

CS 577 - Divide and Conquer Sort & Search

Manolis Vlatakis

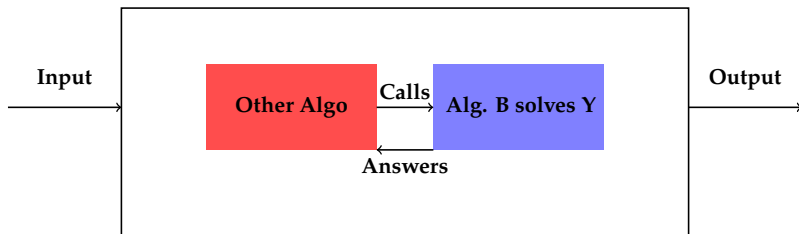
Department of Computer Sciences
University of Wisconsin – Madison

Fall 2024

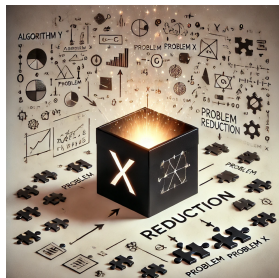


IT'S ALL ABOUT REDUCTION

REDUCTION IS THE SINGLE MOST COMMON TECHNIQUE USED IN DESIGNING ALGORITHMS.



Alg. A solves X



Reducing one problem X to another problem Y means :
Writing an algorithm for X that uses an algorithm for Y as a black box or subroutine.

IT'S ALL ABOUT REDUCTION

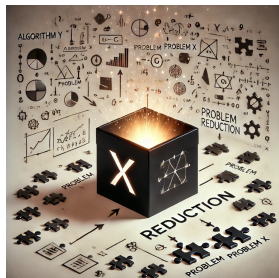
REDUCTION IS THE SINGLE MOST COMMON TECHNIQUE USED IN DESIGNING ALGORITHMS.

Input

Output

🤖 What is an example of an algorithm that uses reduction, which you learned in high school?

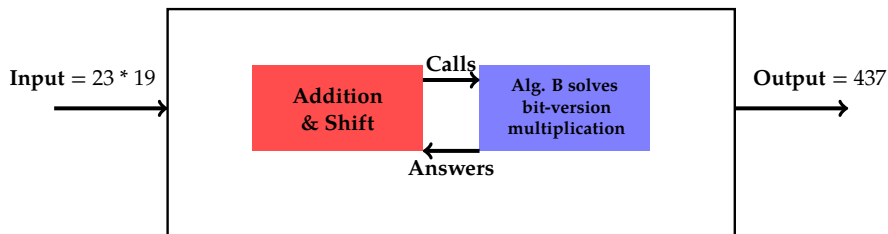
Alg. A solves X



Reducing one problem X to another problem Y means :
Writing an algorithm for X that uses an algorithm for Y as a black box or subroutine.

IT'S ALL ABOUT REDUCTION

INTEGER MULTIPLICATION IS REDUCED TO BIT MULTIPLICATION.



Alg. A solves Integer Multiplication

To compute 23×19 we need $\{2 \cdot 3, 2 \cdot 9, 3 \cdot 1, 3 \cdot 9\}$

DIVIDE AND CONQUER

DIVIDE AND CONQUER (DC)

Overview

- Split problem into smaller sub-problems.
- Solve (usually recurse on) the smaller sub-problems.
- Use the output from the smaller sub-problems to build the solution.

DIVIDE AND CONQUER (DC)

Overview

- Split problem into smaller sub-problems.
- Solve (usually recurse on) the smaller sub-problems.
- Use the output from the smaller sub-problems to build the solution.

Tendencies of DC

- Naturally recursive solutions
- Solving complexities often involve recurrences.
- Often used to improve efficiency of efficient solutions, e.g.

$$O(n^2) \rightarrow O(n \log n)$$

- Used in conjunction with other techniques.

DIVIDE AND CONQUER (DC)

Overview

- Split problem into smaller independent sub-problems.
- Solve (usually recurse on) the smaller sub-problems.
- Use the output from the smaller sub-problems to build the solution.

Tendencies of DC

- Naturally recursive solutions
- Solving complexities often involve recurrences.
- Often used to improve efficiency of efficient solutions, e.g.

$$O(n^2) \rightarrow O(n \log n)$$

- Used in conjunction with other techniques.

SORT

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- 🤖 What is the time complexity?

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- 🤖 What is the time complexity?

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts [Today & Next time]

- Divide & Conquer:
Quick Sort * Merge Sort

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts [Today & Next time]

- Divide & Conquer:
Quick Sort * Merge Sort
- 🤖 What is the time complexity of Quick/Merge Sort?

SORTING

Ordering some (multi)set of n items.

Brute Force

- Test all possible orderings.
- $O(n \cdot n!)$

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts [Today & Next time]

- Divide & Conquer:
Quick Sort ($O(n^2)$) * Merge Sort ($O(n \log n)$)

SORTING

Ordering some (multi)set of n items.

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts [Today & Next time]

- Divide & Conquer:

Quick Sort ($O(n^2)$) * Merge Sort ($O(n \log n)$)

Trick Sorts [Homework]

- Radix Sort ($O(n \lceil \log k \rceil)$), Counting Sort ($O(n + k)$)
- k is the maximum key size.

SORTING

Ordering some (multi)set of n items.

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts [Today & Next time]

- Divide & Conquer:

Quick Sort ($O(n^2)$) * Merge Sort ($O(n \log n)$)

Trick Sorts [Homework]

- Radix Sort ($O(n \lceil \log k \rceil)$), Counting Sort ($O(n + k)$)
- k is the maximum key size.
- 🤖 What value of k would make both sorts have time complexity no better than Merge Sort?

SORTING

Ordering some (multi)set of n items.

Simple Sorts

- Insertion Sort, Selection Sort, Bubble Sort
- $O(n^2)$

Efficient Sorts [Today & Next time]

- Divide & Conquer:

Quick Sort ($O(n^2)$) * Merge Sort ($O(n \log n)$)

Trick Sorts [Homework]

- Radix Sort ($O(n \lceil \log k \rceil)$), Counting Sort ($O(n + k)$)
- k is the maximum key size.
- 🤖 What value of k would make both sorts have time complexity no better than Merge Sort? $\Omega(n \log n)$

MERGESORT

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 :=$ MERGESORT(Front-half of A)

$A_2 :=$ MERGESORT(Back-half of A)

return $\text{MERGE}(A_1, A_2)$

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

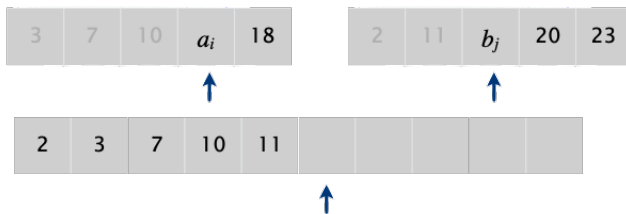
 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

🤖 What is the complexity of MERGE?

SUBROUTINE: MERGE



MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

🤖 What is the complexity of MERGE? $O(n)$

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 :=$ MERGESORT(Front-half of A)

$A_2 :=$ MERGESORT(Back-half of A)

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 :=$ MERGESORT(Front-half of A)

$A_2 :=$ MERGESORT(Back-half of A)

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

- 1 Soundness:

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

- 1 Soundness: List A is sorted after call to MERGESORT.

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

- 1 Soundness: List A is sorted after call to MERGESORT.
Proof:

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 :=$ MERGESORT(Front-half of A)

$A_2 :=$ MERGESORT(Back-half of A)

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

- 1 Soundness: List A is sorted after call to MERGESORT.
Proof: By strong induction on list length:

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

- ① Soundness: List A is sorted after call to MERGESORT.

Proof: By strong induction on list length:

Base case: $k = 1$: List is sorted.

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

- ① **Soundness:** List A is sorted after call to MERGESORT.

Proof: By strong induction on list length:

Base case: $k = 1$: List is sorted.

Inductive step: By ind hyp, A_1 and A_2 are sorted, and, then, by definition, MERGE will produce a sorted list.

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

- 1 Soundness: List A is sorted after call to MERGESORT.

