CS3230 – Design and Analysis of Algorithms (S2 AY2024/25)

Lecture 4b: Average-Case Analysis of Quick Sort

Quick sort

- **Input:** an array $A[1..n]$ of n elements.

- **Partition:**
  - Select a number in $A[1..n]$ as the **pivot**.
  - Rearrange the array to satisfy the condition:

$$A = [\underbrace{\ldots\ldots\ldots\ldots\ldots\ldots}_{\forall x \in A_S,\, x \leq \text{pivot}}^{A_S}\ \text{pivot}\ \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots}_{\forall x \in A_L,\, x \geq \text{pivot}}^{A_L}]$$

- **Recursion:**
  - Recursively sort A_S and A_L.

Quick sort

- **Input:** an array $A[1..n]$ of n elements.

There are various ways to implement this part.

- **Partition:**
  - Select a number in $A[1..n]$ as the **pivot**.
  - Rearrange the array to satisfy the condition:

$$A = [\underbrace{\ldots\ldots\ldots\ldots\ldots\ldots}_{\forall x \in A_S,\, x \le \text{pivot}}^{A_S}\ \text{pivot}\ \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots}_{\forall x \in A_L,\, x \ge \text{pivot}}^{A_L}]$$

- **Recursion:**
  - Recursively sort A_S and A_L.

VisuAlgo (Quick sort): https://visualgo.net/en/sorting?mode=Quick

Quick sort

It is common to choose the <u>first element</u> as the pivot: **pivot** $\leftarrow A[1]$.

- **Input:** an array $A[1..n]$ of n elements.

- **Partition:**
  - Select a number in $A[1..n]$ as the **pivot**.
  - Rearrange the array to satisfy the condition:

$$A = [\overbrace{\dots\dots\dots\dots\dots\dots}^{A_S}\ \text{pivot}\ \overbrace{\dots\dots\dots\dots\dots\dots}^{A_L}]$$

$$\forall x \in A_S, x \leq \text{pivot} \qquad \forall x \in A_L, x \geq \text{pivot}$$

- **Recursion:**
  - Recursively sort A_S and A_L.

VisuAlgo (Quick sort): https://visualgo.net/en/sorting?mode=Quick

Quick sort

- **Input:** an array $A[1..n]$ of n elements.

- **Partition:**
  - Select a number in $A[1..n]$ as the **pivot**.
  - Rearrange the array to satisfy the condition:

$$A = [\underbrace{\ldots\ldots\ldots\ldots\ldots\ldots}^{A_S}\ \text{pivot}\ \underbrace{\ldots\ldots\ldots\ldots\ldots\ldots}^{A_L}]$$

$\forall x \in A_S, x \leq \text{pivot}$ $\qquad$ $\forall x \in A_L, x \geq \text{pivot}$

$\Theta(n)$ time

This step requires comparing **pivot** and all other elements.

- **Recursion:**
  - Recursively sort A_S and A_L.

An example.

VisuAlgo (Quick sort): https://visualgo.net/en/sorting?mode=Quick

Quick sort $T(n)$ time

- **Input:** an array $A[1..n]$ of n elements.

- **Partition:**
  - Select a number in $A[1..n]$ as the **pivot**.
  - Rearrange the array to satisfy the condition:

$$A = [\underbrace{\dots\dots\dots\dots\dots\dots}_{A_S}\ \text{pivot}\ \underbrace{\dots\dots\dots\dots\dots\dots}_{A_L}]$$

$\forall x \in A_S, x \leq \text{pivot}$ $\qquad$ $\forall x \in A_L, x \geq \text{pivot}$

$\Theta(n)$ time

- **Recursion:**
  - Recursively sort A_S and A_L.

$T(j-1) + T(n-j)$ time, if **pivot** is $\underbrace{\text{the } j\text{th smallest element.}}_{\text{Assume that all elements are distinct.}}$

