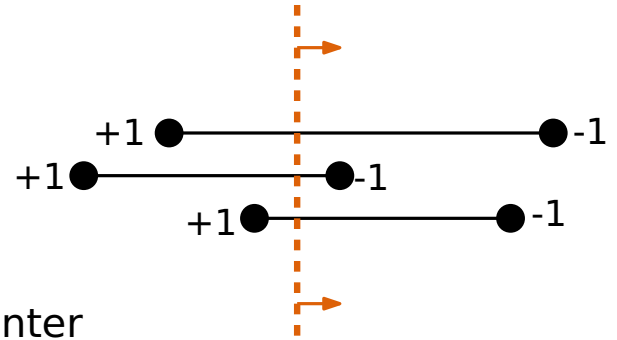


Greedy Algorithms

Problem: Given a collection of time intervals. Find the maximum number of intervals that are active simultaneously.

Scanline!

- split each interval in two events: start, end
- sort all events by time
- scan over events
- maintain the number of active intervals as a counter
- answer will be maximum value of the counter



The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

- Minimizing is hard!

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

- Minimizing is hard!
- Is it easier to find a solution for a given limit on the weight of the heaviest part?

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

- Minimizing is hard!
- Is it easier to find a solution for a given limit on the weight of the heaviest part?

  $\rightarrow$ yes, greedy!

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

- Minimizing is hard!
- Is it easier to find a solution for a given limit on the weight of the heaviest part?

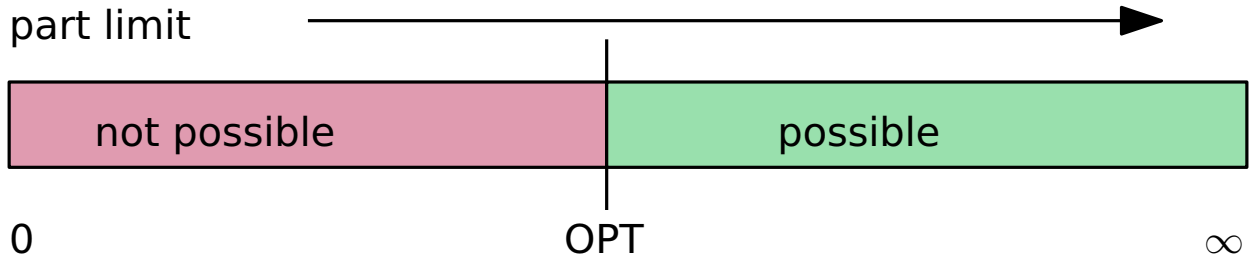
  $\rightarrow$ yes, greedy!
- Solution for a specific limit is also valid for all greater limits

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

- Minimizing is hard!
- Is it easier to find a solution for a given limit on the weight of the heaviest part?

  $\rightarrow$ yes, greedy!
- Solution for a specific limit is also valid for all greater limits

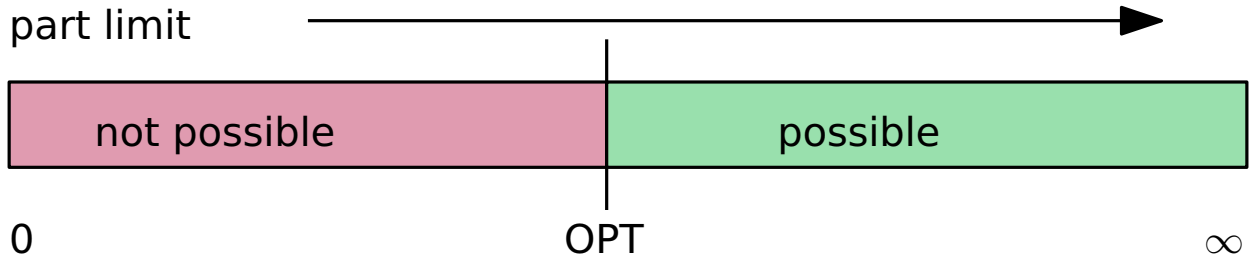


The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

- Minimizing is hard!
- Is it easier to find a solution for a given limit on the weight of the heaviest part?

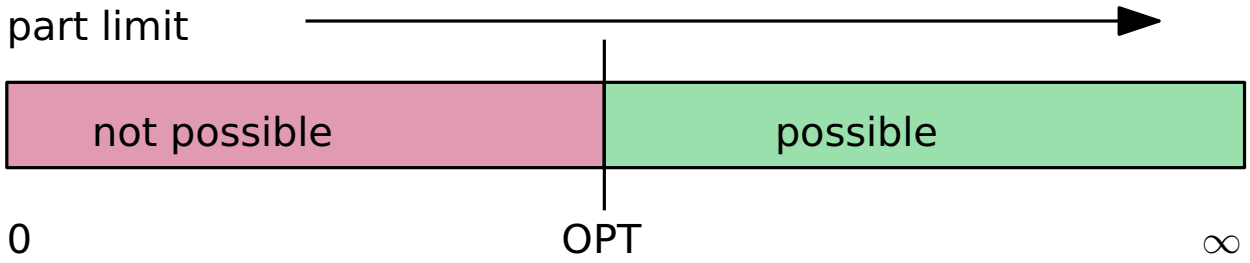
  $\rightarrow$ yes, greedy!
- Solution for a specific limit is also valid for all greater limits



$\rightarrow$ binary search over result

The Secret CompProg Weapon

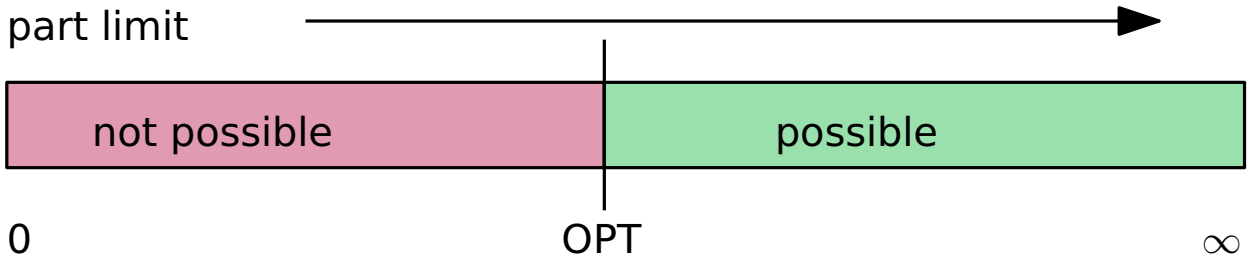
Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



$\rightarrow$ binary search over result

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.



