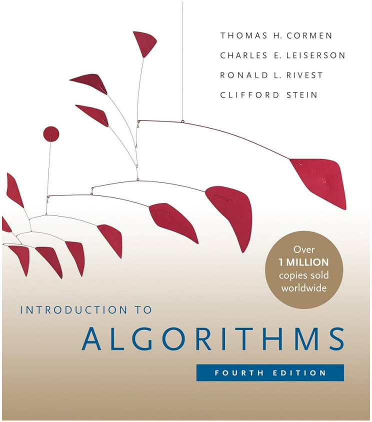
Introduction to Algorithms
Lecture 4: Quicksort

Prof. Charles E. Leiserson and Prof. Erik Demaine
Massachusetts Institute of Technology

August 26, 2025

Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.
Visit https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.
Original slides from *Introduction to Algorithms 6.046J/18.401J*, Fall 2005 Class by Prof. Charles Leiserson and Prof. Erik Demaine. MIT OpenCourseWare Initiative available at https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/.

Plan

Quicksort

- Proposed by C.A.R. Hoare in 1962.

Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.

Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts 'in place' (like insertion sort, but not like merge sort).

Quicksort

- Proposed by C.A.R. Hoare in 1962.
- Divide-and-conquer algorithm.
- Sorts 'in place' (like insertion sort, but not like merge sort).
- Very practical (with tuning).

Plan

Divide and Conquer

Quicksort an n-element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

Divide and Conquer

Quicksort an n-element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

$\leq \; x$	x	$> \; x$

Divide and Conquer

Quicksort an n-element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

$\leq \quad x$	x	$> \quad x$

2. **Conquer:** Recursively sort the two subarrays.

Divide and Conquer

Quicksort an n-element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

$\leq\ x$	x	$>\ x$

2. **Conquer:** Recursively sort the two subarrays.
3. **Combine:** Trivial.

Divide and Conquer

Quicksort an n-element array:

1. **Divide:** Partition the array into two subarrays around a **pivot** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

$\leq \quad x$	x	$> \quad x$

2. **Conquer:** Recursively sort the two subarrays.
3. **Combine:** Trivial.

Key:

Linear-time partitioning subroutine.

Plan

Partitioning Subroutine

procedure PARTITION(A, p, r)
    $x \leftarrow A[r]$
    $i \leftarrow p - 1$
    for $j \leftarrow p$ **to** $r - 1$ **do**
        if $A[j] \leq x$ **then**
            $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
        end if
    end for
    exchange $A[i + 1] \leftrightarrow A[r]$
    return $i + 1$
end procedure

Partitioning Subroutine

procedure PARTITION(A, p, r)
  $x \leftarrow A[r]$
  $i \leftarrow p - 1$
  for $j \leftarrow p$ **to** $r - 1$ **do**
    if $A[j] \leq x$ **then**
      $i \leftarrow i + 1$
      exchange $A[i] \leftrightarrow A[j]$
    end if
  end for
  exchange $A[i + 1] \leftrightarrow A[r]$
  return $i + 1$
end procedure

Running time:

$O(n)$ for n elements.

Partitioning Subroutine

procedure PARTITION(A, p, r)
    $x \leftarrow A[r]$
    $i \leftarrow p - 1$
    for $j \leftarrow p$ **to** $r - 1$ **do**
        if $A[j] \leq x$ **then**
            $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
        end if
    end for
    exchange $A[i + 1] \leftrightarrow A[r]$
    return $i + 1$
end procedure

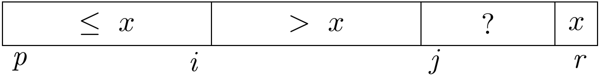
Running time:

$O(n)$ for n elements.

Invariants:

$\leq x$	$> x$	?	x
p i		j	r

Loop Invariants: Definition



At the beginning of each iteration of the loop (lines 4–9), for any array index k:

1. **Low side:** If $p \leq k \leq i$, then $A[k] \leq x$.
2. **High side:** If $i + 1 \leq k \leq j - 1$, then $A[k] \geq x$.
3. **Pivot:** If $k = r$, then $A[k] = x$.

These conditions define the partitioning into three regions:

- Elements $\leq x$ (low side).
- Elements $> x$ (high side).
- The pivot element.

Initialization and Maintenance

Initialization:

- Before first iteration: $i = p - 1$, $j = p$.
- No values yet examined, so invariants hold trivially.
- Line 2 ensures pivot condition (3) holds.

