# Discrete Mathematics and Its Applications 2 (CS147)

***Lecture 15: Analysis of Randomized quick-sort**

**Fanghui Liu**

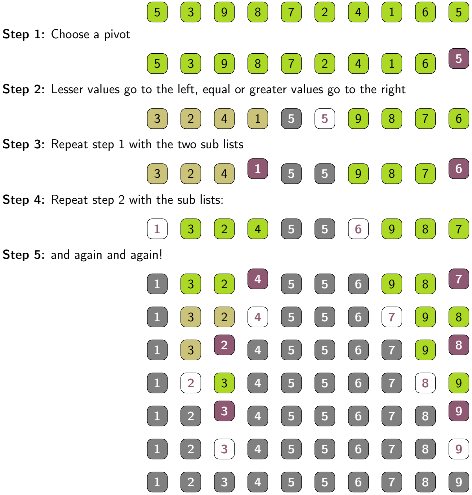
Department of Computer Science, University of Warwick, UK

# Recall Deterministic Quick-sort algorithm in Lecture 8

**Algorithm 1:** Deterministic Quicksort

**Input:** An array $A[1, 2, \ldots, n]$
**Output:** An sorted array $A[1, 2, \ldots, n]$
pivot $\leftarrow A[n]$ [always choose the rightmost element] ;
$S_{\text{smaller}} \leftarrow [], S_{\text{larger}} \leftarrow []$;
**for** $i = 1, \ldots, n$ **do**
  **if** $A[i] \leq$ *pivot* **then**
    $S_{\text{smaller.append}}(A[i])$;
  **end**
  **else** $S_{\text{larger.append}}(A[i])$;
**end**
**return** [Quicksort($S_{\text{smaller}}$), pivot, Quicksort($S_{\text{larger}}$) ];

worst case running time complexity $\Theta(n^2)$.



# Randomized Quick-sort algorithm

**Randomized!**

making the algorithm randomized gives us more control over the running time!

**Algorithm 2:** Randomized Quick-sort

**Input:** An array $A[1, 2, \dots, n]$
**Output:** An sorted array $A[1, 2, \dots, n]$

[randomly choose pivot uniformly] ;
$S_{\text{smaller}} \leftarrow [], S_{\text{larger}} \leftarrow []$;
**for** $i = 1, \dots, n$ **do**
  **if** $A[i] \le$ *pivot* **then**
    $S_{\text{smaller.append}}(A[i])$;
  **end**
  **else** $S_{\text{larger.append}}(A[i])$;
**end**
**return** [Quicksort($S_{\text{smaller}}$), pivot, Quicksort($S_{\text{larger}}$) ];

# Worst-case expected-time bound

- the worst case: $T(n) = \max_{\text{inputs } I \text{ of size } n} T(I)$
- the average case: $T(n) = \operatorname{avg}_{\text{inputs } I \text{ of size } n} T(I)$

# Worst-case expected-time bound

- the worst case: $T(n) = \max_{\text{inputs } I \text{ of size } n} T(I)$
- the average case: $T(n) = \underset{\text{inputs } I \text{ of size } n}{\text{avg}} T(I)$

**Remark:** 1) Merge-sort has both worst-case and average-case time $\Theta(n \log n)$, independent of the input.
2) for some algorithms, the running time depends on the input, e.g., Quick-sort.

## Worst-case expected-time bound

- the worst case: $T(n) = \max_{\text{inputs } I \text{ of size } n} T(I)$
- the average case: $T(n) = \operatorname{avg}_{\text{inputs } I \text{ of size } n} T(I)$

**Remark:** 1) Merge-sort has both worst-case and average-case time $\Theta(n \log n)$, independent of the input.
2) for some algorithms, the running time depends on the input, e.g., Quick-sort.

### Target: Worst-case expected-time bound

We will prove that, for **any** given input array $I$ of $n$ elements, the expected time of randomized quick-sort $\mathbb{E}[T(I)]$ is $\mathcal{O}(n \log n)$.

