ITMO UNIVERSITY

How to Win Coding Competitions: Secrets of Champions

Week 3: Sorting and Search Algorithms
Lecture 7: Lower bound. Stable sorting. Comparators

Maxim Buzdalov
Saint Petersburg 2016

Lower bound for comparison sorting

Theorem. For every comparison-based sorting algorithm, there exists an input with N elements which requires at least $b = \Omega(N \log N)$ comparisons.

Theorem. For every comparison-based sorting algorithm, there exists an input with N elements which requires at least $b = \Omega(N \log N)$ comparisons.

Proof

- Assume inputs are permutations. There are $N!$ different permutations

Lower bound for comparison sorting

Theorem. For every comparison-based sorting algorithm, there exists an input with N elements which requires at least $b = \Omega(N \log N)$ comparisons.

Proof

- Assume inputs are permutations. There are $N!$ different permutations
- From a single comparison, any algorithm receives at most one bit of information

Lower bound for comparison sorting

Theorem. For every comparison-based sorting algorithm, there exists an input with N elements which requires at least $b = \Omega(N \log N)$ comparisons.

Proof

- Assume inputs are permutations. There are $N!$ different permutations
- From a single comparison, any algorithm receives at most one bit of information
- After making t comparisons, an algorithm can distinguish only 2^t cases

Lower bound for comparison sorting

Theorem. For every comparison-based sorting algorithm, there exists an input with N elements which requires at least $b = \Omega(N \log N)$ comparisons.

Proof

- Assume inputs are permutations. There are $N!$ different permutations
- From a single comparison, any algorithm receives at most one bit of information
- After making t comparisons, an algorithm can distinguish only 2^t cases
- Thus, for the lower bound b, it holds that $2^b \geq N!$, or $b \geq \log_2(N!)$

Lower bound for comparison sorting

Theorem. For every comparison-based sorting algorithm, there exists an input with N elements which requires at least $b = \Omega(N \log N)$ comparisons.

Proof

- Assume inputs are permutations. There are $N!$ different permutations
- From a single comparison, any algorithm receives at most one bit of information
- After making t comparisons, an algorithm can distinguish only 2^t cases
- Thus, for the lower bound b, it holds that $2^b \geq N!$, or $b \geq \log_2(N!)$
- By Stirling's approximation:

$$\log_2(N!) \approx \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = n \log_2 n - \Theta(n) + O(\log n) = \Theta(n \log n)$$

Lower bound for comparison sorting

Theorem. For every comparison-based sorting algorithm, there exists an input with N elements which requires at least $b = \Omega(N \log N)$ comparisons.

Proof

- Assume inputs are permutations. There are $N!$ different permutations
- From a single comparison, any algorithm receives at most one bit of information
- After making t comparisons, an algorithm can distinguish only 2^t cases
- Thus, for the lower bound b, it holds that $2^b \geq N!$, or $b \geq \log_2(N!)$
- By Stirling's approximation:

$$\log_2(N!) \approx \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = n\log_2 n - \Theta(n) + O(\log n) = \Theta(n \log n)$$

Mergesort is asymptotically optimal.

Lower bound for comparison sorting

Theorem. For every comparison-based sorting algorithm, there exists an input with N elements which requires at least $b = \Omega(N \log N)$ comparisons.

Proof

- Assume inputs are permutations. There are $N!$ different permutations
- From a single comparison, any algorithm receives at most one bit of information
- After making t comparisons, an algorithm can distinguish only 2^t cases
- Thus, for the lower bound b, it holds that $2^b \geq N!$, or $b \geq \log_2(N!)$
- By Stirling's approximation:

$$\log_2(N!) \approx \log_2\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right) = n\log_2 n - \Theta(n) + O(\log n) = \Theta(n\log n)$$

Mergesort is asymptotically optimal.
Quicksort is asymptotically optimal on average.

Stable sorting I

What is stable sorting?

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
  - Assume elements are already ordered by $\preceq_1$
  - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
  - Assume elements are already ordered by $\preceq_1$
  - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
  - Solution: just use a stable sorting

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
  - Assume elements are already ordered by $\preceq_1$
  - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
  - Solution: just use a stable sorting

Which sorting algorithms are stable?

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
  - Assume elements are already ordered by $\preceq_1$
  - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
  - Solution: just use a stable sorting

Which sorting algorithms are stable?

- Insertion sort:

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
    - Assume elements are already ordered by $\preceq_1$
    - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
    - Solution: just use a stable sorting

Which sorting algorithms are stable?

- Insertion sort: stable, as it never swaps $A[i]$ and $A[i+1]$ if they are equal

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
  - Assume elements are already ordered by $\preceq_1$
  - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
  - Solution: just use a stable sorting

Which sorting algorithms are stable?