Maintenance:

- If $A[j] > x$: only increment j, preserving high side property.
- If $A[j] \leq x$: increment i, swap $A[i]$ and $A[j]$, then increment j.
- Swapping ensures $A[i] \leq x$ and $A[i+1] \ldots A[j-1] > x$.

Termination and Correctness

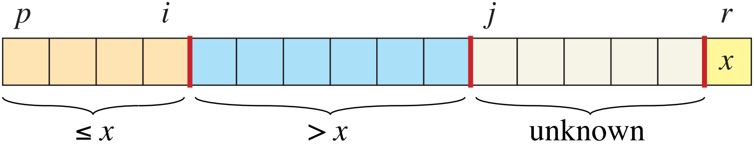
Termination:

- Loop ends when $j = r$.
- The unexamined subarray $A[j \dots r-1]$ is empty.
- All entries are in one of the three invariant regions.

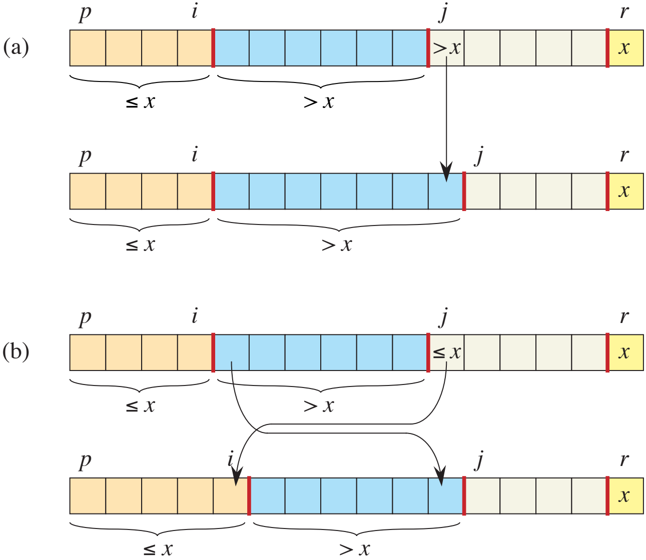
Correctness:

- Array is partitioned into:
  1. Elements $\leq x$ (low side).
  2. Elements $> x$ (high side).
- Pivot is placed immediately after the low side.