$\rightarrow$ binary search over result

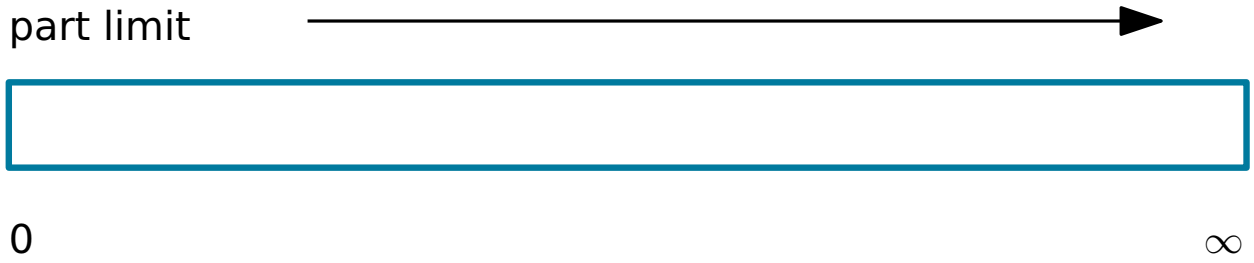
Example:

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.



$\rightarrow$ binary search over result

Example:

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

search range for sum s of heaviest subarray:

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

part limit →

~~0~~ 99 ∞

→ binary search over result

Example:

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

search range for sum s of heaviest subarray: $99 \leq s$

largest single value

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.

part limit →

~~0~~ 99 ~~∞~~ 1569

→ binary search over result

Example:

$k = 4$

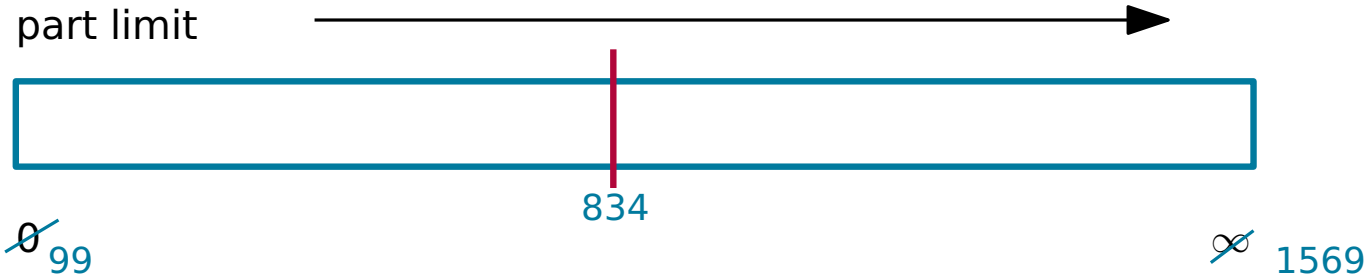
13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

largest single value — sum of all values

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most k *consecutive* parts s.t. the sum of the heaviest subarray is minimal.



$\rightarrow$ binary search over result

Example:

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

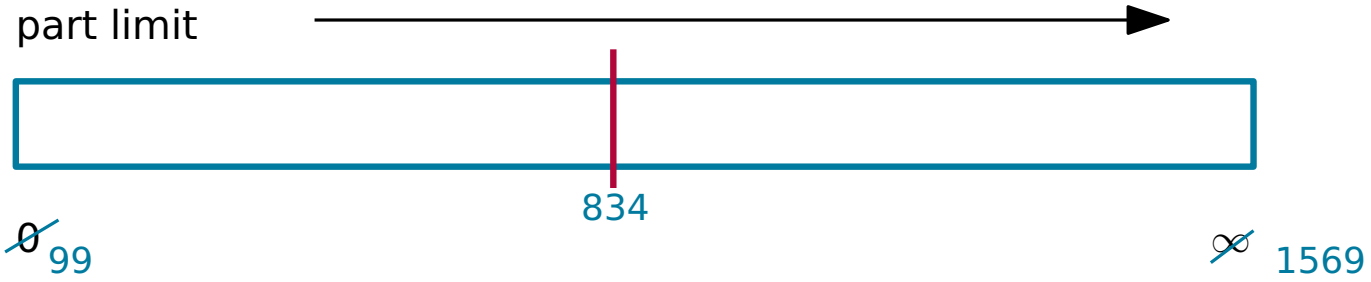
search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

largest single value — sum of all values

1.) try $s = 834$

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



Example:

$\rightarrow$ binary search over result

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

777 792

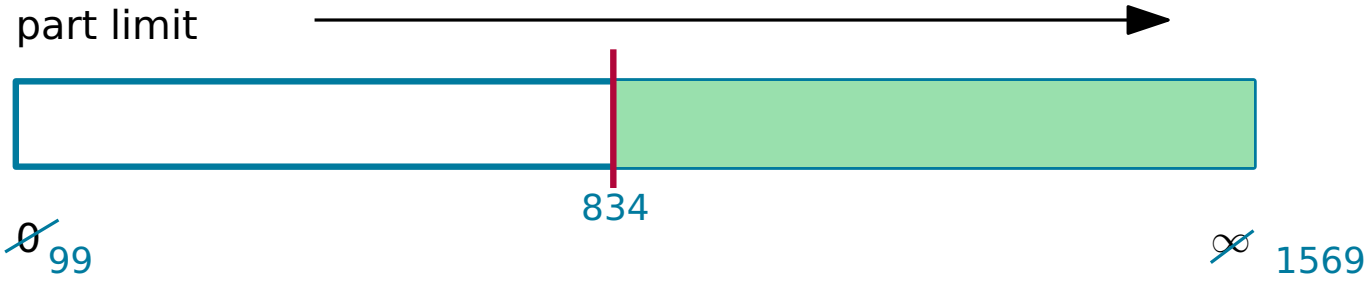
search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

largest single value — sum of all values

1.) try $s = 834$

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



Example:

$\rightarrow$ binary search over result

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

777 792

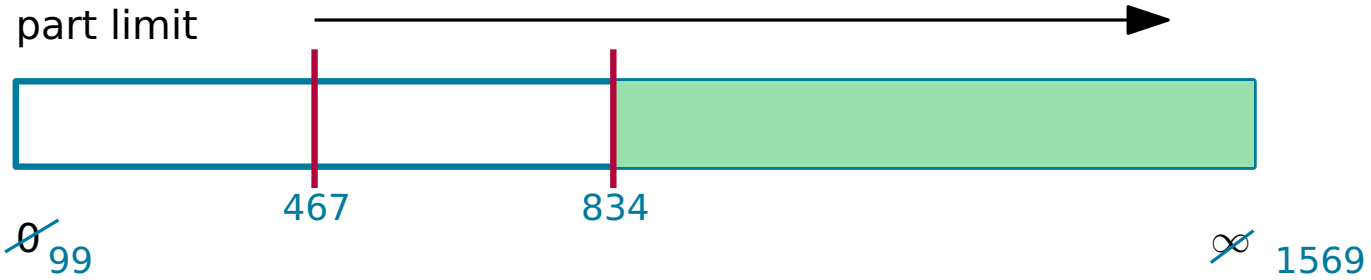
search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

largest single value — sum of all values

1.) try $s = 834$ ✓

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



Example:

$\rightarrow$ binary search over result

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

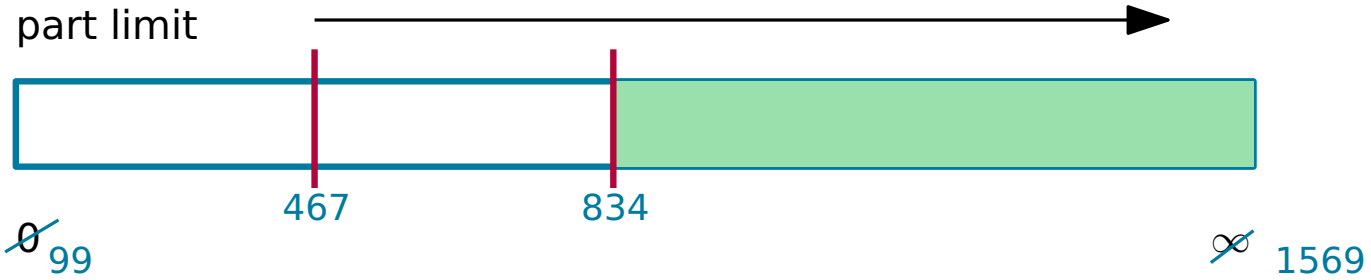
search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

largest single value — sum of all values

1.) try $s = 834$ ✓ 2.) try $s = 467$

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



Example:

$\rightarrow$ binary search over result

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16
430											446								432												261					

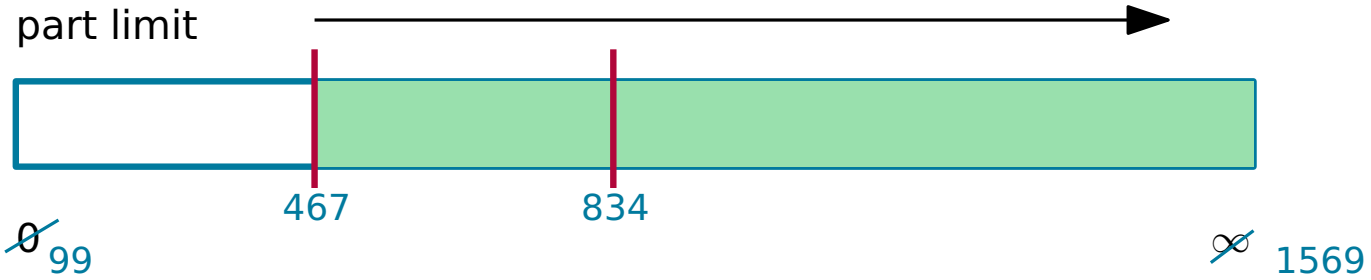
search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

largest single value — sum of all values

1.) try $s = 834$ ✓ 2.) try $s = 467$

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



Example:

$\rightarrow$ binary search over result

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

430 446 432 261

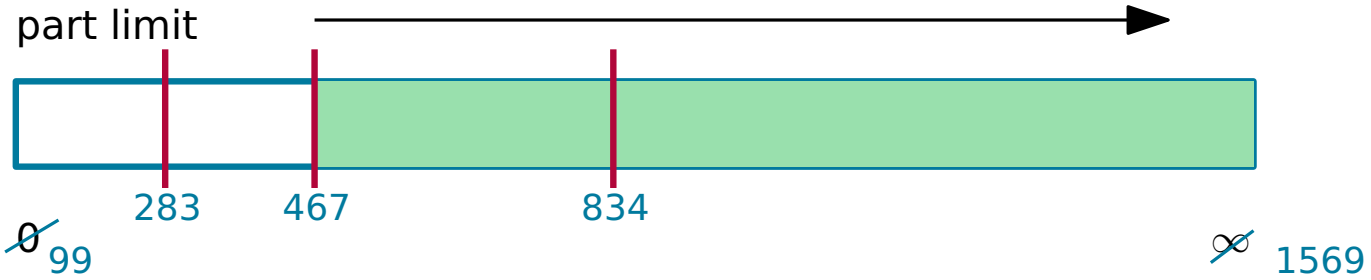
search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

largest single value — sum of all values

1.) try $s = 834$ ✓ 2.) try $s = 467$ ✓

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



Example:

$\rightarrow$ binary search over result

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

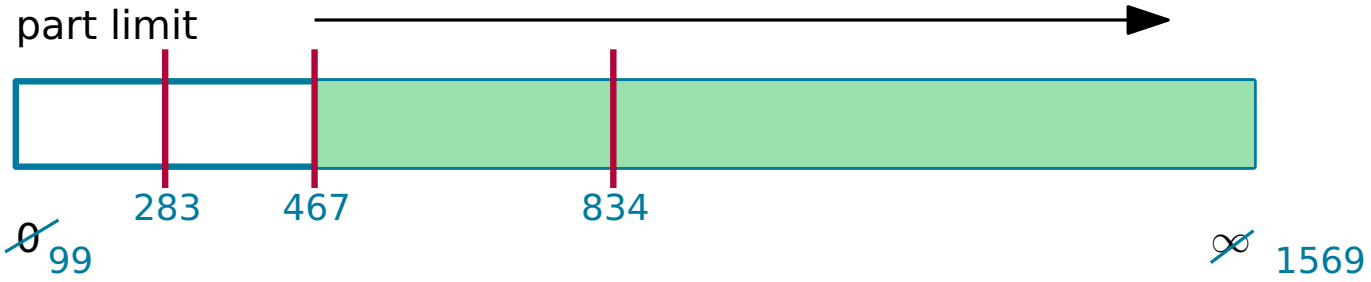
largest single value — sum of all values

1.) try $s = 834$ ✓ 2.) try $s = 467$ ✓

3.) try $s = 283$

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



Example:

$\rightarrow$ binary search over result

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

250 | 273 | 254 | > 283

search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

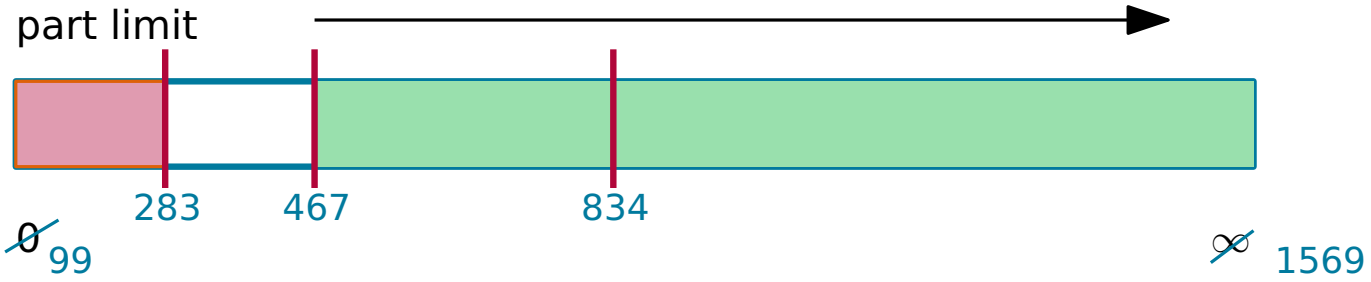
largest single value — sum of all values

1.) try $s = 834$ ✓ 2.) try $s = 467$ ✓

3.) try $s = 283$

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



Example:

→ binary search over result

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

250 | 273 | 254 | > 283

search range for sum *s* of heaviest subarray: $99 \leq s \leq 1569$

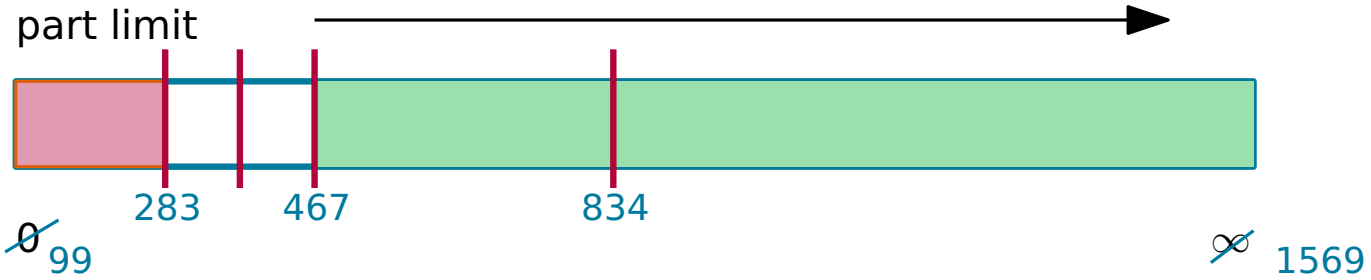
largest single value — sum of all values

1.) try $s = 834$ ✓ 2.) try $s = 467$ ✓

3.) try $s = 283$ ✗

The Secret CompProg Weapon

Problem: Cut an array of positive integers into at most *k consecutive* parts s.t. the sum of the heaviest subarray is minimal.



→ binary search over result

Example:

$k = 4$

13	17	66	22	89	38	5	57	58	45	20	50	43	88	73	11	82	99	42	1	19	33	67	2	89	45	7	73	12	42	91	36	40	23	55	16

search range for sum s of heaviest subarray: $99 \leq s \leq 1569$

99: largest single value; 1569: sum of all values

1.) try $s = 834$ ✓ 2.) try $s = 467$ ✓

3.) try $s = 283$ ✗ 4.) try . . .

repeat until optimum is found!

Binary Search

- very helpful to find elements (or ranges thereof) in containers (even map/set)

`std::lower_bound` / `std::upper_bound`

Binary Search

- very helpful to find elements (or ranges thereof) in containers (even map/set)

  `std::lower_bound` / `std::upper_bound`

- **but:** can also be used over every parameter that is monotone

Binary Search

- very helpful to find elements (or ranges thereof) in containers (even map/set)

  `std::lower_bound` / `std::upper_bound`

- **but:** can also be used over every parameter that is monotone
- often used over solution to turn optimization problem into decision problem

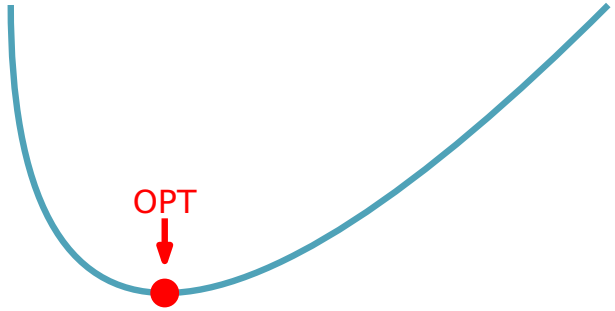
Binary Search

- very helpful to find elements (or ranges thereof) in containers (even map/set)

`std::lower_bound` / `std::upper_bound`

- **but:** can also be used over every parameter that is monotone
- often used over solution to turn optimization problem into decision problem