- Insertion sort: stable, as it never swaps $A[i]$ and $A[i+1]$ if they are equal
- Mergesort:

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
  - Assume elements are already ordered by $\preceq_1$
  - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
  - Solution: just use a stable sorting

Which sorting algorithms are stable?

- Insertion sort: stable, as it never swaps $A[i]$ and $A[i+1]$ if they are equal
- Mergesort: stable, as merge always picks the left element from two equal ones

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
    - Assume elements are already ordered by $\preceq_1$
    - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
    - Solution: just use a stable sorting

Which sorting algorithms are stable?

- Insertion sort: stable, as it never swaps $A[i]$ and $A[i+1]$ if they are equal
- Mergesort: stable, as merge always picks the left element from two equal ones
- Quicksort:

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
  - Assume elements are already ordered by $\preceq_1$
  - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
  - Solution: just use a stable sorting

Which sorting algorithms are stable?

- Insertion sort: stable, as it never swaps $A[i]$ and $A[i+1]$ if they are equal
- Mergesort: stable, as merge always picks the left element from two equal ones
- Quicksort: not stable.

Stable sorting I

What is stable sorting?

- A sorting algorithm is stable if it preserves the order of equal elements
- Why is it useful?
  - Assume elements are already ordered by $\preceq_1$
  - You wish to sort them by $\preceq_2$, but keep equal elements ordered by $\preceq_1$
  - Solution: just use a stable sorting

Which sorting algorithms are stable?

- Insertion sort: stable, as it never swaps $A[i]$ and $A[i+1]$ if they are equal
- Mergesort: stable, as merge always picks the left element from two equal ones
- Quicksort: not stable. In an array of two equal items, it will swap them

Stable sorting II

How to make every sorting stable?

Stable sorting II

How to make every sorting stable?

- Keep original indices with the elements

Stable sorting II

How to make every sorting stable?

- Keep original indices with the elements
  - Before: 5 2 4 4 5 2
  - After: $\langle 5; 1\rangle$ $\langle 2; 2\rangle$ $\langle 4; 3\rangle$ $\langle 4; 4\rangle$ $\langle 5; 5\rangle$ $\langle 2; 6\rangle$

Stable sorting II

How to make every sorting stable?

- Keep original indices with the elements
  - Before: 5 2 4 4 5 2
  - After: $\langle 5;1\rangle$ $\langle 2;2\rangle$ $\langle 4;3\rangle$ $\langle 4;4\rangle$ $\langle 5;5\rangle$ $\langle 2;6\rangle$
- In the sorting, use a modified ordering $\preceq'$:
  - Given $\langle x;a\rangle$ and $\langle y;b\rangle$
  - If $x \ne y$, return $x \preceq y$
  - If $x = y$, return $a \le b$

Stable sorting II

How to make every sorting stable?

- Keep original indices with the elements
  - Before: $5 \quad 2 \quad 4 \quad 4 \quad 5 \quad 2$
  - After: $\langle 5; 1\rangle\ \langle 2; 2\rangle\ \langle 4; 3\rangle\ \langle 4; 4\rangle\ \langle 5; 5\rangle\ \langle 2; 6\rangle$
- In the sorting, use a modified ordering $\preceq'$:
  - Given $\langle x; a\rangle$ and $\langle y; b\rangle$
  - If $x \ne y$, return $x \preceq y$ ← original ordering here
  - If $x = y$, return $a \le b$

Stable sorting II

How to make every sorting stable?

- Keep original indices with the elements
    - Before: 5 2 4 4 5 2
    - After: $\langle 5;1\rangle$ $\langle 2;2\rangle$ $\langle 4;3\rangle$ $\langle 4;4\rangle$ $\langle 5;5\rangle$ $\langle 2;6\rangle$
- In the sorting, use a modified ordering $\preceq'$:
    - Given $\langle x;a\rangle$ and $\langle y;b\rangle$
    - If $x \neq y$, return $x \preceq y$ ← original ordering here
    - If $x = y$, return $a \leq b$

Pros:

- Can use any sorting, don't need to care of implementation

Stable sorting II

How to make every sorting stable?