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Program Correctness:

- 1 Soundness: List A is sorted after call to MERGESORT.
- 2 Complete: Handles lists of any size, and each recursion makes progress towards base case by splitting the list in half.

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

① Soundness:

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

- 1 **Soundness**: By strong induction, after k iterations, the list S is correctly sorted up to the k -th element.
 - **Base Case**: For $k = 1$, S contains the smallest element from $A[1]$ and $B[1]$, which is correctly placed.

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

- 1 **Soundness**: By strong induction, after k iterations, the list S is correctly sorted up to the k -th element.
 - **Base Case**: For $k = 1$, S contains the smallest element from $A[1]$ and $B[1]$, which is correctly placed.
 - **Inductive Step**: Assume the list S is correctly sorted up to k elements. In the $(k + 1)$ -th iteration, MERGE adds the smallest element from A or B that hasn't been added yet, ensuring that S remains sorted.

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

- 1 **Soundness**: By strong induction, after k iterations, the list S is correctly sorted up to the k -th element.
 - **Base Case**: For $k = 1$, S contains the smallest element from $A[1]$ and $B[1]$, which is correctly placed.
 - **Inductive Step**: Assume the list S is correctly sorted up to k elements. In the $(k + 1)$ -th iteration, MERGE adds the smallest element from A or B that hasn't been added yet, ensuring that S remains sorted. **Why is it sufficient to check the front of A & B ?**

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

- 1 **Soundness:** By strong induction, after k iterations, the list S is correctly sorted up to the k -th element.
- 2 **Completeness:**

MERGE

Algorithm: MERGE

Input : Two lists of comparable items: A and B .

Output: A merged list.

Initialize S to an empty list.

while either A or B is not empty **do**

 | Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

end

return S

Correctness of MERGE:

- 1 **Soundness**: By strong induction, after k iterations, the list S is correctly sorted up to the k -th element.
- 2 **Completeness**: The process continues until all elements from both A and B have been merged into S , ensuring no element is missed.

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ **then return** A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Run time Considerations:

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ then return A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Run time Considerations:

- Cost to MERGE: $O(n)$.

MERGESORT

Algorithm: MERGESORT

Input : A list A of n comparable items.

Output: A sorted list A .

if $|A| = 1$ then return A

$A_1 := \text{MERGESORT}(\text{Front-half of } A)$

$A_2 := \text{MERGESORT}(\text{Back-half of } A)$

return $\text{MERGE}(A_1, A_2)$

Run time Considerations:

- Cost to MERGE: $O(n)$.
- Recurrences: 2 calls to MERGESORT with lists half the size.

MERGESORT RECURRENCE

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

MERGESORT RECURRENCE

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Notes

- More precise: $T(n) \leq T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + cn$
- Usually, we can asymptotically ignore floor and ceilings.
- Essentially, we are assuming n is a power of 2.
- Alternate form: $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n); T(1) \leq O(1)$

MERGESORT RECURRENCE

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Notes

- More precise: $T(n) \leq T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + cn$
- Usually, we can asymptotically ignore floor and ceilings.
- Essentially, we are assuming n is a power of 2.
- Alternate form: $T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + O(n); T(1) \leq O(1)$

Methods

- Unwind / Recurrence Tree
- Guess
- Master Theorem
- Nuclear Bomb Theorem / Master Master Theorem

UNWIND MERGESORT RECURRENCE

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

UNWIND MERGESORT RECURRENCE

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \end{aligned}$$

UNWIND MERGESORT RECURRENCE

$$\begin{aligned}T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\&\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\&\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn\end{aligned}$$

UNWIND MERGESORT RECURRENCE

$$\begin{aligned}T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\ &\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\ &\vdots \\ &\leq 2^k T\left(\frac{n}{2^k}\right) + kcn\end{aligned}$$

UNWIND MERGESORT RECURRENCE

$$\begin{aligned}T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\ &\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\ &\vdots \\ &\leq 2^k T\left(\frac{n}{2^k}\right) + kcn\end{aligned}$$

$$\begin{aligned}1 &= \frac{n}{2^k} \\ \iff 2^k &= n \\ \iff k &= \log_2(n)\end{aligned}$$

UNWIND MERGESORT RECURRENCE

$$\begin{aligned}T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\ &\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\ &\vdots \\ &\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\ &= nT(1) + cn \log(n)\end{aligned}$$

$$\begin{aligned}1 &= \frac{n}{2^k} \\ \iff 2^k &= n \\ \iff k &= \log_2(n)\end{aligned}$$

UNWIND MERGESORT RECURRENCE

$$\begin{aligned}
 T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
 &\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
 &\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\
 &\vdots \\
 &\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\
 &= nT(1) + cn \log(n) \\
 &= cn + cn \log n
 \end{aligned}$$

$$\begin{aligned}
 1 &= \frac{n}{2^k} \\
 \iff 2^k &= n \\
 \iff k &= \log_2(n)
 \end{aligned}$$

UNWIND MERGESORT RECURRENCE

$$\begin{aligned}
 T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
 &\leq 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\
 &\leq 2\left(2\left(2T\left(\frac{n}{2^3}\right) + c\frac{n}{2^2}\right) + c\frac{n}{2}\right) + cn \\
 &\vdots \\
 &\leq 2^k T\left(\frac{n}{2^k}\right) + kcn \\
 &= nT(1) + cn \log(n) \\
 &= cn + cn \log n \\
 &= O(n \log(n))
 \end{aligned}$$

$$\begin{aligned}
 1 &= \frac{n}{2^k} \\
 \iff 2^k &= n \\
 \iff k &= \log_2(n)
 \end{aligned}$$

RECURSION TREE METHOD

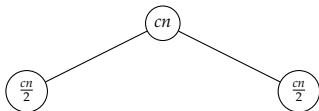
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

$$\textcircled{cn}$$

¹Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

RECURSION TREE METHOD

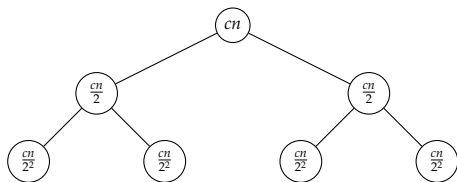
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



¹Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

RECURSION TREE METHOD

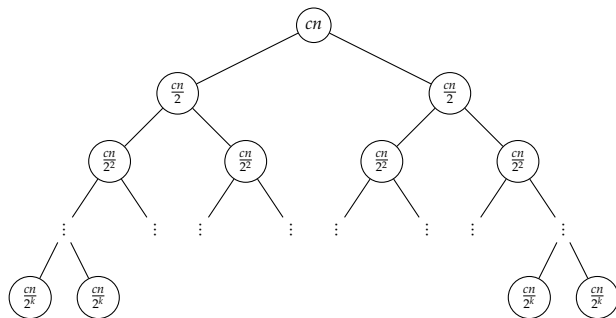
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



¹Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

RECURSION TREE METHOD

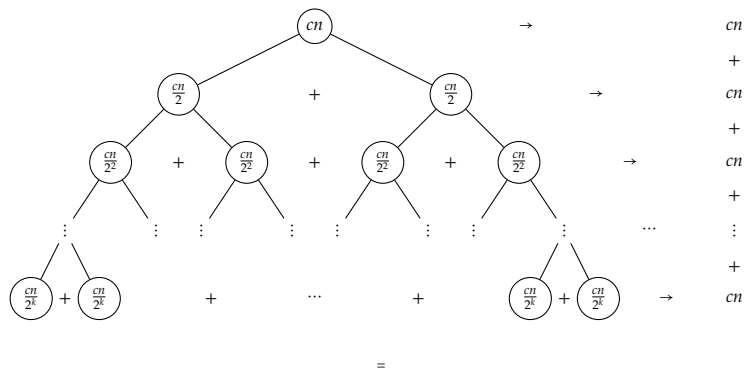
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