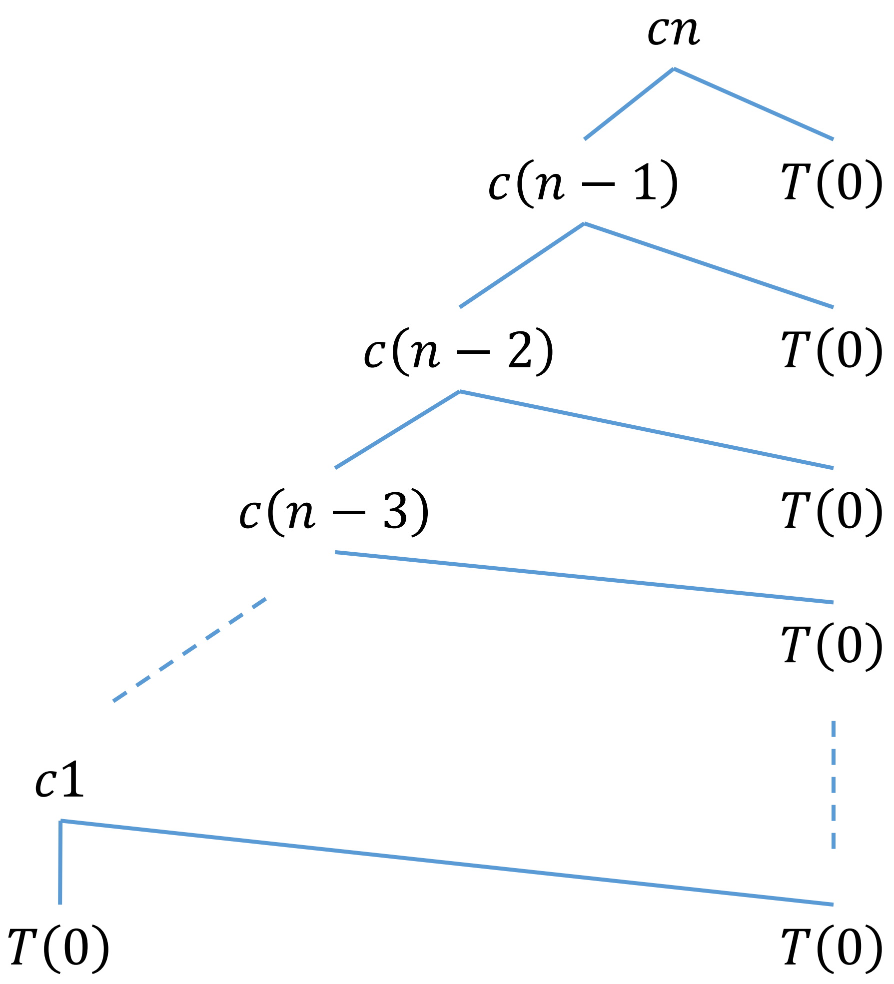
VisuAlgo (Quick sort): https://visualgo.net/en/sorting?mode=Quick

Recurrence

- Suppose **pivot** is the jth smallest element.
    - $T(n) = T(j-1) + T(n-j) + cn$

Worst-case running time

- Suppose **pivot** is the jth smallest element.
    - $T(n) = T(j-1) + T(n-j) + cn$
- **Intuition:** Worst case seems to be $j = 1$ or $j = n$.
    - $T(n) = T(0) + T(n-1) + cn \in \Theta(n^2)$
    - $T(n) \in \Theta(n^2)$



A more formal proof

- Suppose **pivot** is the jth smallest element.
  - $T(n) = T(j-1) + T(n-j) + cn$

- **Goal:** $\boxed{T(n) = \max_{j\in[n]}\{T(j-1) + T(n-j) + cn\}}$ ▷ $\boxed{T(n) \in O(n^2)}$

- Guess $T(r) \leq c_1 r^2$ and prove it by induction.
- **Base case:** $T(0) = 0.$ Just simply not invoke any recursive call with $n = 0$.

A more formal proof

- Suppose **pivot** is the jth smallest element.
  - $T(n) = T(j-1) + T(n-j) + cn$

- **Goal:** $\boxed{T(n) = \max_{j \in [n]} \{T(j-1) + T(n-j) + cn\}}$ $\triangleright$ $\boxed{T(n) \in O(n^2)}$

- Guess $T(r) \leq \; c_1 r^2$ and prove it by induction.
- **Base case:** $T(0) = 0$.
- **Inductive step:** $(n \geq 1)$

$$
\begin{aligned}
T(n) &= \max_{j \in [n]} \{T(j-1) + T(n-j) + cn\} \\
&\leq \max_{j \in [n]} \{c_1(j^2 - 2j + 1 + n^2 - 2nj + j^2) + cn\} \\
&= \max_{j \in [n]} \{c_1\left(n^2 + 1 \; - 2j(n + 1 - j)\right) + cn\}
\end{aligned}
$$