- Keep original indices with the elements
  - Before: 5 2 4 4 5 2
  - After: $\langle 5;1\rangle$ $\langle 2;2\rangle$ $\langle 4;3\rangle$ $\langle 4;4\rangle$ $\langle 5;5\rangle$ $\langle 2;6\rangle$
- In the sorting, use a modified ordering $\preceq'$:
  - Given $\langle x;a\rangle$ and $\langle y;b\rangle$
  - If $x \neq y$, return $x \preceq y$ ← original ordering here
  - If $x = y$, return $a \leq b$

Pros:

- Can use any sorting, don't need to care of implementation

Cons:

- Additional memory needed for storing the indices
- More complicated ordering, may further decrease performance

A comparator (as in `java.util.Comparator<T>`)
is a custom ordering for sorting certain objects for specific needs

Comparators

A comparator (as in `java.util.Comparator<T>`)
is a custom ordering for sorting certain objects for specific needs

Requirements for a comparator:

- Total (every two elements should be reported as either “$<$”, “$=$”, or “$>$”)
- If it reports $a = b$, then a and b must be equal (in a problem-dependent sense)
- More specifically, if $a = b$, then $b = a$
- If $a < b$, then it should be that $b > a$ (and vice versa)

Comparators

A comparator (as in `java.util.Comparator<T>`)
is a custom ordering for sorting certain objects for specific needs

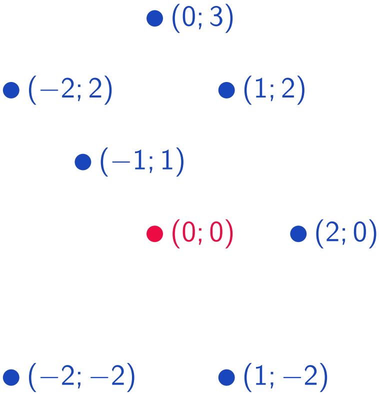
Requirements for a comparator:

- Total (every two elements should be reported as either "$<$", "$=$", or "$>$")
- If it reports $a = b$, then a and b must be equal (in a problem-dependent sense)
- More specifically, if $a = b$, then $b = a$
- If $a < b$, then it should be that $b > a$ (and vice versa)

Implementation of a comparator:

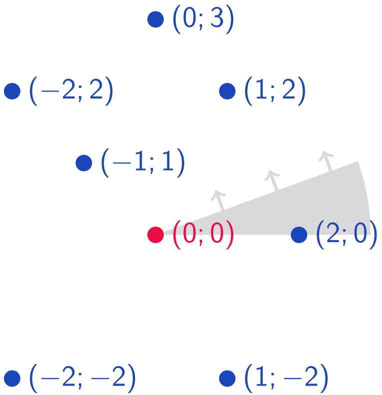
- `Java`-way: return 0 if "equal", negative if "less", positive if "greater"
- `C++`-way: return `true` if "less", `false` otherwise

A non-trivial comparator



Sort the points counterclockwise around the red dot

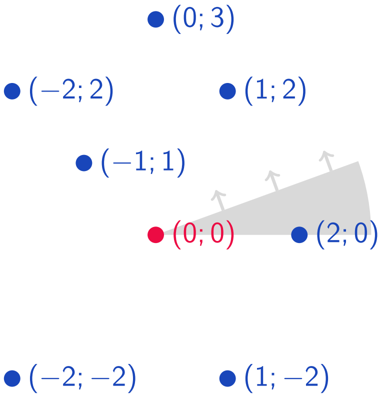
A non-trivial comparator



Sort the points counterclockwise around the red dot

- Need to break a circle: start at $\overrightarrow{Ox}$

A non-trivial comparator

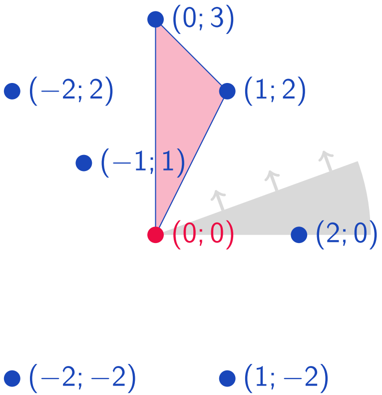


Sort the points counterclockwise around the red dot

- Need to break a circle: start at $\overrightarrow{Ox}$

Idea: compare using oriented triangle area

A non-trivial comparator



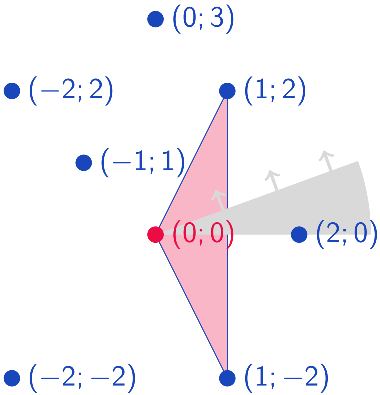
Sort the points counterclockwise around the red dot

- Need to break a circle: start at $\overrightarrow{Ox}$

Idea: compare using oriented triangle area

```
function LessThan($(x_1, y_1)$, $(x_2, y_2)$)
    return $x_1 \cdot y_2 - x_2 \cdot y_1 > 0$
end function
```