¹Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

RECURSION TREE METHOD

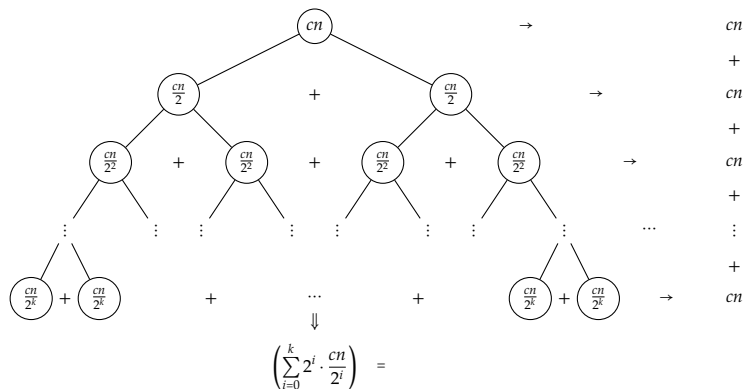
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



¹Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

RECURSION TREE METHOD

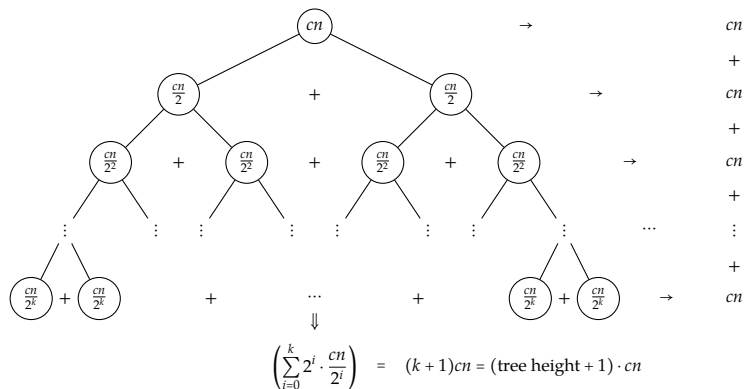
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



¹Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

RECURSION TREE METHOD

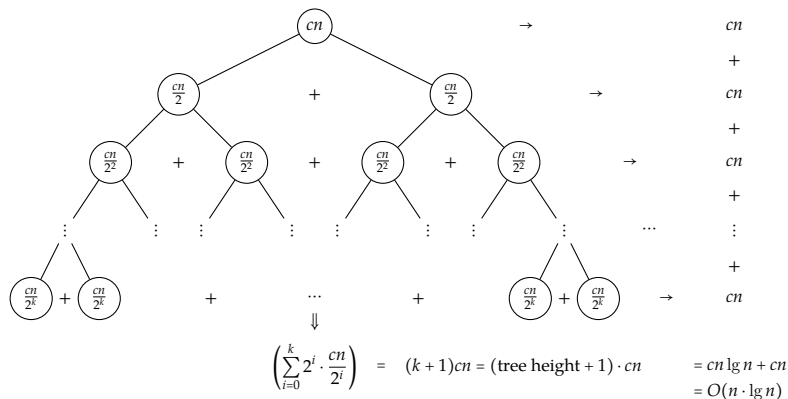
$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



¹Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

RECURSION TREE METHOD

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$



¹Based on: <http://www.texample.net/tikz/examples/merge-sort-recursion-tree/>

GUESS METHOD

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Procedure

- 1 Guess: Seems like $O(n \log n)$ -ish.

GUESS METHOD

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Procedure

- 1 Guess: Seems like $O(n \log n)$ -ish.
- 2 Prove by induction! Not valid without proof!

PROVE RECURRENCE BY STRONG INDUCTION

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq cn \lg n + cn; T(1) \leq c$$

PROVE RECURRENCE BY STRONG INDUCTION

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq cn \lg n + cn; T(1) \leq c$$

Base Case: $n = 2$.

$$\begin{aligned} T(2) &= 2 \cdot T(1) + 2c \leq 4c \\ &= c \cdot 2 \lg 2 + 2c \end{aligned}$$

PROVE RECURRENCE BY STRONG INDUCTION

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq cn \lg n + cn; T(1) \leq c$$

Base Case: $n = 2$.

$$\begin{aligned} T(2) &= 2 \cdot T(1) + 2c \leq 4c \\ &= c \cdot 2 \lg 2 + 2c \end{aligned}$$

Inductive step:

$$\begin{aligned} T(k) &= 2 \cdot T(k/2) + ck \\ &\leq 2 \left(\frac{ck}{2} \lg \frac{k}{2} + \frac{ck}{2} \right) + ck \\ &= ck \lg(k/2) + 2ck \\ &= ck \lg k - ck + 2ck \\ &= ck \lg k + ck \end{aligned}$$

PROVE RECURRENCE BY STRONG INDUCTION

$$T(n) \leq 2 \cdot T\left(\frac{n}{2}\right) + cn \leq cn \lg n + cn; T(1) \leq c$$

Base Case: $n = 2$.

$$\begin{aligned} T(2) &= 2 \cdot T(1) + 2c \leq 4c \\ &= c \cdot 2 \lg 2 + 2c \end{aligned}$$

Inductive step:

$$\begin{aligned} T(k) &= 2 \cdot T(k/2) + ck \\ &\leq 2 \left(\frac{ck}{2} \lg \frac{k}{2} + \frac{ck}{2} \right) + ck \\ &= ck \lg(k/2) + 2ck \\ &= ck \lg k - ck + 2ck \\ &= ck \lg k + ck \end{aligned}$$

$\therefore O(n \log n)$

GENERALIZED RECURRENCE

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

GENERALIZED RECURRENCE

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

GENERALIZED RECURRENCE

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

$O(n^{\lg q})$

GENERALIZED RECURRENCE

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

$O(n^{\lg q})$

Case $q = 2$

$O(n \log n)$

GENERALIZED RECURRENCE

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

$O(n^{\lg q})$

Case $q = 2$

$O(n \log n)$

Case $q = 1$

GENERALIZED RECURRENCE

$$T(n) \leq q \cdot T\left(\frac{n}{2}\right) + cn; T(1) \leq c$$

Case $q > 2$

$O(n^{\lg q})$

Case $q = 2$

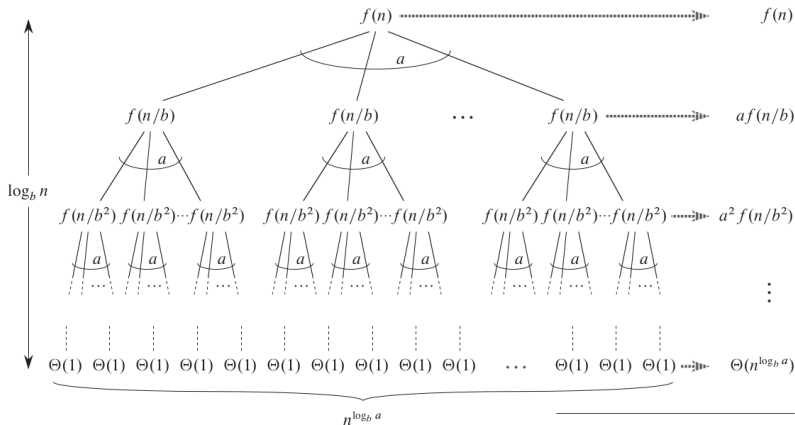
$O(n \log n)$

Case $q = 1$

$O(n)$

MASTER THEOREM

COOKBOOK RECURRENCE SOLVING



$$\text{Total: } \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

MASTER THEOREM

COOKBOOK RECURRENCE SOLVING

Theorem 1

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then $T(n)$ has the following asymptotic bounds:

- ❶ If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- ❷ If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- ❸ If $\Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $a \cdot f(n/b) \leq c \cdot f(n)$ for some constant $c < 1$ and all sufficiently large n , then $T(n) = \Theta(f(n))$.

NUCLEAR BOMB / MASTER MASTER THEOREM

AKRA AND BAZZI, 1998

Theorem 2

Given a recurrence of the form:

$$T(n) = \sum_{i=1}^k a_i T(n/b_i) + f(n) ,$$

where k is a constant, $a_i > 0$ and $b_i > 1$ are constants for all i , and $f(n) = \Omega(n^c)$ and $f(n) = O(n^d)$ for some constants $0 < c \leq d$. Then,

$$T(n) = \Theta \left(n^\rho \left(1 + \int_1^n \frac{f(u)}{u^{\rho+1}} du \right) \right) ,$$

where ρ is the unique real solution to the equation

$$\sum_{i=1}^k \frac{a_i}{b_i^\rho} = 1 .$$

SEARCH

SEARCHING FOR AN ELEMENT

UNSORTED ARRAY

Linear Search

- Brute force approach: check every item in order.