A more formal proof

- Suppose **pivot** is the jth smallest element.
  - $T(n) = T(j-1) + T(n-j) + cn$

- **Goal:** $\boxed{T(n) = \max_{j \in [n]}\{T(j-1) + T(n-j) + cn\}}$ $\triangleright$ $\boxed{T(n) \in O(n^2)}$

- Guess $T(r) \leq c_1 r^2$ and prove it by induction.
- **Base case:** $T(0) = 0$.
- **Inductive step:** $(n \geq 1)$

$$
\begin{aligned}
T(n) &= \max_{j \in [n]}\{T(j-1) + T(n-j) + cn\} \\
&\leq \max_{j \in [n]}\{c_1(j^2 - 2j + 1 + n^2 - 2nj + j^2) + cn\} \\
&= \max_{j \in [n]}\left\{c_1\left(n^2 + 1 - 2j(n+1-j)\right) + cn\right\} \\
&\leq c_1(n^2 - 2n + 1) + cn = c_1 n^2 + cn - c_1(2n-1) \leq \cdots
\end{aligned}
$$

$2j(n+1-j)$ is smallest when $j = 1$ or $j = n$.

A more formal proof

- Suppose **pivot** is the jth smallest element.
  - $T(n) = T(j-1) + T(n-j) + cn$

- **Goal:** $\boxed{T(n) = \max_{j \in [n]}\{T(j-1) + T(n-j) + cn\}}$ $\triangleright$ $\boxed{T(n) \in O(n^2)}$

- Guess $T(r) \leq c_1 r^2$ and prove it by induction.
- **Base case:** $T(0) = 0$.
- **Inductive step:** $(n \geq 1)$

$$
\begin{aligned}
T(n) &= \max_{j \in [n]}\{T(j-1) + T(n-j) + cn\} \\
&\leq \max_{j \in [n]}\{c_1(j^2 - 2j + 1 + n^2 - 2nj + j^2) + cn\} \\
&= \max_{j \in [n]}\{c_1\big(n^2 + 1 - 2j(n + 1 - j)\big) + cn\} \\
&\leq c_1(n^2 - 2n + 1) + cn = c_1 n^2 + cn - c_1(2n-1) \leq c_1 n^2
\end{aligned}
$$

Select c_1 so that $\forall (n \geq 1), cn \leq c_1(2n-1)$.

Average-case analysis

- The worst-case bound $T(n) \in \Theta(n^2)$ does not capture the typical performance of quick sort.
- **Next:** average-case analysis.

Average-case analysis

- The worst-case bound $T(n) \in \Theta(n^2)$ does not capture the typical performance of quick sort.
- **Next:** average-case analysis.

- Assume all numbers are distinct.
- Let $a_1 < a_2 < \cdots < a_n$ be the input numbers in the sorted order.
- Fixing $(a_1, a_2, \dots, a_n)$, the input array A can be described by a **permutation** π of $(a_1, a_2, \dots, a_n)$.

Average-case analysis

- The worst-case bound $T(n) \in \Theta(n^2)$ does not capture the typical performance of quick sort.
- **Next:** average-case analysis.

- Assume all numbers are distinct.
- Let $a_1 < a_2 < \cdots < a_n$ be the input numbers in the sorted order.
- Fixing $(a_1, a_2, \ldots, a_n)$, the input array A can be described by a **permutation** π of $(a_1, a_2, \ldots, a_n)$.

- The execution of the quick sort algorithm:
  - It depends only on π.
  - It is independent of the actual values of $(a_1, a_2, \ldots, a_n)$.

Quick sort is **comparison-based**.

Average-case analysis

- The **average-case** running time $A(n)$ is the average running time over all inputs of size n.

There are $n!$ permutations of $(a_1, a_2, \dots, a_n)$.

$$A(n) = \sum_{\pi} \frac{1}{n!} \cdot (\text{running time of quick sort on } \pi)$$

The summation is over all permutations π of $(a_1, a_2, \dots, a_n)$.

The execution of the quick sort algorithm:

- It depends only on π.
- It is independent of the actual values of $(a_1, a_2, \dots, a_n)$.

Average-case analysis

Observation: $A(n)$ is also the expected running time when the permutation π is chosen uniformly at random.

