CS 577- Intro to Algorithms

Divide and Conquer

Dieter van Melkebeek

September 10, 2020

Outline

Paradigm

1. Break up given instance into smaller ones.
2. Recursively solve those.
3. Combine their solutions into one for the given instance.

Outline

Paradigm

1. Break up given instance into smaller ones.
2. Recursively solve those.
3. Combine their solutions into one for the given instance.

Today

- Common pattern
  - Sorting (Mergesort)
  - Counting inversions

Outline

Paradigm

1. Break up given instance into smaller ones.
2. Recursively solve those.
3. Combine their solutions into one for the given instance.

Today

- Common pattern
  - Sorting (Mergesort)
  - Counting inversions
- Lower bound for sorting

Problem vs Algorithm vs Program

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.
- Precise (mathematical) description that stands on its own.

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.
- Precise (mathematical) description that stands on its own.
- Example: Powering

  Input: (a, b) with $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$

  Output: a^b

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.
- Precise (mathematical) description that stands on its own.
- Example: Powering

  Input: (a, b) with $a \in \mathbb{Z}$ and $b \in \mathbb{Z}^+$

  Output: a^b
- Example: Sorting

  Input: An array $A[1, \dots, n]$ of length $n \geq 1$.

  Output: Sort(A), i.e., a copy of A sorted from the smallest to the largest element

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.
- Precise (mathematical) description that stands on its own.

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.
- Precise (mathematical) description that stands on its own.

Algorithm

- *How* to realize the transformation.

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.
- Precise (mathematical) description that stands on its own.

Algorithm

- *How* to realize the transformation.
- Precise high-level description in words and/or pseudocode.

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.
- Precise (mathematical) description that stands on its own.

Algorithm

- *How* to realize the transformation.
- Precise high-level description in words and/or pseudocode.
- Examples for Sorting: Selection Sort, Insertion Sort, Bubblesort, Quicksort, Merge Sort, . . .

Problem vs Algorithm vs Program

Problem (specification, method)

- *What* transformation from inputs to outputs to realize.
- Precise (mathematical) description that stands on its own.

Algorithm

- *How* to realize the transformation.
- Precise high-level description in words and/or pseudocode.
- Examples for Sorting: Selection Sort, Insertion Sort, Bubblesort, Quicksort, Merge Sort, ...

Program (code, implementation)

- Precise detailed description in a particular programming language

Mergesort in words

Mergesort in pseudocode

Input: $A[1, \dots, n]$
Output: $\mathrm{Sort}(A)$

procedure $\textsc{Merge-Sort}(A)$
    if $n = 1$ **then**
        return A
    else
        $m \leftarrow \lfloor n/2 \rfloor$
        $L \leftarrow \textsc{Merge-Sort}(A[1, \dots, m])$
        $R \leftarrow \textsc{Merge-Sort}(A[m+1, \dots, n])$
        return $\textsc{Merge}(L, R)$

where Merge

Input: sorted arrays $L[1, \dots, n]$ and $R[1, \dots, m]$ with $n, m \geq 1$.

Output: $\mathrm{Sort}(LR)$

Mergesort – correctness

Mergesort – correctness

Meaning

On every valid input, the algorithm produces a correct output.

Mergesort – correctness

Meaning

On every valid input, the algorithm produces a correct output.

Theorem

For every integer $n \geq 1$, $P(n)$ holds, where
$P(n)$: On every array $A[1, \ldots, n]$, Merg-Sort(A) returns Sort(A).

Mergesort – correctness

Meaning

On every valid input, the algorithm produces a correct output.

Theorem

For every integer $n \geq 1$, $P(n)$ holds, where
$P(n)$: On every array $A[1, \ldots, n]$, Merg-Sort(A) returns Sort(A).

Proof

- Base case: $n = 1$

Mergesort – correctness

Meaning

On every valid input, the algorithm produces a correct output.

Theorem

For every integer $n \geq 1$, $P(n)$ holds, where
$P(n)$: On every array $A[1, \dots, n]$, Merg-Sort(A) returns Sort(A).

Proof

- Base case: $n = 1$
- Induction step:
  - For every integer $n \geq 1$, $P(n) \Rightarrow P(n+1)$

Mergesort – correctness

Meaning

On every valid input, the algorithm produces a correct output.