Regions in the Partition Procedure



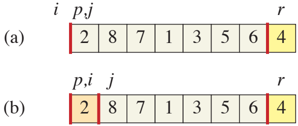
Handling Cases During Partition



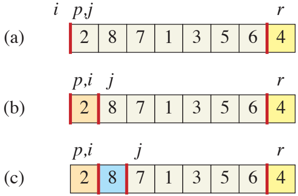
Example of partitioning

	i	p,j							r
(a)		2	8	7	1	3	5	6	4

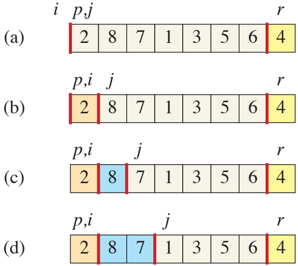
Example of partitioning



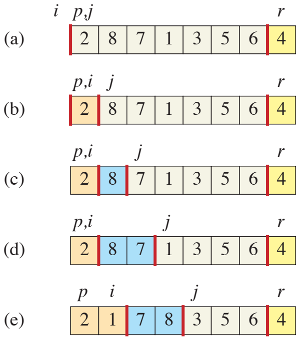
Example of partitioning



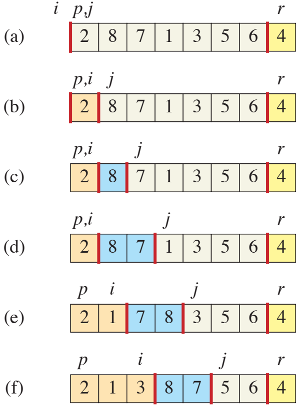
Example of partitioning



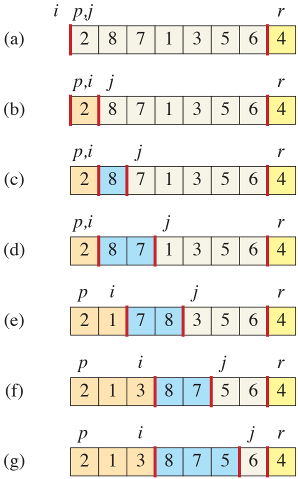
Example of partitioning



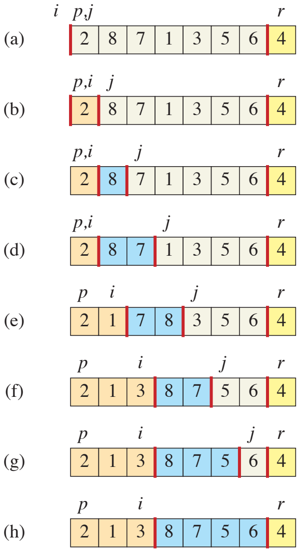
Example of partitioning



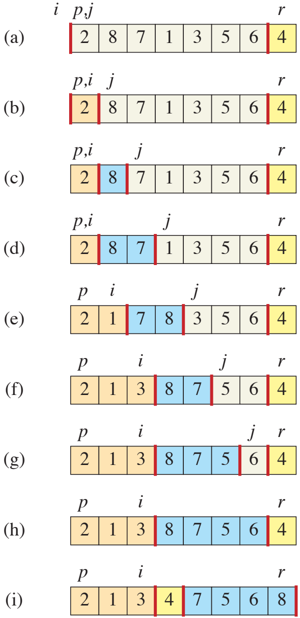
Example of partitioning



Example of partitioning



Example of partitioning



Pseudocode for Quicksort

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A,p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure
```

Pseudocode for Quicksort

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q ← PARTITION(A,p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure
```

Initial call:

QUICKSORT(A, 1, n)

Plan

Performance of Quicksort

- Assume all input elements are distinct.

Performance of Quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.

Performance of Quicksort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.

Worst-case of Quicksort

- Input sorted or reverse sorted.

Worst-case of Quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.

Worst-case of Quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

Worst-case of Quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Worst-case of Quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned} T(n) =& T(0) + T(n-1) + \Theta(n) \\ =& \Theta(1) + T(n-1) + \Theta(n) \\ =& T(n-1) + \Theta(n) \\ =& \Theta(n^2) \end{aligned}$$

Arithmetic Series!

Worst-case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$

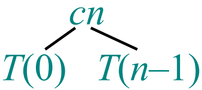
Worst-case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$

$T(n)$

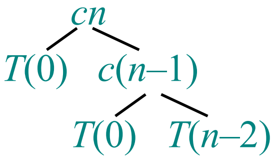
Worst-case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$



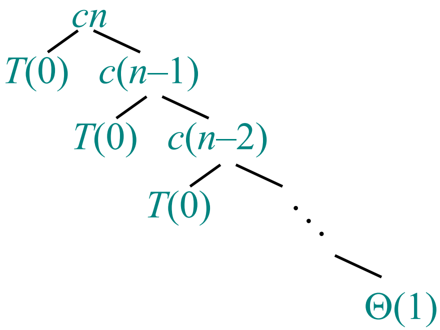
Worst-case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$



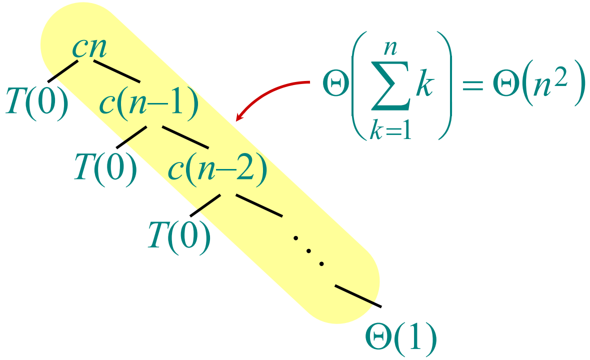
Worst-case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$



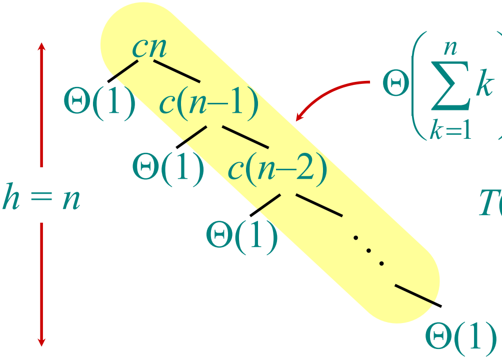
Worst-case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$



Worst-case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(n^2\right)$$

$$T(n) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Plan

Best-case Performance

For intuition only!

If we're lucky, PARTITION splits the array evenly:

Best-case Performance

For intuition only!

If we're lucky, PARTITION splits the array evenly:

$$
\begin{aligned}
T(n) =& 2T\left(\frac{n}{2}\right) + \Theta(n) \\
=& \Theta(n \lg n)
\end{aligned}
$$

Best-case Performance

For intuition only!