- The **average-case** running time $A(n)$ is the average running time over all inputs of size n.

Each permutation is chosen with a probability of $\frac{1}{n!}$.

$$A(n) = \sum_{\pi} \frac{1}{n!} \cdot (\text{running time of quick sort on } \pi)$$

The summation is over all permutations π of $(a_1, a_2, \ldots, a_n)$.

The execution of the quick sort algorithm:

- It depends only on π.
- It is independent of the actual values of $(a_1, a_2, \ldots, a_n)$.

Uniformity

If **pivot** $= a_j$, then the elements in the two recursive calls are as follows:

- A_S: $a_1, a_2, \dots, a_{j-1}$
- A_L: $a_{j+1}, a_{j+2}, \dots, a_n$

Suppose the input permutation π of $(a_1, a_2, \dots, a_n)$ is uniformly random.

Observation 1: The **pivot** is selected uniformly at random.

- $\forall (j \in [n]), \Pr[\textbf{pivot} = a_j] = \frac{1}{n}$.

Uniformity

If **pivot** $= a_j$, then the elements in the two recursive calls are as follows:

- A_S: $a_1, a_2, \dots, a_{j-1}$
- A_L: $a_{j+1}, a_{j+2}, \dots, a_n$

Suppose the input permutation π of $(a_1, a_2, \dots, a_n)$ is uniformly random.

Observation 1: The **pivot** is selected uniformly at random.

- $\forall (j \in [n]), \Pr[\textbf{pivot} = a_j] = \frac{1}{n}$.

Reason:

- The **pivot** is selected as the first element: **pivot** $\leftarrow A[1]$.
- If π is uniformly random, then each element has equal chance to be the first element.

Uniformity

If **pivot** $= a_j$, then the elements in the two recursive calls are as follows:

- $A_S: a_1, a_2, \dots, a_{j-1}$
- $A_L: a_{j+1}, a_{j+2}, \dots, a_n$

Suppose the input permutation π of $(a_1, a_2, \dots, a_n)$ is uniformly random.

Observation 1: The **pivot** is selected uniformly at random.

- $\forall (j \in [n]), \Pr[\textbf{pivot} = a_j] = \frac{1}{n}$.

Observation 2: The permutations for both recursive calls are also uniformly random.

- Recursive call on A_S: Each permutation of $(a_1, a_2, \dots, a_{j-1})$ appears with equal probability.
- Recursive call on A_L: Each permutation of $(a_{j+1}, a_{j+2}, \dots, a_n)$ appears with equal probability.

Uniformity

If **pivot** $= a_j$, then the elements in the two recursive calls are as follows:

- $A_S: a_1, a_2, \ldots, a_{j-1}$
- $A_L: a_{j+1}, a_{j+2}, \ldots, a_n$

Suppose the input permutation π of $(a_1, a_2, \ldots, a_n)$ is uniformly random.

Observation 1: The **pivot** is selected uniformly at random.

- $\forall (j \in [n]), \Pr[\textbf{pivot} = a_j] = \frac{1}{n}$.

Observation 2: The permutations for both recursive calls are also uniformly random.

- Recursive call on A_S: Each permutation of $(a_1, a_2, \ldots, a_{j-1})$ appears with equal probability.
- Recursive call on A_L: Each permutation of $(a_{j+1}, a_{j+2}, \ldots, a_n)$ appears with equal probability.

Reason:

- If **pivot** $= a_j$, then the partition algorithm never compares any two elements in $(a_1, a_2, \ldots, a_{j-1})$.

Uniformity

If **pivot** $= a_j$, then the elements in the two recursive calls are as follows:

- $A_S: a_1, a_2, \dots, a_{j-1}$
- $A_L: a_{j+1}, a_{j+2}, \dots, a_n$

Suppose the input permutation π of $(a_1, a_2, \dots, a_n)$ is uniformly random.

Observation 1: The **pivot** is selected uniformly at random.

- $\forall (j \in [n]), \Pr[\textbf{pivot} = a_j] = \frac{1}{n}$.

Observation 2: The permutations for both recursive calls are also uniformly random.

- Recursive call on A_S: Each permutation of $(a_1, a_2, \dots, a_{j-1})$ appears with equal probability.
- Recursive call on A_L: Each permutation of $(a_{j+1}, a_{j+2}, \dots, a_n)$ appears with equal probability. ⟵ Similar

Reason:

- If **pivot** $= a_j$, then the partition algorithm never compares any two elements in $(a_1, a_2, \dots, a_{j-1})$.