Ternary Search



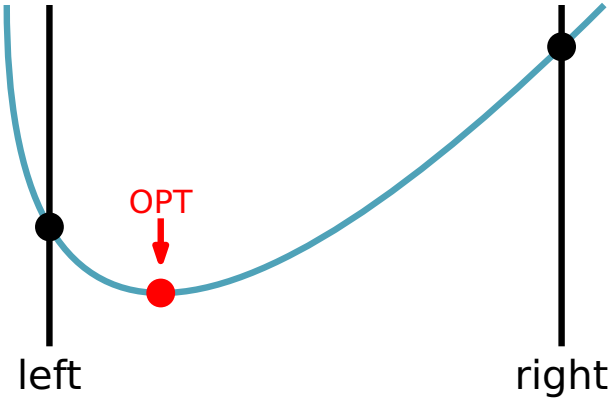
Binary Search

- very helpful to find elements (or ranges thereof) in containers (even map/set)

`std::lower_bound` / `std::upper_bound`

- **but:** can also be used over every parameter that is monotone
- often used over solution to turn optimization problem into decision problem

Ternary Search



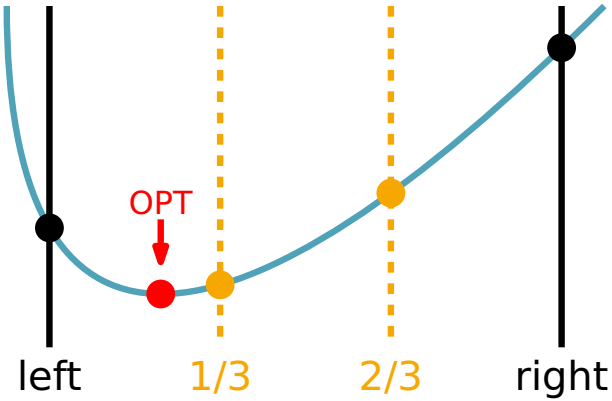
Binary Search

- very helpful to find elements (or ranges thereof) in containers (even map/set)

`std::lower_bound` / `std::upper_bound`

- **but:** can also be used over every parameter that is monotone
- often used over solution to turn optimization problem into decision problem

Ternary Search



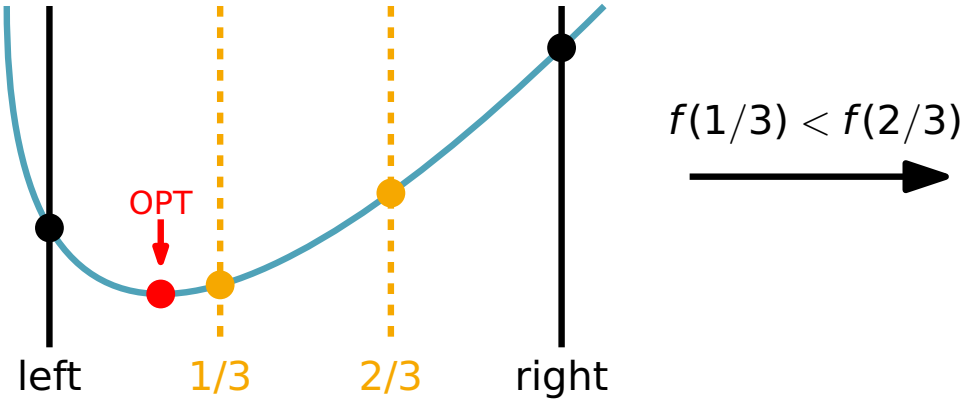
Binary Search

- very helpful to find elements (or ranges thereof) in containers (even map/set)

`std::lower_bound` / `std::upper_bound`

- **but:** can also be used over every parameter that is monotone
- often used over solution to turn optimization problem into decision problem

Ternary Search



$f(1/3) < f(2/3)$

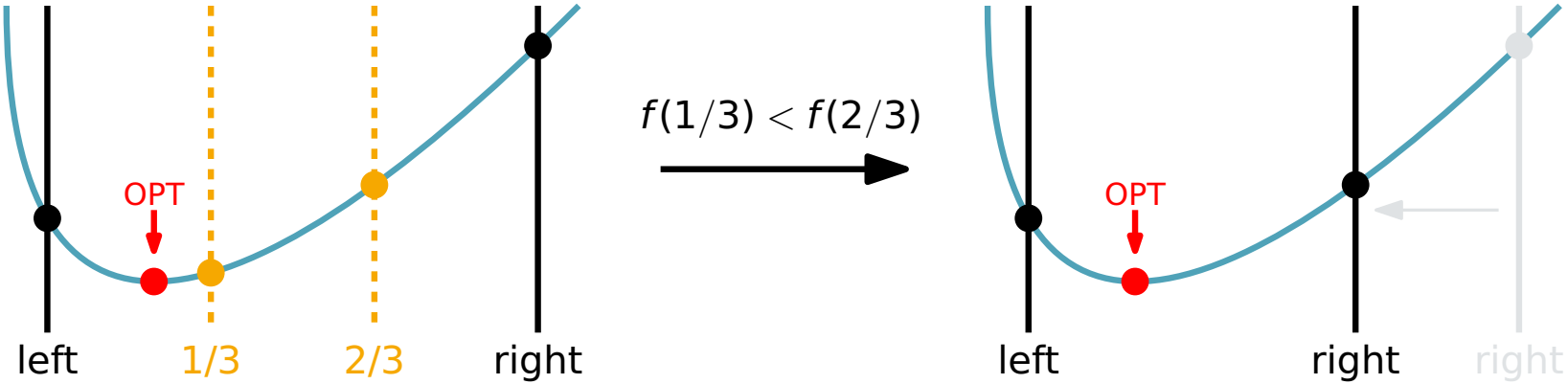
Binary Search

- very helpful to find elements (or ranges thereof) in containers (even map/set)

`std::lower_bound` / `std::upper_bound`

- **but:** can also be used over every parameter that is monotone
- often used over solution to turn optimization problem into decision problem

Ternary Search



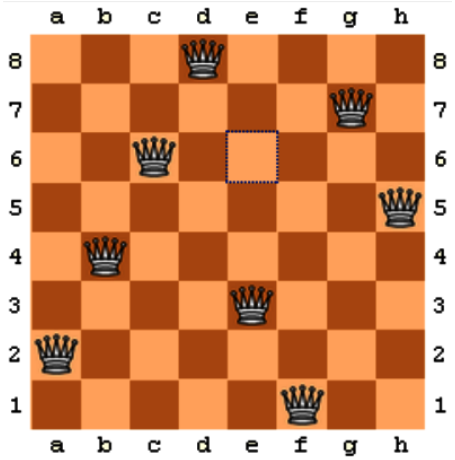
Brute Force

- Idea: search complete search space
- pro: very easy, always correct
- con: often too slow

Brute Force

- Idea: search complete search space
- pro: very easy, always correct
- con: often too slow

Example: Place 8 queens on a chess board s.t. they cannot capture each other.