Algorithm 31: LINEAR SEARCH

Input : A list of items $A[1 \dots n]$ and a target element x .

Output: The index i such that $A[i] = x$, or -1 if x is not found.

```
for  $i \leftarrow 1$  to  $n$  do
    if  $A[i] = x$  then
        return  $i$  // Return the index if  $x$  is found
    end
end
return  $-1$  // Return  $-1$  if  $x$  is not found in the list
```

SEARCHING FOR AN ELEMENT

UNSORTED ARRAY

Linear Search

- Brute force approach: check every item in order.
- 🤖 What is the time complexity to search through n items?

Algorithm 32: LINEAR SEARCH

Input : A list of items $A[1 \dots n]$ and a target element x .

Output: The index i such that $A[i] = x$, or -1 if x is not found.

```
for  $i \leftarrow 1$  to  $n$  do
  if  $A[i] = x$  then
    return  $i$  // Return the index if  $x$  is found
  end
end
return  $-1$  // Return  $-1$  if  $x$  is not found in the list
```

SEARCHING FOR AN ELEMENT

UNSORTED ARRAY

Linear Search

- Brute force approach: check every item in order.
- 🤖 What is the time complexity to search through n items?

Algorithm 33: LINEAR SEARCH

Input : A list of items $A[1 \dots n]$ and a target element x .

Output: The index i such that $A[i] = x$, or -1 if x is not found.

for $i \leftarrow 1$ to n do

 if $A[i] = x$ then

 return i

 // Return the index if x is found

 end

end

return -1

 // Return -1 if x is not found in the list

Time complexity:

$$T(n) = T(n - 1) + \Theta(1)$$

SEARCHING FOR AN ELEMENT

UNSORTED ARRAY

Linear Search

- Brute force approach: check every item in order.
- Time complexity: $O(n)$

Algorithm 34: LINEAR SEARCH

Input : A list of items $A[1 \dots n]$ and a target element x .

Output: The index i such that $A[i] = x$, or -1 if x is not found.

```

for  $i \leftarrow 1$  to  $n$  do
    if  $A[i] = x$  then
        return  $i$  // Return the index if  $x$  is found
    end
end
return  $-1$  // Return  $-1$  if  $x$  is not found in the list

```

Time complexity:

$$T(n) = T(n - 1) + \Theta(1)$$

SEARCHING FOR AN ELEMENT

SORTED ARRAY

Divide and Conquer Approach

- Start by dividing the array in half and compare the target element with the middle element.

SEARCHING FOR AN ELEMENT

SORTED ARRAY

Divide and Conquer Approach

- Start by dividing the array in half and compare the target element with the middle element.
- If the target is less than the middle element, search the left half; otherwise, search the right half.

SEARCHING FOR AN ELEMENT

SORTED ARRAY

Divide and Conquer Approach

- Start by dividing the array in half and compare the target element with the middle element.
- If the target is less than the middle element, search the left half; otherwise, search the right half.
- 🤖 What is the time complexity of searching through n sorted items?

SEARCHING FOR AN ELEMENT

SORTED ARRAY

- 🤖 What is the time complexity of searching through n sorted items?

Algorithm 35: BINARY SEARCH

Input : A sorted list of items $A[1 \dots n]$ and a target element x .

Output: The index i such that $A[i] = x$, or -1 if x is not found.

```

while low  $\leq$  high do
    mid  $\leftarrow$   $\lfloor \frac{\text{low} + \text{high}}{2} \rfloor$ 
    if  $A[\text{mid}] = x$  then
        return mid // Return the index if  $x$  is found
    else
        if  $A[\text{mid}] < x$  then
            low  $\leftarrow$  mid + 1
        else
            high  $\leftarrow$  mid - 1
        end
    end
end
return -1 // Return -1 if  $x$  is not found in the list

```

SEARCHING FOR AN ELEMENT

SORTED ARRAY

Divide and Conquer Approach

- Start by dividing the array in half and compare the target element with the middle element.
- If the target is less than the middle element, search the left half; otherwise, search the right half.
- 🤖 What is the time complexity of searching through n sorted items?

Time complexity:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

SEARCHING FOR AN ELEMENT

SORTED ARRAY

Divide and Conquer Approach

- Start by dividing the array in half and compare the target element with the middle element.
- If the target is less than the middle element, search the left half; otherwise, search the right half.
- 🤖 What is the time complexity of searching through n sorted items?

Time complexity:

$$T(n) = T\left(\frac{n}{4}\right) + \Theta(1) + \Theta(1)$$

SEARCHING FOR AN ELEMENT

SORTED ARRAY

Divide and Conquer Approach

- Start by dividing the array in half and compare the target element with the middle element.
- If the target is less than the middle element, search the left half; otherwise, search the right half.
- 🤖 What is the time complexity of searching through n sorted items?

Time complexity:

$$T(n) = T\left(\frac{n}{4}\right) + \Theta(1) + \Theta(1) = \dots = T\left(\frac{n}{2^k}\right) + k \times \Theta(1)$$

SEARCHING FOR AN ELEMENT

SORTED ARRAY

Divide and Conquer Approach

- Start by dividing the array in half and compare the target element with the middle element.
- If the target is less than the middle element, search the left half; otherwise, search the right half.
- 🤖 What is the time complexity of searching through n sorted items? Time complexity: $O(\log n)$

Time complexity:

$$T(n) = T\left(\underbrace{\frac{n}{2^k}}_{k=\log n} = 1\right) + \log n \times \Theta(1) = \Theta(\log n)$$

LOWER BOUNDS

CAN WE DO BETTER THAN
THIS?

TECHNIQUES TO PROVE LOWER BOUNDS

Lower bounds are used:

- To determine the minimum amount of work any algorithm must do to solve a problem.

TECHNIQUES TO PROVE LOWER BOUNDS

Lower bounds are used:

- To determine the minimum amount of work any algorithm must do to solve a problem.
- To recognize when further improvement is futile.

REMINDER: WORST-CASE COMPLEXITY

Worst-case Complexity

Definition: Considering all possible inputs, what is the worst possible performance of the algorithm?

- Provides an absolute guarantee on performance.
- Based on the single most challenging input.

$$T_{\text{worst}}^A(n) = \max_{i \in I, |i|=n} T_A(i)$$

What does the worst-case complexity of a problem mean?

$$T_{\text{worst}}^{\Pi}(n) = \min_{A \text{ solves } \Pi} T_{\text{worst}}^A(n)$$

REMINDER: WORST-CASE COMPLEXITY

Worst-case Complexity

Definition: Considering all possible inputs, what is the worst possible performance of the algorithm?

- Provides an absolute guarantee on performance.
- Based on the single most challenging input.

$$T_{\text{worst}}^A(n) = \max_{i \in I, |i|=n} T_A(i)$$

What does the worst-case complexity of a problem mean?

$$T_{\text{worst}}^{\Pi}(n) = \min_{A \text{ solves } \Pi} T_{\text{worst}}^A(n)$$

$T_{\text{worst}}^{\Pi}(n) \leq T_{\text{worst}}^A(n)$ for any algorithm A that solves Π

REMINDER: WORST-CASE COMPLEXITY

Worst-case Complexity

Definition: Considering all possible inputs, what is the worst possible performance of the algorithm?

- Provides an absolute guarantee on performance.
- Based on the single most challenging input.

$$T_{\text{worst}}^A(n) = \max_{i \in I, |i|=n} T_A(i)$$

What does the worst-case complexity of a problem mean?

$$T_{\text{worst}}^{\Pi}(n) = \min_{A \text{ solves } \Pi} T_{\text{worst}}^A(n)$$

$T_{\text{worst}}^{\Pi}(n) \leq T_{\text{worst}}^A(n)$ for any algorithm A that solves Π

An algorithm **upper-bounds** the complexity of a problem.

REMINDER: WORST-CASE COMPLEXITY

Worst-case Complexity

Definition: Considering all possible inputs, what is the worst possible performance of the algorithm?