At the start, the input permutation π restricted to $(a_1, a_2, \dots, a_{j-1})$ is uniformly random.

$\triangleright$

At the end, the permutation of $(a_1, a_2, \dots, a_{j-1})$ is still uniformly random.

Uniformity

- Suppose $X = (x_1, x_2, x_3)$ is a uniformly random permutation of $(1, 2, 3)$.

$X = (x_1, x_2, x_3)$

$\downarrow$ Swap x_2 and x_3.

Still uniformly random:
- $(1,2,3) \rightarrow (1,3,2)$
- $(1,3,2) \rightarrow (1,2,3)$
- $(2,1,3) \rightarrow (2,3,1)$
- $(2,3,1) \rightarrow (2,1,3)$
- $(3,1,2) \rightarrow (3,2,1)$
- $(3,2,1) \rightarrow (3,1,2)$

No comparison is made.

$X = (x_1, x_2, x_3)$

$\downarrow$ Swap x_2 and x_3 if $x_2 > x_3$.

Not uniformly random:
- $(1,2,3) \rightarrow (1,2,3)$
- $(1,3,2) \rightarrow (1,2,3)$
- $(2,1,3) \rightarrow (2,1,3)$
- $(2,3,1) \rightarrow (2,1,3)$
- $(3,1,2) \rightarrow (3,1,2)$
- $(3,2,1) \rightarrow (3,1,2)$

A comparison is made.

Recurrence

If **pivot** $= a_j$, then the elements in the two recursive calls are as follows:

- A_S: $a_1, a_2, \dots, a_{j-1}$
- A_L: $a_{j+1}, a_{j+2}, \dots, a_n$

Suppose the input permutation π of $(a_1, a_2, \dots, a_n)$ is uniformly random.

Observation 1: The **pivot** is selected uniformly at random.

Observation 2: The permutations for both recursive calls are also uniformly random.

Recall: $A(n)$ is the expected running time when the permutation π is chosen uniformly at random.

$$A(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [A(j-1) + A(n-j) + cn]$$

$\forall (j \in [n]), \Pr[\textbf{pivot} = a_j] = \frac{1}{n}$

The cost to perform the partition.

Conditioning on **pivot** $= a_j$, the expected running time of the two recursive calls.

Solving the recurrence

$$A(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [A(j-1) + A(n-j) + cn] = cn + \frac{2}{n} \cdot \sum_{j=0}^{n-1} A(j)$$

Solving the recurrence

$$A(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [A(j-1) + A(n-j) + cn] = cn + \frac{2}{n} \cdot \sum_{j=0}^{n-1} A(j)$$

$\downarrow$

- $n \cdot A(n) = cn^2 + 2 \cdot \sum_{j=0}^{n-1} A(j)$
- $(n-1) \cdot A(n-1) = c(n-1)^2 + 2 \cdot \sum_{j=0}^{n-2} A(j)$

Solving the recurrence

$$A(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [A(j-1) + A(n-j) + cn] = cn + \frac{2}{n} \cdot \sum_{j=0}^{n-1} A(j)$$

- $n \cdot A(n) = cn^2 + 2 \cdot \sum_{j=0}^{n-1} A(j)$
- $(n-1) \cdot A(n-1) = c(n-1)^2 + 2 \cdot \sum_{j=0}^{n-2} A(j)$

- $n \cdot A(n) - (n-1) \cdot A(n-1) = c(2n-1) + 2A(n-1)$

Solving the recurrence

$$A(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [A(j-1) + A(n-j) + cn] = cn + \frac{2}{n} \cdot \sum_{j=0}^{n-1} A(j)$$

- $n \cdot A(n) = cn^2 + 2 \cdot \sum_{j=0}^{n-1} A(j)$
- $(n-1) \cdot A(n-1) = c(n-1)^2 + 2 \cdot \sum_{j=0}^{n-2} A(j)$

- $n \cdot A(n) - (n-1) \cdot A(n-1) = c(2n-1) + 2A(n-1)$
- $n \cdot A(n) - (n+1) \cdot A(n-1) = \big(n \cdot A(n) - (n-1) \cdot A(n-1)\big) - 2A(n-1) = c(2n-1)$