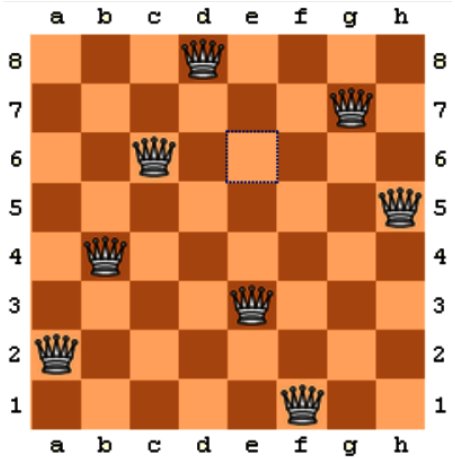
Brute Force

- Idea: search complete search space
- pro: very easy, always correct
- con: often too slow

Filter

- enumerate all possible candidates
- discard invalid ones

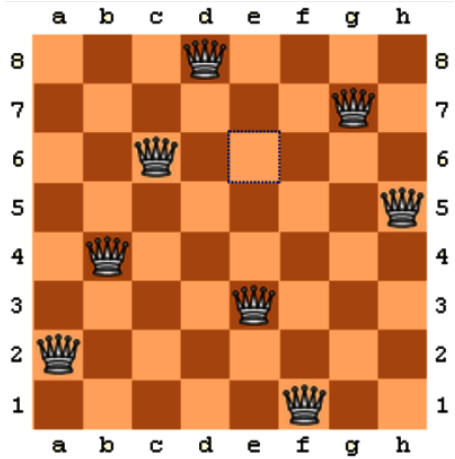
Example: Place 8 queens on a chess board s.t. they cannot capture each other.



Brute Force

- Idea: search complete search space
- pro: very easy, always correct
- con: often too slow

Example: Place 8 queens on a chess board s.t. they cannot capture each other.



Filter

- enumerate all possible candidates
- discard invalid ones

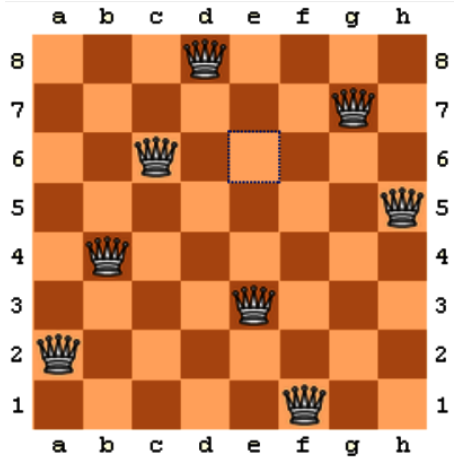
all $n!$ permuations ($n \leq 11$)
$\rightarrow$ `std::next_permutation`

all 2^n subsets ($n \leq 20$)
$\rightarrow$ bitsets

Brute Force

- Idea: search complete search space
- pro: very easy, always correct
- con: often too slow

Example: Place 8 queens on a chess board s.t. they cannot capture each other.



Filter

- enumerate all possible candidates
- discard invalid ones

all $n!$ permuations ($n \leq 11$)
$\rightarrow$ `std::next_permutation`

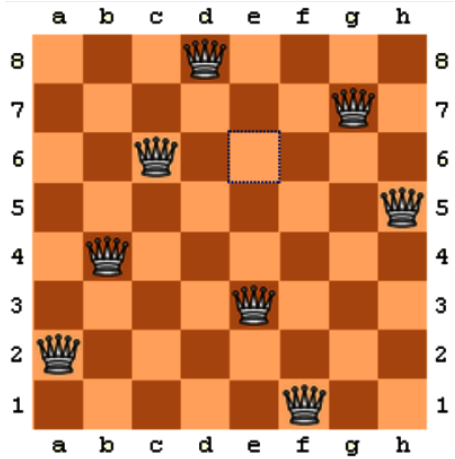
all 2^n subsets ($n \leq 20$)
$\rightarrow$ bitsets

```
for(int mask=0; mask < 1<<n; mask++)
```

Brute Force

- Idea: search complete search space
- pro: very easy, always correct
- con: often too slow

Example: Place 8 queens on a chess board s.t. they cannot capture each other.



Filter

- enumerate all possible candidates
- discard invalid ones

all $n!$ permuations ($n \leq 11$)
$\rightarrow$ `std::next_permutation`

all 2^n subsets ($n \leq 20$)
$\rightarrow$ bitsets

```
for(int mask=0; mask < 1<<n; mask++)
```

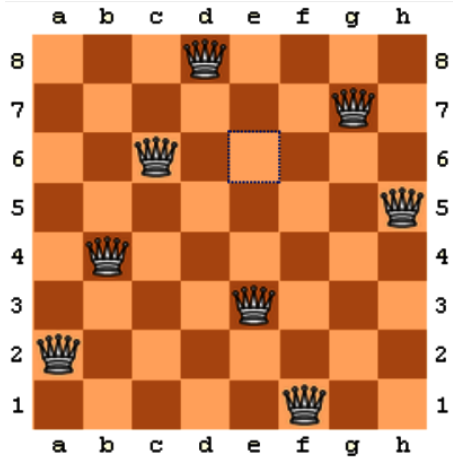
Generator

- generate only valid candidates

Brute Force

- Idea: search complete search space
- pro: very easy, always correct
- con: often too slow

Example: Place 8 queens on a chess board s.t. they cannot capture each other.