- Provides an absolute guarantee on performance.
- Based on the single most challenging input.

$$T_{\text{worst}}^A(n) = \max_{i \in I, |i|=n} T_A(i)$$

What does the worst-case complexity of a problem mean?

$$T_{\text{worst}}^{\Pi}(n) = \min_{A \text{ solves } \Pi} T_{\text{worst}}^A(n)$$

$T_{\text{worst}}^{\Pi}(n) \leq T_{\text{worst}}^A(n)$ for any algorithm A that solves Π

🧠 How do we show a lower bound in complexity?

REMINDER: WORST-CASE COMPLEXITY

Worst-case Complexity

Definition: Considering all possible inputs, what is the worst possible performance of the algorithm?

- Provides an absolute guarantee on performance.
- Based on the single most challenging input.

$$T_{\text{worst}}^A(n) = \max_{i \in I, |i|=n} T_A(i)$$

What does the worst-case complexity of a problem mean?

$$T_{\text{worst}}^{\Pi}(n) = \min_{A \text{ solves } \Pi} T_{\text{worst}}^A(n)$$

$T_{\text{worst}}^{\Pi}(n) \leq T_{\text{worst}}^A(n)$ for any algorithm A that solves Π

We must prove that certain steps are **necessary** for any algorithm ☺!!!

TECHNIQUES TO PROVE LOWER BOUNDS

Main Ideas (*Proof by Contradiction*):

1. Adversary Argument

- Imagine an adversary making the problem as hard as possible for the algorithm.

TECHNIQUES TO PROVE LOWER BOUNDS

Main Ideas (*Proof by Contradiction*):

1. Adversary Argument

- Imagine an adversary making the problem as hard as possible for the algorithm.
- The adversary strategically forces the algorithm to take the maximum number of steps.

TECHNIQUES TO PROVE LOWER BOUNDS

Main Ideas (*Proof by Contradiction*):

1. Adversary Argument

- Imagine an adversary making the problem as hard as possible for the algorithm.
- The adversary strategically forces the algorithm to take the maximum number of steps.
- This approach proves that any algorithm must take a certain number of steps to guarantee correctness.

TECHNIQUES TO PROVE LOWER BOUNDS

Main Ideas (*Proof by Contradiction*):

2. Decision Tree Model & Pigeonhole Principle

- **Decision Tree Model:** Visualizes the algorithm's decision-making process, where each node is a **query**, and each leaf is an **outcome**.

TECHNIQUES TO PROVE LOWER BOUNDS

Main Ideas (*Proof by Contradiction*):

2. Decision Tree Model & Pigeonhole Principle

- **Decision Tree Model:** Visualizes the algorithm's decision-making process, where each node is a **query**, and each leaf is an **outcome**.
- **Pigeonhole Principle:** More outcomes than execution paths mean some paths must produce multiple outputs violating the correctness.

TECHNIQUES TO PROVE LOWER BOUNDS

Main Ideas (*Proof by Contradiction*):

2. Decision Tree Model & Pigeonhole Principle

- **Decision Tree Model:** Visualizes the algorithm's decision-making process, where each node is a **query**, and each leaf is an **outcome**.
- **Pigeonhole Principle:** More outcomes than execution paths mean some paths must produce multiple outputs violating the correctness.
- *The height of the decision tree represents the worst-case number of steps the algorithm takes.*

ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

Can we search for an element x in an unsorted array A without scanning the entire array?

ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

Can we search for an element x in an unsorted array A without scanning the entire array?

For contradiction, assume it is possible with at most $n - 2$ steps:

ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

Can we search for an element x in an unsorted array A without scanning the entire array?

For contradiction, assume it is possible with at most $n - 2$ steps:

- **Adversary's Strategy:**
 - The adversary knows your search strategy and aims to make it as difficult as possible.

ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

Can we search for an element x in an unsorted array A without scanning the entire array?

For contradiction, assume it is possible with at most $n - 2$ steps:

- **Adversary's Strategy:**
 - The adversary knows your search strategy and aims to make it as difficult as possible.
 - Initially, the adversary has not committed to where x is located in the array.

ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

Can we search for an element x in an unsorted array A without scanning the entire array?

For contradiction, assume it is possible with at most $n - 2$ steps:

- **Adversary's Strategy:**

- The adversary knows your search strategy and aims to make it as difficult as possible.
- Initially, the adversary has not committed to where x is located in the array.

- **Your Search Process:**

- You examine elements one by one, starting from any element.
- If you recheck, you don't gain any new information about the rest of the array.

ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

Can we search for an element x in an unsorted array A without scanning the entire array?

For contradiction, assume it is possible with at most $n - 2$ steps:

- **Adversary's Strategy:**
 - The adversary knows your search strategy and aims to make it as difficult as possible.
 - Initially, the adversary has not committed to where x is located in the array.
- **Your Search Process:**
 - You examine elements one by one, starting from any element.
 - If you recheck, you don't gain any new information about the rest of the array.
 - For each element $A[i]$ you check, the adversary can respond with any value not equal to x .

ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

Can we search for an element x in an unsorted array A without scanning the entire array?

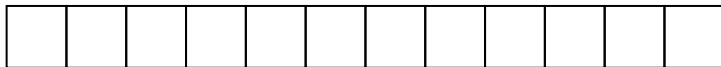
For contradiction, assume it is possible with at most $n - 2$ steps:

- **Adversary's Strategy:**
 - The adversary knows your search strategy and aims to make it as difficult as possible.
 - Initially, the adversary has not committed to where x is located in the array.
- **Your Search Process:**
 - You examine elements one by one, starting from any element.
 - If you recheck, you don't gain any new information about the rest of the array.
 - For each element $A[i]$ you check, the adversary can respond with any value not equal to x .
- **Worst-Case Scenario:** Knowing x is in the array, the adversary can place it in the last unchecked position ☺

ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

ILLUSTRATION

We: - Is it here?



ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

ILLUSTRATION

We: - Is it here?

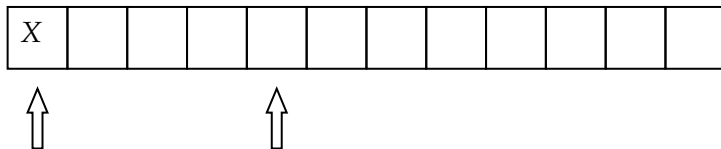
Adversary: -No 😊



ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

ILLUSTRATION

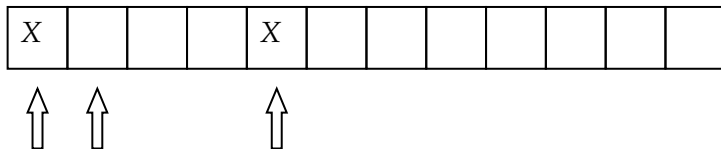
We: - Is it here?



ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

ILLUSTRATION

We: - Is it here?

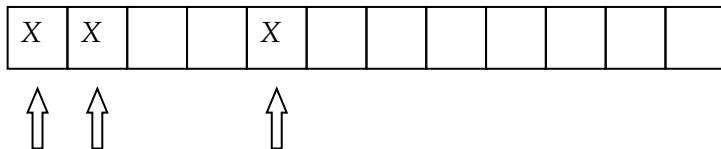


ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

ILLUSTRATION

We: - Is it here?

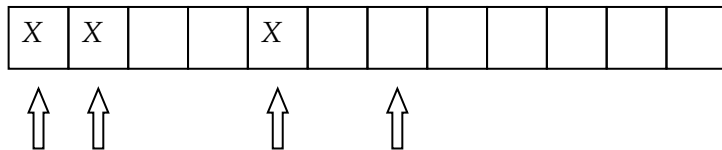
Adversary: -No 😊



ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

ILLUSTRATION

We: - Is it here?

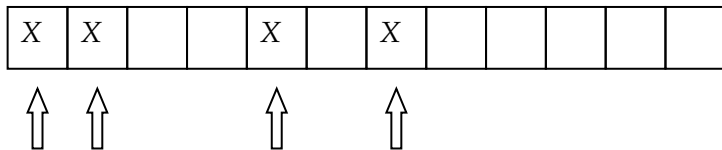


ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

ILLUSTRATION

We: - Is it here?