If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) = & 2T\left(\frac{n}{2}\right) + \Theta(n) \\ = & \Theta(n \lg n) \end{aligned}$$

- Same as merge-sort.

Best-case Performance

For intuition only!

If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) =& 2T\left(\frac{n}{2}\right) + \Theta(n) \\ =& \Theta(n \lg n) \end{aligned}$$

- Same as merge-sort.
- Which case is this?

Best-case Performance

For intuition only!

If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) =& 2T\left(\frac{n}{2}\right) + \Theta(n) \\ =& \Theta(n \lg n) \end{aligned}$$

- Same as merge-sort.
- Which case is this? Case 2.

Best-case Performance

For intuition only!

If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) =& 2T\left(\frac{n}{2}\right) + \Theta(n) \\ =& \Theta(n \lg n) \end{aligned}$$

- Same as merge-sort.
- Which case is this? Case 2.

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

Best-case Performance

For intuition only!

If we're lucky, PARTITION splits the array evenly:

$$\begin{aligned} T(n) =& 2T\left(\frac{n}{2}\right) + \Theta(n) \\ =& \Theta(n \lg n) \end{aligned}$$

- Same as merge-sort.
- Which case is this? Case 2.

What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T\left(\frac{1}{10}n\right) + T\left(\frac{9}{10}n\right) + \Theta(n)$$

What is the solution to this recurrence?

Performance of “Almost-best” Case

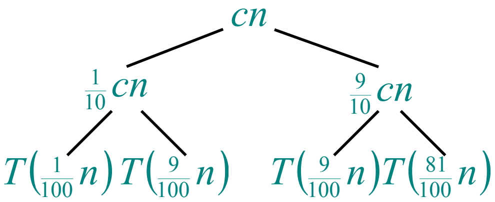
$$T(n)$$

Performance of “Almost-best” Case

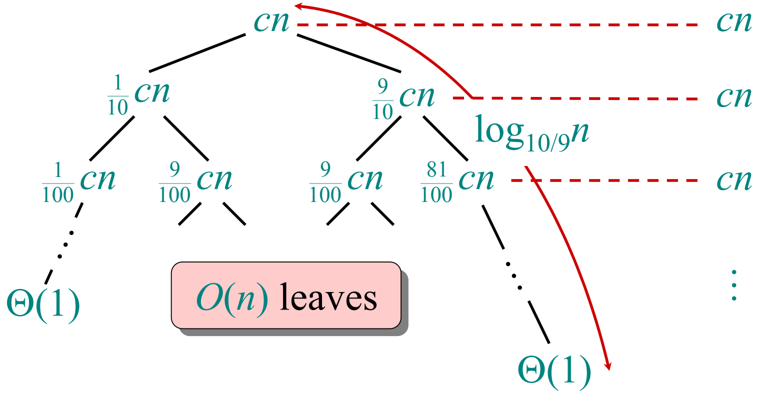
$$cn$$

$$T\left(\tfrac{1}{10}n\right) \qquad\qquad T\left(\tfrac{9}{10}n\right)$$

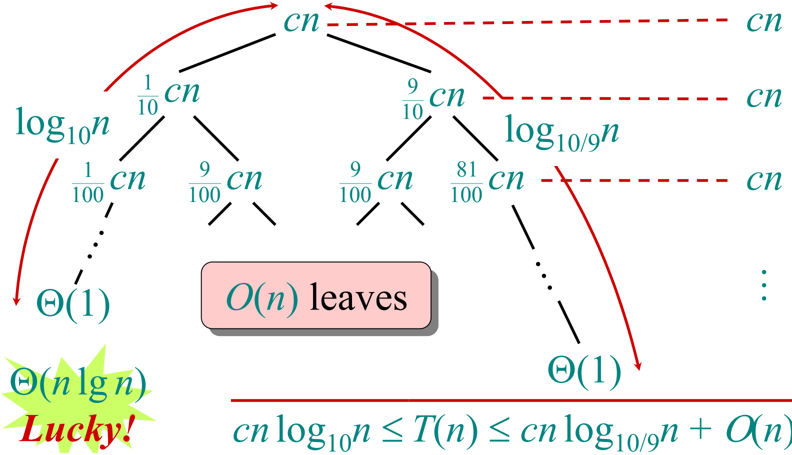
Performance of "Almost-best" Case



Performance of "Almost-best" Case



Performance of "Almost-best" Case



$$cn \log_{10} n \le T(n) \le cn \log_{10/9} n + O(n)$$

More Intuition

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ...

$$L(n) = 2U\left(\frac{n}{2}\right) + \Theta(n) \qquad \textit{lucky}$$

$$U(n) = L\left(n-1\right) + \Theta(n) \qquad \textit{unlucky}$$

Solving:

$$\begin{aligned} L(n) &= 2\left(L\left(\frac{n}{2}-1\right) + \Theta(\frac{n}{2})\right) + \Theta(n) \\ &= 2L\left(\frac{n}{2}-1\right) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \textbf{Lucky!} \end{aligned}$$

How can we make sure we are usually lucky?