Filter

- enumerate all possible candidates
- discard invalid ones

all $n!$ permuations ($n \leq 11$)
$\rightarrow$ `std::next_permutation`

all 2^n subsets ($n \leq 20$)
$\rightarrow$ bitsets

```
for(int mask=0; mask < 1<<n; mask++)
```

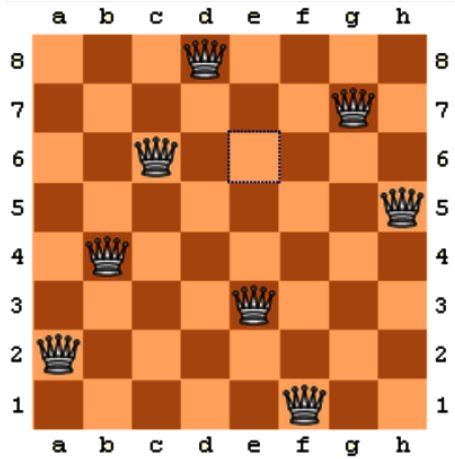
Generator

- generate only valid candidates
- usually faster than filter

Brute Force

- Idea: search complete search space
- pro: very easy, always correct
- con: often too slow

Example: Place 8 queens on a chess board s.t. they cannot capture each other.



Filter

- enumerate all possible candidates
- discard invalid ones

all $n!$ permuations ($n \leq 11$)
$\rightarrow$ `std::next_permutation`

all 2^n subsets ($n \leq 20$)
$\rightarrow$ bitsets

```
for(int mask=0; mask < 1<<n; mask++)
```

Generator

- generate only valid candidates
- usually faster than filter
- implemented recursively with backtracking + pruning

Runtime Complexity in Practice

How long will my algorithm run?

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range,
  that is $\approx 10^9$ CPU cycles per second

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range,
  that is $\approx 10^9$ CPU cycles per second
- not every instruction will take
  just one cycle

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range,
  that is $\approx 10^9$ CPU cycles per second
- not every instruction will take
  just one cycle
- CPU instructions are pipelined

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range,
  that is $\approx 10^9$ CPU cycles per second
- not every instruction will take
  just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range,
  that is $\approx 10^9$ CPU cycles per second
- not every instruction will take
  just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck
  loops does not matter

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(n)$	10^6	$\approx 10^6$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(n)$	10^6	$\approx 10^6$
$O(n^2)$	10^4	$\approx 10^8$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(n)$	10^6	$\approx 10^6$
$O(n^2)$	10^4	$\approx 10^8$
$O(n^3)$	400	$\approx 6 \cdot 10^7$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(n)$	10^6	$\approx 10^6$
$O(n^2)$	10^4	$\approx 10^8$
$O(n^3)$	400	$\approx 6 \cdot 10^7$
$O(2^n n)$	22	$\approx 9 \cdot 10^7$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(n)$	10^6	$\approx 10^6$
$O(n^2)$	10^4	$\approx 10^8$
$O(n^3)$	400	$\approx 6 \cdot 10^7$
$O(2^n n)$	22	$\approx 9 \cdot 10^7$
$O(n!)$	11	$\approx 4 \cdot 10^7$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(n)$	10^6	$\approx 10^6$
$O(n^2)$	10^4	$\approx 10^8$
$O(n^3)$	400	$\approx 6 \cdot 10^7$
$O(2^n n)$	22	$\approx 9 \cdot 10^7$
$O(n!)$	11	$\approx 4 \cdot 10^7$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(\log n)$	10^{18}	≈ 64
$O(\sqrt{n})$	10^{16}	$\approx 10^8$
$O(n)$	10^6	$\approx 10^6$
$O(n^2)$	10^4	$\approx 10^8$
$O(n^3)$	400	$\approx 6 \cdot 10^7$
$O(2^n n)$	22	$\approx 9 \cdot 10^7$
$O(n!)$	11	$\approx 4 \cdot 10^7$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(\log n)$	10^{18}	≈ 64
$O(\sqrt{n})$	10^{16}	$\approx 10^8$
$O(n)$	10^6	$\approx 10^6$
$O(n^2)$	10^4	$\approx 10^8$
$O(n^3)$	400	$\approx 6 \cdot 10^7$
$O(2^n n)$	22	$\approx 9 \cdot 10^7$
$O(n!)$	11	$\approx 4 \cdot 10^7$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Complexity	Input Size	Estimate
$O(\log n)$	10^{18}	≈ 64
$O(\sqrt{n})$	10^{16}	$\approx 10^8$
$O(n)$	10^6	$\approx 10^6$
$O(n \log n)$	$5 \cdot 10^5$	$\approx 10^7$
$O(n \log^2 n)$	$2 \cdot 10^5$	$\approx 6 \cdot 10^7$
$O(n\sqrt{n})$	$2 \cdot 10^5$	$\approx 8 \cdot 10^7$
$O(n^2)$	10^4	$\approx 10^8$
$O(n^3)$	400	$\approx 6 \cdot 10^7$
$O(2^n n)$	22	$\approx 9 \cdot 10^7$
$O(n!)$	11	$\approx 4 \cdot 10^7$

Runtime Complexity in Practice

How long will my algorithm run?

- consider a timelimit of 1s
- computers operate in GHz range, that is $\approx 10^9$ CPU cycles per second
- not every instruction will take just one cycle
- CPU instructions are pipelined
- LOC in hot loop does not matter
- number of (asymptotic) bottleneck loops does not matter

Random Fact: there are 26 lowercase latin letters

Complexity	Input Size	Estimate
$O(\log n)$	10^{18}	≈ 64
$O(\sqrt{n})$	10^{16}	$\approx 10^8$
$O(n)$	10^6	$\approx 10^6$
$O(n \log n)$	$5 \cdot 10^5$	$\approx 10^7$
$O(n \log^2 n)$	$2 \cdot 10^5$	$\approx 6 \cdot 10^7$
$O(n\sqrt{n})$	$2 \cdot 10^5$	$\approx 8 \cdot 10^7$
$O(n^2)$	10^4	$\approx 10^8$
$O(n^3)$	400	$\approx 6 \cdot 10^7$
$O(2^n n)$	22	$\approx 9 \cdot 10^7$
$O(n!)$	11	$\approx 4 \cdot 10^7$

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

Contest Checklist

Wrong Answer

- check for overflow

  `int` until 2*e*9; `long long` until 9*e*18

- check for overflow again; even `long`, `size_t` can be 32 bit

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0
- discuss solution with pair programming partner again

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0
- discuss solution with pair programming partner again
- If you did greedy, better try DP

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0
- discuss solution with pair programming partner again
- If you did greedy, better try DP

Timelimit

- ios::sync_with_stdio(false)?