Adversary: -No 😊

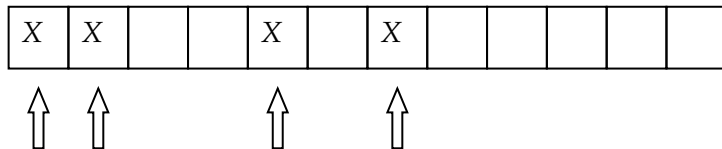


ADVERSARY ARGUMENT FOR $\Omega(n)$ LINEAR SEARCH

ILLUSTRATION

We: - Is it here?

Adversary: -No 😊



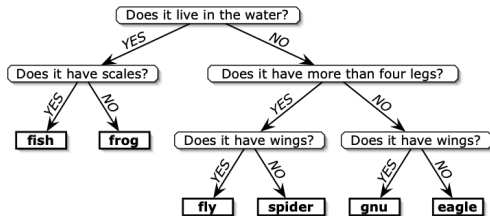
If we use $n - 2$ queries, at the end adversary can hide x to the unchecked one.

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY

Assume that your algorithm can only ask comparison questions of the form $x \stackrel{>}{=} k?$.

What is the worst-case performance of the FINDANIMAL?



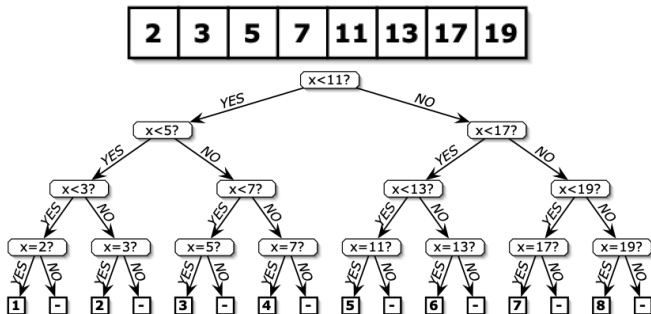
A decision tree to choose one of six animals.

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY

Worst-Case Performance

The worst-case performance of our algorithm is determined by the height of the decision tree.

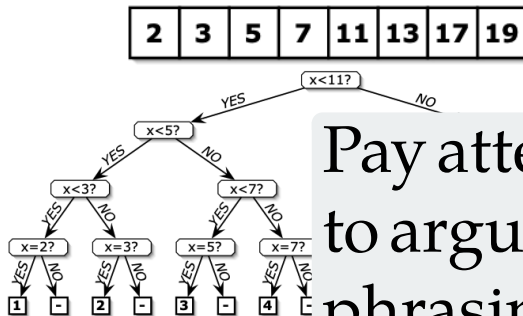


$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY

Worst-Case Performance

The worst-case performance of our algorithm is determined by the height of the decision tree.



Pay attention
to argument
phrasing!!!

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

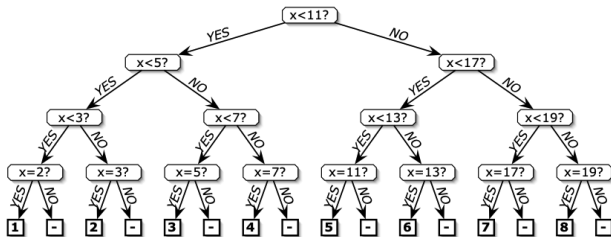
SORTED ARRAY

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.
- There are arrays where the answer could be at the first position, the second position, ..., the last position, or the element may not be present at all.

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.
- Since the algorithm relies on binary comparisons (with 2 possible outcomes), it essentially constructs a binary decision tree.

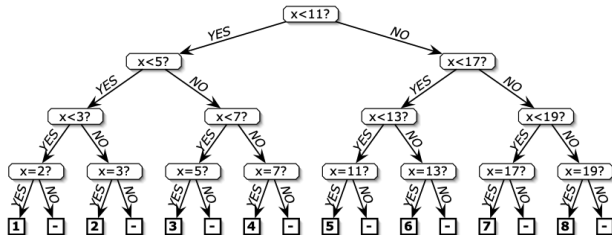


Why does the algorithm construct a binary decision tree?

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.

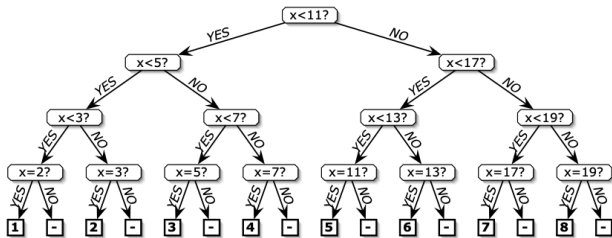


Why does the algorithm construct a **binary** decision tree?

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.

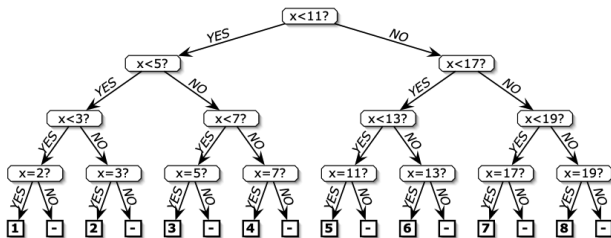


- Because each query/decision provides a "Yes" or "No" answer, which naturally forms a binary structure.

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.

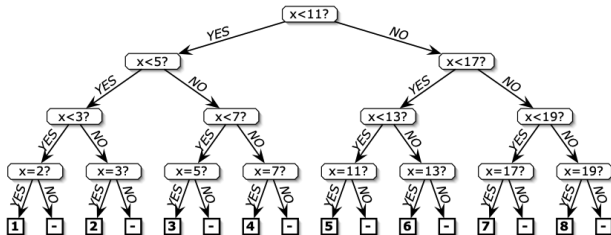


- Because each query/decision provides a "Yes" or "No" answer, which naturally forms a binary structure.
- If the algorithm could ask questions with 3, 4, or k outcomes, it would form a ternary, quaternary, or k -ary tree instead.

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.

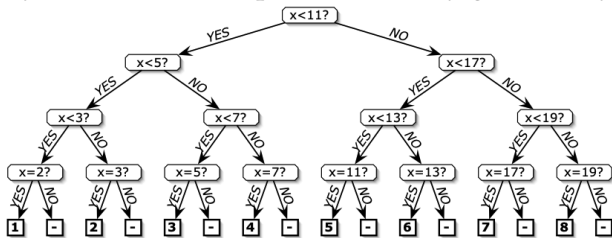


Why is the algorithm modeled by a tree?

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.



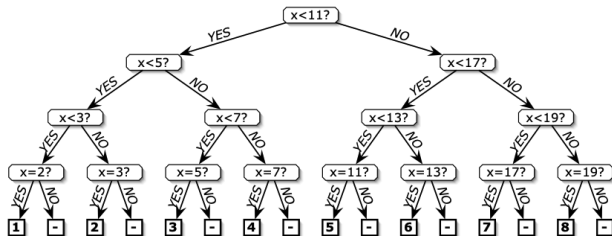
🤖 Why is the algorithm modeled by a tree?

Since the input data doesn't change, there's no need to ask the same question twice. Hence, the algorithm doesn't form cycles, but rather a tree.

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.



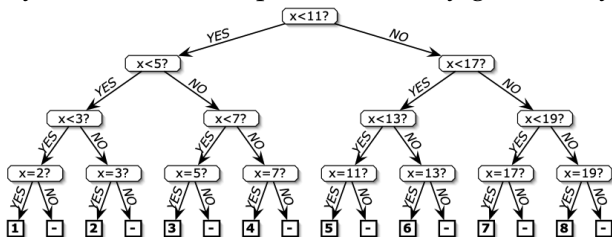
🧠 Why is the algorithm modeled by a tree?

🧠 How many different answers can the algorithm give?

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.



Why is the algorithm modeled by a tree?

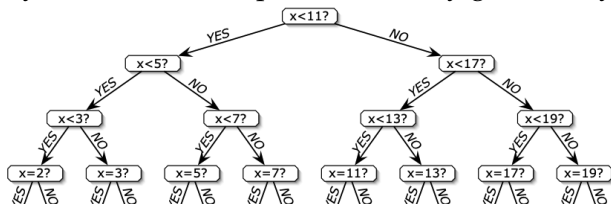
How many different answers can the algorithm give?

As many as the number of leaves in the tree.