Solving the recurrence

$$A(n) = \frac{1}{n} \cdot \sum_{j=1}^{n} [A(j-1) + A(n-j) + cn] = cn + \frac{2}{n} \cdot \sum_{j=0}^{n-1} A(j)$$

- $n \cdot A(n) = cn^2 + 2 \cdot \sum_{j=0}^{n-1} A(j)$
- $(n-1) \cdot A(n-1) = c(n-1)^2 + 2 \cdot \sum_{j=0}^{n-2} A(j)$

- $n \cdot A(n) - (n-1) \cdot A(n-1) = c(2n-1) + 2A(n-1)$
- $n \cdot A(n) - (n+1) \cdot A(n-1) = \big(n \cdot A(n) - (n-1) \cdot A(n-1)\big) - 2A(n-1) = c(2n-1)$

Dividing by $n(n+1)$

$$\frac{A(n)}{n+1} - \frac{A(n-1)}{n} = \frac{c(2n-1)}{n(n+1)} < \frac{c(2n+2)}{n(n+1)} = \frac{2c}{n}$$

Solving the recurrence

$$\frac{A(n)}{n+1} < 2c \cdot \underbrace{\left(\frac{1}{n} + \frac{1}{n-1} + \frac{1}{n-2} + \cdots + \frac{1}{2}\right)}_{O(\log n)} + \underbrace{\frac{A(1)}{2}}_{O(1)}$$

$$A(n) \in O(n \log n)$$

$$\frac{A(n)}{n+1} - \frac{A(n-1)}{n} < \frac{2c}{n}$$

$$\frac{A(n-1)}{n} - \frac{A(n-2)}{n-1} < \frac{2c}{n-1}$$

$$\frac{A(n-2)}{n-1} - \frac{A(n-3)}{n-2} < \frac{2c}{n-2}$$

$$\vdots$$

$$\frac{A(2)}{3} - \frac{A(1)}{2} < \frac{2c}{2}$$

Question

Who is the **Master of Algorithms** pictured below?

- Tony Hoare
- John Hopcroft
- Ronald Rivest
- Andrew Yao

Answer

Tony Hoare

- 1980 Turing Award.
- Inventor of **quick sort** and **quick select**.
- He has made foundational contributions to programming languages, algorithms, operating systems, formal verification, and concurrent computing.

Source: https://en.wikipedia.org/wiki/Tony_Hoare

Desirable properties of sorting algorithms

- Small running time:
  - Worst case.
  - Average case.
- Comparison-based algorithms.
- What else?

Stable sorting

- **Stable** sorting algorithm:
  - For elements of equal values, the original ordering is preserved.
  - If $A[i] = A[j]$ and $i < j$, then $A[i]$ must be before $A[j]$ in the output.

Stable sorting

- **Stable** sorting algorithm:
  - For elements of equal values, the original ordering is preserved.
  - If $A[i] = A[j]$ and $i < j$, then $A[i]$ must be before $A[j]$ in the output.

- Insertion sort is stable.
- Merge sort is stable if implemented properly.
- Most of the implementations of quick sort are not stable.

In-place sorting

- A sorting algorithm is **in-place** if it uses very little extra memory besides the input array.

In-place sorting

- A sorting algorithm is **in-place** if it uses very little extra memory besides the input array.

- Insertion sort uses only $O(1)$ extra memory.
- Merge sort uses $O(n)$ extra memory.
- Quicksort uses $O(\log n)$ extra memory if implemented properly.

After partitioning, the sub-array with the fewer elements is recursively sorted first.

Desirable properties of sorting algorithms

- Small running time:
  - Worst case.
  - Average case.

- Additional desirable properties:
  - Comparison-based.
  - Stable.
  - In-place.

They are highly dependent on the specific way the algorithm is implemented.

https://en.wikipedia.org/wiki/Sorting_algorithm#Comparison_of_algorithms

Acknowledgement

- The slides are modified from previous editions of this course and similar course elsewhere.
- **List of credits:**
  - Surender Baswana
  - Arnab Bhattacharya
  - Diptarka Chakraborty
  - Yi-Jun Chang
  - Erik Demaine
  - Steven Halim
  - Sanjay Jain
  - Wee Sun Lee
  - Charles Leiserson
  - Hon Wai Leong
  - Warut Suksompong
  - Wing-Kin Sung