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0
- discuss solution with pair programming partner again
- If you did greedy, better try DP

Timelimit

- ios::sync_with_stdio(false)?
- compute worst case number of operations in python shell

  $>\approx$500kk is too slow. You get a feel for these numbers!

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0
- discuss solution with pair programming partner again
- If you did greedy, better try DP

Timelimit

- ios::sync_with_stdio(false)?
- compute worst case number of operations in python shell

  $>\approx$500kk is too slow. You get a feel for these numbers!

- change std::endl to '\n'

  have a look at Jason Turner's
  "C++ Weekly - Ep 7 Stop Using std::endl"

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0
- discuss solution with pair programming partner again
- If you did greedy, better try DP

Timelimit

- ios::sync_with_stdio(false)?
- compute worst case number of operations in python shell

  $>\approx$500kk is too slow. You get a feel for these numbers!

- change std::endl to '\n'

  have a look at Jason Turner's
  "C++ Weekly - Ep 7 Stop Using std::endl"

- passed container by value?

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0
- discuss solution with pair programming partner again
- If you did greedy, better try DP

Timelimit

- ios::sync_with_stdio(false)?
- compute worst case number of operations in python shell

  $>\approx$500kk is too slow. You get a feel for these numbers!

- change std::endl to '\n'

  have a look at Jason Turner's
  "C++ Weekly - Ep 7 Stop Using std::endl"

- passed container by value?
- check that input and output are not interleaved; or use cin.tie(nullptr)

Contest Checklist

Wrong Answer

- check for overflow

  `int` until $2e9$; `long long` until $9e18$

- check for overflow again; even `long`, `size_t` can be 32 bit
- make sure IO is not optimized
- edge case missing? e.g. n=0
- discuss solution with pair programming partner again
- If you did greedy, better try DP

Timelimit

- ios::sync_with_stdio(false)?
- compute worst case number of operations in python shell

  $>\approx$500kk is too slow. You get a feel for these numbers!

- change std::endl to '\n'

  have a look at Jason Turner's
  "C++ Weekly - Ep 7 Stop Using std::endl"

- passed container by value?
- check that input and output are not interleaved; or use cin.tie(nullptr)
- discuss solution with pair programming partner again

C++ Materials

Reference

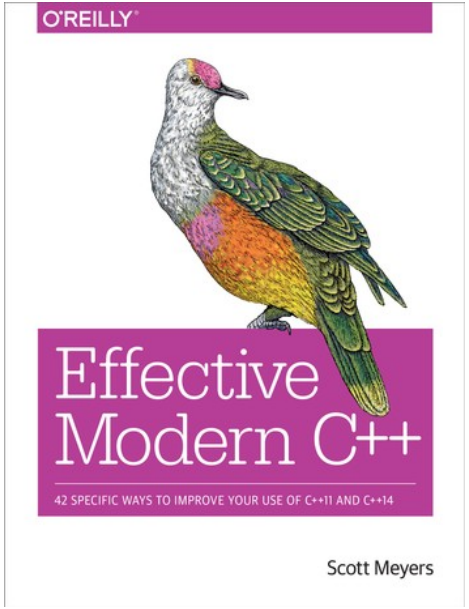
- `https://cppreference.com`

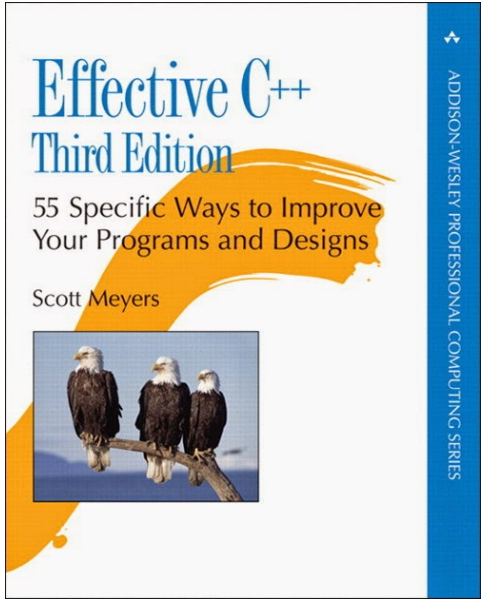
Scott Meyers Books

- Effective C++ (best one)
- Effective Modern C++
- More Effective C++
- Effective STL

Youtube Channels

- Jason Turner
- CppCon





- CppCon 2018: Jonathan Boccara “105 STL Algorithms in Less Than an Hour”
- CppCon 2017: Matt Godbolt “What Has My Compiler Done for Me Lately? Unbolting the Compiler’s Lid”

more concise C++

```
vector<int> vec;
for(int i=0; i<n; ++i) {
        int e; cin>>e;
        vec.push_back(e);
}
```

```
vector<int> vec(n,0);
for(auto& e : vec) cin>>e;
```

```
for(int i=0; i<n; ++i) {
        cout << res[i].first << ' ';
        cout << res[i].second << '\n';
}
```

```
for(auto [a,b] : res)
        cout << a << ' ' << b << '\n';
```

```
cout << "? insert 42\n";
cout.flush();
```

```
// DO NOT USE: #define endl  '\n'
cout << "? insert 42" << endl;
```

```
struct cmp {
bool operator()(int a, int b) { /*TODO*/ }
};
sort(vec.begin(), vec.end(), cmp{})
```

```
#define all(a)      (a).begin(),(a).end()
sort(all(vec), [](int a, int b) {
        /*TODO*/
});
```

```
int a = get<0>(pq.top());
int b = get<1>(pq.top());
int c = get<2>(pq.top());
```

```
auto [a,b,c] = pq.top();
```

Rust Materials

References

- the rust std docs
  `https://doc.rust-lang.org/std/`
- the rust book
  `https://doc.rust-lang.org/book/`

Learning Materials

- rust by example
  `https://doc.rust-lang.org/rust-by-example/`
- interactive rustlings tutorial
  `https://github.com/rust-lang/rustlings`