$\Omega(\log n)$ SEARCH BASED ON COMPARISONS

SORTED ARRAY [2,3,5,7,11,13,17,19]

- Suppose we have an algorithm that uses comparisons to correctly solve the search problem for any given array.



Any algorithm/tree with k comparisons can have at most 2^k leaves. If M answers are needed, then $2^k \geq M$, which implies $k \geq \log_2 M$.

🤖 How many different answers can the algorithm give?

As many as the number of leaves in the tree.

LOWER BOUND IN COMPARISON SORTING ALGORITHMS

Let's see what we've learned...

🧐 If you need to sort an array and only perform comparisons, how many comparisons are necessary to account for all possible orderings?

LOWER BOUND IN COMPARISON SORTING ALGORITHMS

Let's see what we've learned...

🤖 If you need to sort an array and only perform comparisons, how many comparisons are necessary to account for all possible orderings?

For an array of 3 elements, consider all possible permutations:

$$\left\{ \begin{array}{l} A[1] < A[2] < A[3], \\ A[2] < A[1] < A[3], \\ A[1] < A[3] < A[2], \\ A[2] < A[3] < A[1], \\ A[3] < A[1] < A[2], \\ A[3] < A[2] < A[1] \end{array} \right.$$

LOWER BOUND IN COMPARISON SORTING ALGORITHMS

Let's see what we've learned...

🤖 If you need to sort an array and only perform comparisons, how many comparisons are necessary to account for all possible orderings?

Every comparison-based algorithm after k comparisons must produce a decision tree with at least $N!$ leaves. Therefore, it needs at least:

$$2^k \geq N! \Rightarrow k \geq \log_2 N! = \Omega(N \log N)$$

comparisons.

$$\begin{cases} A[3] < A[1] < A[3], \\ A[3] < A[2] < A[1] \end{cases}$$

LOWER BOUNDS

REDUCTIONS TO OTHER PROBLEMS

☹ Assume that you have an algorithm A (comparison-based) that sorts a sequence that is almost correct except for the last two elements:


$$\text{Input: } \{a_i\}_{i=1}^n \implies \text{Output: } \{a_{\pi(i_1)} < a_{\pi(i_2)} < \dots < a_{\pi(i_{n-2})}\}$$

But you do not know the order of the last two elements. Why does Algo A have a lower bound of $\Omega(n \log n)$?

Consider both cases: with and without counting.

LOWER BOUNDS

REDUCTIONS TO OTHER PROBLEMS

 Assume that you have an algorithm A (comparison-based) that sorts a sequence that is almost correct except for the last two elements:

$$\text{Input: } \{a_i\}_{i=1}^n \implies \text{Output: } \{a_{\pi(i_1)} < a_{\pi(i_2)} < \dots < a_{\pi(i_{n-2})}\}$$


But you do not know the order of the last two elements. Why does Algo A have a lower bound of $\Omega(n \log n)$?

Consider both cases: with and without counting.

- **With Counting:** The number of possible outputs is $n!/2!$, because the algorithm is indifferent regarding the last two positions. Therefore, the time complexity is $\Omega(\log(n!/2))$.

LOWER BOUNDS

REDUCTIONS TO OTHER PROBLEMS

 Assume that you have an algorithm A (comparison-based) that sorts a sequence that is almost correct except for the last two elements:

$$\text{Input: } \{a_i\}_{i=1}^n \implies \text{Output: } \{a_{\pi(i_1)} < a_{\pi(i_2)} < \dots < a_{\pi(i_{n-2})}\}$$

But you do not know the order of the last two elements. Why does Algo A have a lower bound of $\Omega(n \log n)$?

Consider both cases: with and without counting.


- **Without Counting:** Assume that A has a complexity of $o(n \log n)$. Then, I can create an algorithm B as follows:

Algorithm B : {Apply A ; Compare the last two elements}

This algorithm B sorts an arbitrary array in $o(n \log n) + \Theta(1) = o(n \log n)$ using only comparisons

LOWER BOUNDS

REDUCTIONS TO OTHER PROBLEMS

 Assume that your array contains only ones and zeros. What is the lower bound? What is the upper bound?

LOWER BOUNDS

REDUCTIONS TO OTHER PROBLEMS

🧐 Assume that your array contains only ones and zeros. What is the lower bound? What is the upper bound?

- The higher lower bound reflects the complexity better. Can you suggest a trivial lower bound of $\Omega(1)$?

LOWER BOUNDS

REDUCTIONS TO OTHER PROBLEMS

🤖 Assume that your array contains only ones and zeros. What is the lower bound? What is the upper bound?

- **Lower Bound:** Consider the case where the array has $n - 1$ ones and 1 zero. In this case, the complexity is $\log(n!/(n - 1)!) = \log n$.
The adversary argument gives a higher lower bound since the zero is the minimum element.

LOWER BOUNDS

REDUCTIONS TO OTHER PROBLEMS

🤖 Assume that your array contains only ones and zeros. What is the lower bound? What is the upper bound?

- **More Interesting Case:** If the zeros and ones are split evenly, the complexity is:

$$\log \left(\frac{2n!}{n!n!} \right) = \log \binom{2n}{n} = \log(2n!) - 2 \log(n!)$$

Using Stirling's approximation, we get:

$$2n \log(2n) - 2n + O(\log 2n) - 2n \log n + 2n - 2O(\log n) = \Omega(n)$$

LOWER BOUNDS

REDUCTIONS TO OTHER PROBLEMS

🧠 Assume that your array contains only ones and zeros. What is the lower bound? What is the upper bound?

- **Upper Bound:** A linear-time comparison-based algorithm via PARTITION (Teaser for QUICKSORT).

BINARY SEARCH:

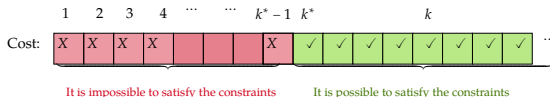
FROM DECISION TO OPTIMIZATION

BINARY SEARCH IN THE SOLUTION SPACE

HIGH-LEVEL TECHNIQUE

Perhaps one of the most fundamental techniques in solving difficult problems—and one of the key tricks in computer science interviews.

Binary search is not only applicable to sorted arrays but also to the solution space.



BINARY SEARCH IN THE SOLUTION SPACE

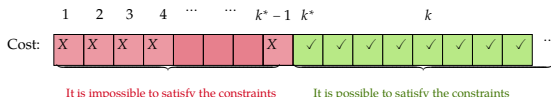
HIGH-LEVEL TECHNIQUE

Binary search is not only applicable to sorted arrays but also to the solution space.

For example:

- Compute the minimum cost in a Constraint-Satisfaction problem (CSP) with n constraints.
- Define a cost function and query: "Is there a solution with cost at most k ?"
- If yes, check for a solution with cost $\leq k - 1$, and so on.
- \vdots
- If no, find the optimal solution.

Do you need to go step-by-step, or can you apply binary search to find the optimal solution?



BINARY SEARCH IN THE SOLUTION SPACE

EXAMPLE

Problem Definition

- You're planning a long trip with $k \geq 2$ (e.g. 10) days and need to visit $n > k$ locations (e.g. Big cities of USA).
- The distances between consecutive stops are d_1, d_2, \dots, d_n where d_1 is the distance from the start, d_2 is the distance from the first to the second stop, etc.
- To avoid fatigue, you want to minimize the maximum distance traveled in a single day. You can stop for an overnight rest only at selected stops.

BINARY SEARCH IN THE SOLUTION SPACE

EXAMPLE

Problem Definition

- You're planning a long trip with $k \geq 2$ (e.g. 10) days and need to visit $n > k$ locations (e.g. Big cities of USA).
- The distances between consecutive stops are d_1, d_2, \dots, d_n where d_1 is the distance from the start, d_2 is the distance from the first to the second stop, etc.
- To avoid fatigue, you want to minimize the maximum distance traveled in a single day. You can stop for an overnight rest only at selected stops.

$$\min \text{cost}(\text{Separation}\{d_1, \dots, d_n \mid k\}) = \min_{\text{Separation}\{d_1, \dots, d_n \mid k\}} \max_{\text{day} \in [k]} \sum d_i^{[\text{day}]}$$

BINARY SEARCH FOR THE OPTIMAL SOLUTION

Step-by-Step Approach

- We start by defining the feasible region: the minimum possible value and the maximum possible value for the longest single-day distance.