Theorem

For every integer $n \geq 1$, $P(n)$ holds, where
$P(n)$: On every array $A[1, \dots, n]$, Merg-Sort(A) returns Sort(A).

Proof

- Base case: $n = 1$
- Induction step:
  - For every integer $n \geq 1$, $P(n) \Rightarrow P(n+1)$
  - For every integer $n \geq 1$, $P(1) \wedge P(2) \dots P(n) \Rightarrow P(n+1)$

Merging Sorted Arrays

Merging Sorted Arrays

- PowerPoint presentation
- Correctness
- Running time: $O(n + m)$

Mergesort – running time

Mergesort – running time

- Recurrence relation

Mergesort – running time

- Recurrence relation
- Recursion tree

Counting Inversions

Counting Inversions

Definition

An inversion in an array $A[1, \dots, n]$ is a pair $(i, j) \in [n] \times [n]$ with $i < j$ and $A[i] > A[j]$.

Counting Inversions

Definition

An inversion in an array $A[1, \ldots, n]$ is a pair $(i, j) \in [n] \times [n]$ with $i < j$ and $A[i] > A[j]$.

Example

$A = [3, 5, 4, 7, 3, 1]$

Counting Inversions

Definition

An inversion in an array $A[1, \ldots, n]$ is a pair $(i, j) \in [n] \times [n]$ with $i < j$ and $A[i] > A[j]$.

Example

$A = [3, 5, 4, 7, 3, 1]$

Problem

Input: array $A[1, \ldots, n]$ of integers with $n \geq 1$

Output: $\mathrm{Inv}(A) \doteq$ number of inversions in A

Counting Inversions

Definition

An inversion in an array $A[1, \ldots, n]$ is a pair $(i, j) \in [n] \times [n]$ with $i < j$ and $A[i] > A[j]$.

Example

$A = [3, 5, 4, 7, 3, 1]$

Problem

Input: array $A[1, \ldots, n]$ of integers with $n \geq 1$

Output: $\text{Inv}(A) \doteq$ number of inversions in A

Bounds on $\text{Inv}(A)$

Count in words

Count in pseudocode

Input: $A[1, \ldots, n]$
Output: $\mathrm{Sort}(A)$

procedure $\textsc{Merge-Sort}(A)$
    if $n = 1$ **then**
        return A
    else
        $m \leftarrow \lfloor n/2 \rfloor$
        $L \leftarrow \textsc{Merge-Sort}(A[1, \ldots, m])$
        $R \leftarrow \textsc{Merge-Sort}(A[m+1, \ldots, n])$
        return $\textsc{Merge}(L, R)$

where Count-Cross

Input: sorted arrays $L[1, \ldots, n]$ and $R[1, \ldots, m]$ with $n, m \geq 1$.

Output: $\mathrm{Inv}(LR)$

Count – running time

Improved Count

Input: $A[1 \cdots n]$, an array of length $n \geq 1$
Output: (Inv(A),Sort(A))

procedure COUNT-AND-SORT(A)
    if $n = 1$ **then**
        return $(0, A)$
    else
        $m \leftarrow \lfloor n/2 \rfloor$
        $(c_L, L) \leftarrow$ COUNT-AND-SORT($A[1, \ldots, m]$)
        $(c_R, R) \leftarrow$ COUNT-AND-SORT($A[m+1, \ldots, n]$)
        $c_{\text{cross}} \leftarrow$ COUNT-CROSS(L, R)
        $c \leftarrow c_L + c_R + c_{\text{cross}}$
        $B \leftarrow$ MERGE(L, R)
        return $(c_L + c_R + c_{\text{cross}}, B)$

Improved Count

Input: $A[1 \cdots n]$, an array of length $n \geq 1$
Output: $(\mathrm{Inv}(A), \mathrm{Sort}(A))$

procedure $\textsc{Count-And-Sort}(A)$
    if $n = 1$ **then**
        return $(0, A)$
    else
        $m \leftarrow \lfloor n/2 \rfloor$
        $(c_L, L) \leftarrow \textsc{Count-And-Sort}(A[1, \ldots, m])$
        $(c_R, R) \leftarrow \textsc{Count-And-Sort}(A[m+1, \ldots, n])$
        $c_{\text{cross}} \leftarrow \textsc{Count-Cross}(L, R)$
        $c \leftarrow c_L + c_R + c_{\text{cross}}$
        $B \leftarrow \textsc{Merge}(L, R)$
        return $(c_L + c_R + c_{\text{cross}}, B)$

Running time: $O(n \log n)$

Sorting Lower Bound

Sorting Lower Bound

Theorem

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ comparisons on arrays of length n.