A non-trivial comparator



Sort the points counterclockwise around the red dot

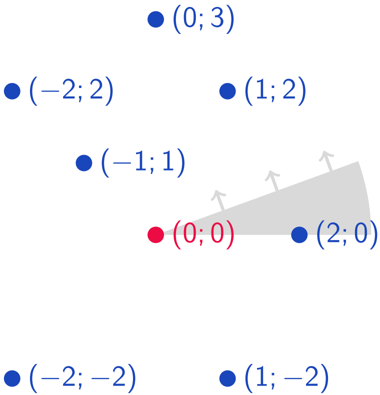
- Need to break a circle: start at $\overrightarrow{Ox}$

Idea: compare using oriented triangle area

Wait, what happens there?

```
function LessThan($(x_1, y_1), (x_2, y_2)$)
    return $x_1 \cdot y_2 - x_2 \cdot y_1 > 0$
end function
```

A non-trivial comparator



Sort the points counterclockwise around the red dot

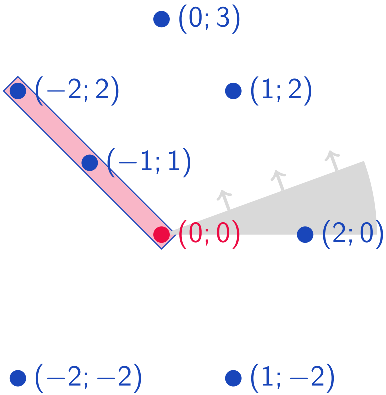
- Need to break a circle: start at $\overrightarrow{Ox}$

Idea: compare using oriented triangle area

…but first check the halfplanes

```
function LessThan($(x_1, y_1)$, $(x_2, y_2)$)
    $h_1 \leftarrow 0$, $h_2 \leftarrow 0$
    if $y_1 < 0$ or ($y_1 = 0$ and $x_1 < 0$) then $h_1 \leftarrow 1$ end if
    if $y_2 < 0$ or ($y_2 = 0$ and $x_2 < 0$) then $h_2 \leftarrow 1$ end if
    if $h_1 \neq h_2$ then return $h_1 < h_2$ end if
    return $x_1 \cdot y_2 - x_2 \cdot y_1 > 0$
end function
```

A non-trivial comparator



Sort the points counterclockwise around the red dot

- Need to break a circle: start at $\overrightarrow{Ox}$

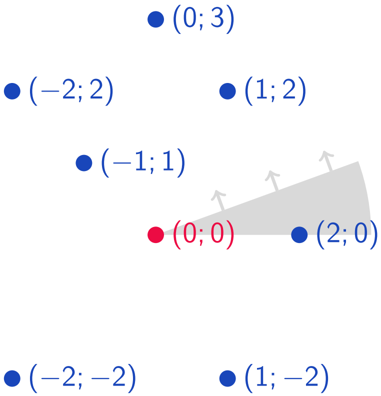
Idea: compare using oriented triangle area

…but first check the halfplanes

Okay, but what happens for colinear points?

```
function LessThan($(x_1, y_1)$, $(x_2, y_2)$)
    $h_1 \leftarrow 0$, $h_2 \leftarrow 0$
    if $y_1 < 0$ or ($y_1 = 0$ and $x_1 < 0$) then $h_1 \leftarrow 1$ end if
    if $y_2 < 0$ or ($y_2 = 0$ and $x_2 < 0$) then $h_2 \leftarrow 1$ end if
    if $h_1 \neq h_2$ then return $h_1 < h_2$ end if
    return $x_1 \cdot y_2 - x_2 \cdot y_1 > 0$
end function
```

A non-trivial comparator



Sort the points counterclockwise around the red dot

- Need to break a circle: start at $\overrightarrow{Ox}$

Idea: compare using oriented triangle area

. . . but first check the halfplanes

. . . and, from colinear points, favor closer ones.

```
function LessThan($(x_1, y_1)$, $(x_2, y_2)$)
    $h_1 \leftarrow 0$, $h_2 \leftarrow 0$
    if $y_1 < 0$ or ($y_1 = 0$ and $x_1 < 0$) then $h_1 \leftarrow 1$ end if
    if $y_2 < 0$ or ($y_2 = 0$ and $x_2 < 0$) then $h_2 \leftarrow 1$ end if
    if $h_1 \neq h_2$ then return $h_1 < h_2$ end if
    $z \leftarrow x_1 \cdot y_2 - x_2 \cdot y_1$
    if $z = 0$ then return $x_1^2 + y_1^2 < x_2^2 + y_2^2$ end if
    return $z > 0$
end function
```