Plan

Randomized Quicksort

IDEA:
Partition around a **random** element.

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

Plan

Randomized Quicksort Analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the **indicator random variable**.

$$X_k = \begin{cases} 1 & \text{if } \textsc{Partition} \text{ generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

Randomized Quicksort Analysis

Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.

For $k = 0, 1, \ldots, n-1$, define the **indicator random variable**.

$$X_k = \begin{cases} 1 & \text{if } \textsc{Partition} \text{ generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = 0 \cdot Pr\{X_k = 0\} + 1 \cdot Pr\{X_k = 1\} = Pr\{X_k = 1\} = \frac{1}{n}$, since all splits are equally likely, assuming element are distinct.

Analysis (Cont.)

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \quad \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split.} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))$$

Plan

Calculating expectation

Take expectations of both sides.

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right]$$

Calculating expectation

Linearity of expectation.

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E\left[X_k(T(k) + T(n-k-1) + \Theta(n))\right]
\end{aligned}
$$

Calculating expectation

Independence of X_k from other random choices.

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E\left[X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]
\end{aligned}
$$

Calculating expectation

$E[X_k] = \frac{1}{n}$.

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E\left[X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\
&= \frac{1}{n} \sum_{k=0}^{n-1} E[T(k) + T(n-k-1) + \Theta(n)]
\end{aligned}
$$

Calculating expectation

Linearity of expectation.

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E\left[X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)
\end{aligned}
$$

Calculating expectation

Summations have identical terms.

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E\left[X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n) \\
&= \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)
\end{aligned}
$$

Calculating expectation

$n \cdot \Theta(n) = \Theta(n^2)$.

$$\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E\left[X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n) \\
&= \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\Theta(n^2)
\end{aligned}$$

Calculating expectation

Sum up.

$$
\begin{aligned}
E[T(n)] &= E\left[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E\left[X_k(T(k) + T(n-k-1) + \Theta(n))\right] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)] \\
&= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n) \\
&= \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + \Theta(n)
\end{aligned}
$$

Hairy recurrence

$$E[T(n)] = \frac{2}{n} \sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0$, 1 terms can be absorbed in the $\Theta(n)$.)

Prove:
$E[T(n)] \le an \lg n$ for constant $a > 0$.

- Choose a large enough so that $an \lg n$ dominates $E[T(n)]$ for sufficiently small $n \ge 2$.

Use fact:
$\sum_{k=2}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ (exercise).

Substitution method

Substitute inductive hypothesis.

$$E[T(n)] \leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

Substitution method

Use fact.

$$
\begin{aligned}
E[T(n)] &\leq \frac{2}{n}\sum_{k=2}^{n-1} ak\lg k + \Theta(n) \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n^2\lg n - \frac{1}{8}n^2\right) + \Theta(n)
\end{aligned}
$$

Substitution method

$$
\begin{aligned}
E[T(n)] &\leq \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an \lg n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\leq an \lg n,
\end{aligned}
$$

if a is chosen large enough so that $\dfrac{an}{4}$ dominates the $\Theta(n)$.

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.

Quicksort in practice

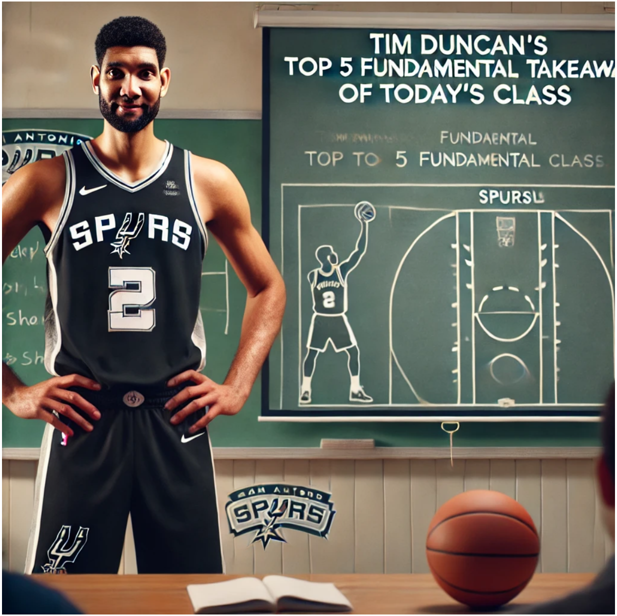
- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from **code tuning**.