BINARY SEARCH FOR THE OPTIMAL SOLUTION

Step-by-Step Approach

- We start by defining the feasible region: the minimum possible value and the maximum possible value for the longest single-day distance.
- Using binary search, we iteratively narrow down the range by checking whether a given midpoint value can be a valid solution (i.e., if it is possible to split the journey into k days where no day exceeds this distance).

BINARY SEARCH FOR THE OPTIMAL SOLUTION

Step-by-Step Approach

- We start by defining the feasible region: the minimum possible value and the maximum possible value for the longest single-day distance.
- Using binary search, we iteratively narrow down the range by checking whether a given midpoint value can be a valid solution (i.e., if it is possible to split the journey into k days where no day exceeds this distance).
- The optimal solution is found when the range converges, providing the minimal maximum distance that allows the trip to be completed in k days.

BINARY SEARCH FOR THE OPTIMAL SOLUTION

Step-by-Step Approach

- We start by defining the feasible region: the minimum possible value and the maximum possible value for the longest single-day distance.
- Using binary search, we iteratively narrow down the range by checking whether a given midpoint value can be a valid solution (i.e., if it is possible to split the journey into k days where no day exceeds this distance).
- The optimal solution is found when the range converges, providing the minimal maximum distance that allows the trip to be completed in k days.

Computational Complexity

- The binary search operates on the solution space, which has a logarithmic range (based on the total distance of the trip).
- For each midpoint value tested, a linear scan of the array d_1, d_2, \dots, d_n is performed to check feasibility.
- Therefore, the overall complexity is $O(n \log D)$, where D is the range of

INVERSION COUNT

COUNTING INVERSIONS

Inversion

Given a list A of comparable items. An inversion is a pair of items (a_i, a_j) such that $a_i > a_j$ and $i < j$, where i and j are the index of the items in A .

COUNTING INVERSIONS

Inversion

Given a list A of comparable items. An inversion is a pair of items (a_i, a_j) such that $a_i > a_j$ and $i < j$, where i and j are the index of the items in A .

Inversion Count

Count the number of inversions in a list A , containing n comparable items.

PART 1: GIVE A $\Theta(n^2)$ SOLUTION.

Algorithm: CHECKALLPAIRS

Input : A list A of n comparable items.

Output: Number of inversions in A .

Let $c := 0$

```
for  $i := 1$  to  $\text{len}(A) - 1$  do
  for  $j := i$  to  $\text{len}(A)$  do
    if  $A[i] > A[j]$  then
       $c := c + 1$ 
    end
  end
end
end
return  $c$ 
```

PART 1: GIVE A $\Theta(n^2)$ SOLUTION.

Algorithm: CHECKALLPAIRS

Input : A list A of n comparable items.

Output: Number of inversions in A .

Let $c := 0$

```
for  $i := 1$  to  $\text{len}(A) - 1$  do
  for  $j := i + 1$  to  $\text{len}(A)$  do
    if  $A[i] > A[j]$  then
       $c := c + 1$ 
    end
  end
end
return  $c$ 
```

Analysis

- Correct: Checks all pairs and counts the inversions.

Teaser for Homework ☺

Idea...In order to sort an array, you have to fix all the inversions

PART 1: GIVE A $\Theta(n^2)$ SOLUTION.

Algorithm: CHECKALLPAIRS

Input : A list A of n comparable items.

Output: Number of inversions in A .

Let $c := 0$

```

for  $i := 1$  to  $\text{len}(A) - 1$  do
  |
  | for  $j := i$  to  $\text{len}(A)$  do
  | |   if  $A[i] > A[j]$  then
  | | |    $c := c + 1$ 
  | |   end
  | end
end
return  $c$ 

```

Analysis

- Correct: Checks all pairs and counts the inversions.
- Complexity: For each i , check $n - i$ pairs. Overall:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2) .$$

Teaser for Homework ☺

Idea...In order to sort an array, you have to fix all the inversions

PART 1: GIVE A $\Theta(n^2)$ SOLUTION.

Algorithm: CHECKALLPAIRS

Input : A list A of n comparable items.

Output: Number of inversions in A .

Let $c := 0$

```

for  $i := 1$  to  $\text{len}(A) - 1$  do
  |
  | for  $j := i$  to  $\text{len}(A)$  do
  | |   if  $A[i] > A[j]$  then
  | | |    $c := c + 1$ 
  | |   end
  | end
end
return  $c$ 
  
```

Analysis

- Correct: Checks all pairs and counts the inversions.
- Complexity: For each i , check $n - i$ pairs. Overall:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2) .$$

Teaser for Homework ☺

Idea...In order to sort an array, you have to fix all the inversions

PART 2: GIVE A $\Theta(\#INVERSIONS)$ SOLUTION.

Observation: *Inversions*

- An inversion in an array $A[1 \dots n]$ is a pair of indices $i < j$ such that $A[i] > A[j]$.
- The number of inversions indicates how far the array is from being sorted.

PART 2: GIVE A $\Theta(\#INVERSIONS)$ SOLUTION.

Observation: *Inversions*

- An inversion in an array $A[1 \dots n]$ is a pair of indices $i < j$ such that $A[i] > A[j]$.
- The number of inversions indicates how far the array is from being sorted.

Insertion Sort:

- Insertion Sort corrects each inversion one by one.

PART 2: GIVE A $\Theta(\#INVERSIONS)$ SOLUTION.

Observation: *Inversions*

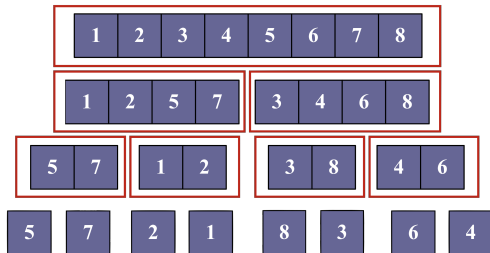
- An inversion in an array $A[1 \dots n]$ is a pair of indices $i < j$ such that $A[i] > A[j]$.
- The number of inversions indicates how far the array is from being sorted.

Insertion Sort:

- Insertion Sort corrects each inversion one by one.
- The total number of swaps made by Insertion Sort equals the number of inversions in the array.
- Time complexity of Insertion Sort in this context is $\Theta(\#Inversions)$.

PART 3: GIVE AN $O(n \log n)$ SOLUTION.

- Can we count the inversions using faster sorting algorithms?
 - Modify the merge step in Merge Sort to count inversions.
 - While merging two sorted halves, every time an element from the right half is placed before an element from the left half, it contributes to inversions.



PART 3: GIVE AN $O(n \log n)$ SOLUTION.

Algorithm: COUNTSORT

Input : A list A of n comparable items.

Output: A sorted array and the number of inversions.

if $|A| = 1$ **then return** $(A, 0)$

$(A_1, c_1) :=$ COUNTSORT(Front-half of A)

$(A_2, c_2) :=$ COUNTSORT(Back-half of A)

$(A, c) :=$ MERGECOUNT(A_1, A_2)

return $(A, c + c_1 + c_2)$

PART 3: GIVE AN $O(n \log n)$ SOLUTION.

Algorithm: MERGECOUNT

Input : Two lists of comparable items: A and B .

Output: A merged list and the count of inversions.

Initialize S to an empty list and $c := 0$.

while either A or B is not empty **do**

 Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

if Appended item is from B **then**

 | $c := c + |A|$.

end

end

return (S, c)

PART 3: GIVE AN $O(n \log n)$ SOLUTION.

Algorithm: MERGECOUNT

Input : Two lists of comparable items: A and B .

Output: A merged list and the count of inversions.

Initialize S to an empty list and $c := 0$.

while either A or B is not empty **do**

 Pop and append $\min\{\text{front of } A, \text{front of } B\}$ to S .

if Appended item is from B **then**

 | $c := c + |A|$.

end

Analysis

- Correctness: Need to show that the inversions are correctly counted.
- Complexity: Same recurrence as MERGESORT, leading to $O(n \log n)$.

APPENDIX

REFERENCES

IMAGE SOURCES I



WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON

<https://brand.wisc.edu/web/logos/>



<https://people.csail.mit.edu/virgi/>