Sorting Lower Bound

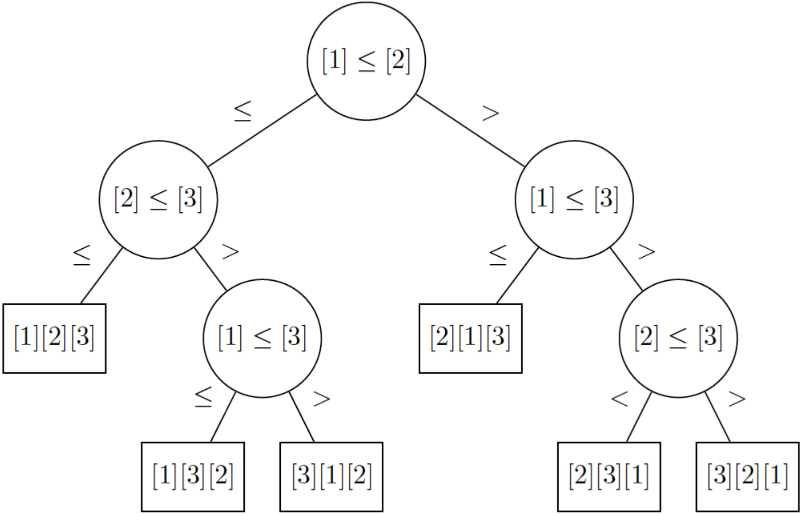
Theorem

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ comparisons on arrays of length n.

Proof

- Every such algorithm can be modeled as a decision tree.

Decision Tree



Sorting Lower Bound

Sorting Lower Bound

Theorem

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ comparisons on arrays of length n.

Proof

- Every such algorithm for a given n can be modeled as a binary decision tree T.
- Depth d of T is the maximum number c of comparisons that A makes on arrays of length n.
- Number ℓ of leaves is at least $n! \doteq 1 \cdot 2 \cdot \ldots \cdot n$.
- $\ell \geq 2^d$

Sorting Lower Bound

Theorem

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ comparisons on arrays of length n.

Proof

- Every such algorithm for a given n can be modeled as a binary decision tree T.
- Depth d of T is the maximum number c of comparisons that A makes on arrays of length n.
- Number ℓ of leaves is at least $n! \doteq 1 \cdot 2 \cdot \ldots \cdot n$.
- $\ell \geq 2^d$
- $c = d \geq \log(\ell) \geq \log(n!)$

Sorting Lower Bound

Theorem

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ comparisons on arrays of length n.

Proof

- Every such algorithm for a given n can be modeled as a binary decision tree T.
- Depth d of T is the maximum number c of comparisons that A makes on arrays of length n.
- Number ℓ of leaves is at least $n! \doteq 1 \cdot 2 \cdot \ldots \cdot n$.
- $\ell \geq 2^d$
- $c = d \geq \log(\ell) \geq \log(n!)$
- $(n/2)^{n/2} \leq n! \leq n^n$

Sorting Lower Bound

Theorem

Every comparison-based sorting algorithm takes $\Omega(n \log n)$ comparisons on arrays of length n.

Proof

- Every such algorithm for a given n can be modeled as a binary decision tree T.
- Depth d of T is the maximum number c of comparisons that A makes on arrays of length n.
- Number ℓ of leaves is at least $n! \doteq 1 \cdot 2 \cdot \ldots \cdot n$.
- $\ell \geq 2^d$
- $c = d \geq \log(\ell) \geq \log(n!)$
- $(n/2)^{n/2} \leq n! \leq n^n$
- $\log(n!) = \Theta(n \log n)$