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from **code tuning**.
- Quicksort behaves well even with caching and virtual memory.

End of Lecture 4.

TDT5FTOTC


TIM DUNCAN'S
TOP 5 FUNDAMENTAL TAKEAW
OF TODAY'S CLASS
FUNDAENTAL
TOP TO 5 FUNDAMENTAL CLASS
SPURSL
SPURS
2
SPURS

Top 5 Fundamental Takeaways

Top 5 Fundamental Takeaways

5. **Quicksort is a Divide-and-Conquer Algorithm** – It recursively partitions an array around a pivot and sorts the subarrays efficiently.

Top 5 Fundamental Takeaways

5 **Quicksort is a Divide-and-Conquer Algorithm** – It recursively partitions an array around a pivot and sorts the subarrays efficiently.

4 **Partitioning is the Core of Quicksort** – The partitioning step ensures elements are correctly placed around the pivot in $O(n)$ time.

Top 5 Fundamental Takeaways

5 **Quicksort is a Divide-and-Conquer Algorithm** – It recursively partitions an array around a pivot and sorts the subarrays efficiently.

4 **Partitioning is the Core of Quicksort** – The partitioning step ensures elements are correctly placed around the pivot in $O(n)$ time.

3 **Best-case and Worst-case Analysis** – Quicksort runs in $O(n \log n)$ in the best case but can degrade to $O(n^2)$ if poorly partitioned.

Top 5 Fundamental Takeaways

5 **Quicksort is a Divide-and-Conquer Algorithm** – It recursively partitions an array around a pivot and sorts the subarrays efficiently.

4 **Partitioning is the Core of Quicksort** – The partitioning step ensures elements are correctly placed around the pivot in $O(n)$ time.

3 **Best-case and Worst-case Analysis** – Quicksort runs in $O(n \log n)$ in the best case but can degrade to $O(n^2)$ if poorly partitioned.

2 **Randomized Quicksort Helps Avoid Worst-case Behavior** – Choosing a random pivot prevents consistently bad splits and ensures an expected $O(n \log n)$ runtime.

Top 5 Fundamental Takeaways

5 **Quicksort is a Divide-and-Conquer Algorithm** – It recursively partitions an array around a pivot and sorts the subarrays efficiently.

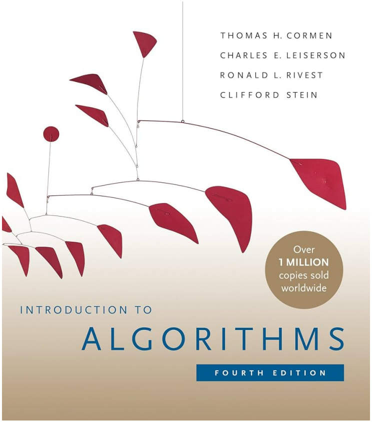
4 **Partitioning is the Core of Quicksort** – The partitioning step ensures elements are correctly placed around the pivot in $O(n)$ time.

3 **Best-case and Worst-case Analysis** – Quicksort runs in $O(n \log n)$ in the best case but can degrade to $O(n^2)$ if poorly partitioned.

2 **Randomized Quicksort Helps Avoid Worst-case Behavior** – Choosing a random pivot prevents consistently bad splits and ensures an expected $O(n \log n)$ runtime.

1 **Quicksort is Highly Efficient in Practice** – It outperforms merge sort in most cases and benefits from hardware optimizations.

Introduction to Algorithms



Content has been extracted from *Introduction to Algorithms*, Fourth Edition, by Cormen, Leiserson, Rivest, and Stein. MIT Press. 2022.
Visit https://mitpress.mit.edu/9780262046305/introduction-to-algorithms/.
Original slides from *Introduction to Algorithms 6.046J/18.401J*, Fall 2005 Class by Prof. Charles Leiserson and Prof. Erik Demaine. MIT OpenCourseWare Initiative available at https://ocw.mit.edu/courses/6-046j-introduction-to-algorithms-sma-5503-fall-2005/.