- This is worst-case expected-time bound, better than the average case w.r.t the requirement on the inputs

# Analysis via Recurrence

## Theorem (Recall: total expectation theorem)

*Given a probability space* $(\Omega, \mathcal{F}, \Pr)$, *consider a partition* $\{B_j\}_{j=1}^n$ *of* $\Omega$, *then the expectation of a random variable* $X$ *can be represented as*

$$\mathbb{E}(X) = \sum_{j=1}^{n} \mathbb{E}(X|B_j)\Pr(B_j)$$

## Analysis via Recurrence

### Theorem (Recall: total expectation theorem)

*Given a probability space* $(\Omega, \mathcal{F}, \mathrm{Pr})$, *consider a partition* $\{B_j\}_{j=1}^n$ *of* $\Omega$, *then the expectation of a random variable* $X$ *can be represented as*

$$\mathbb{E}(X) = \sum_{j=1}^{n} \mathbb{E}(X|B_j)\mathrm{Pr}(B_j)$$

- Given an array A of size $n$, let $C_n$ be the number of comparisons needed for $A$
- $C_n$ is a random variable

# Analysis via Recurrence

## Theorem (Recall: total expectation theorem)

*Given a probability space $(\Omega, \mathcal{F}, \Pr)$, consider a partition $\{B_j\}_{j=1}^n$ of $\Omega$, then the expectation of a random variable $X$ can be represented as*

$$\mathbb{E}(X) = \sum_{j=1}^{n} \mathbb{E}(X|B_j)\Pr(B_j)$$

- Given an array A of size $n$, let $C_n$ be the number of comparisons needed for $A$
- $C_n$ is a random variable
- event $B_j$: choose the $j$-th smallest value of $A$ (i.e., rank $j$) as the pivot
- $\Pr(B_j) = 1/n$

$$M_n := \mathbb{E}(C_n) = \sum_{j=1}^{n} \mathbb{E}(C_n|B_j)\Pr(B_j)$$

# Solution

event $B_j$: the selected pivot is the $j$-th smallest value

- we take $n - 1$ comparisons for split

# Solution

event $B_j$: the selected pivot is the $j$-th smallest value

- we take $n-1$ comparisons for split
- the set $A^j_{\text{left}}$ of values smaller than it has size $j-1$
- the set $A^j_{\text{right}}$ of values greater has size $n-j$

# Solution

event $B_j$: the selected pivot is the $j$-th smallest value

- we take $n-1$ comparisons for split
- the set $A^j_{\text{left}}$ of values smaller than it has size $j-1$
- the set $A^j_{\text{right}}$ of values greater has size $n-j$

Given event $B_j$, the needed comparisons are $n-1+C_{j-1}+C_{n-j}$

# Solution

event $B_j$: the selected pivot is the $j$-th smallest value

- we take $n-1$ comparisons for split
- the set $A^j_{\text{left}}$ of values smaller than it has size $j-1$
- the set $A^j_{\text{right}}$ of values greater has size $n-j$

Given event $B_j$, the needed comparisons are $n-1+C_{j-1}+C_{n-j}$

$$M_n := \mathbb{E}(C_n) = \sum_{j=1}^{n} \mathbb{E}(C_n|B_j)\Pr(B_j)$$

$$= \sum_{j=1}^{n}(n-1+M_{j-1}+M_{n-j})\frac{1}{n}$$

$$= n-1+\frac{2}{n}\sum_{j=1}^{n-1} M_j \,.$$

# Results

**Theorem**

$M_n = \mathcal{O}(n \log n)$

**Proof.**

(Guess and) Verify by induction.[1] □

[1] https://www.cl.cam.ac.uk/teaching/1920/Probablty/materials/Lecture5.pdf for details.

# Results

**Theorem**

$M_n = \mathcal{O}(n \log n)$

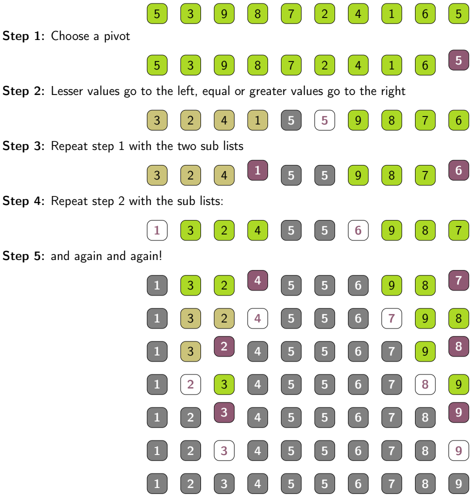
**Proof.**

(Guess and) Verify by induction.[1] □

In the next...
Slick analysis of QuickSort

[1] https://www.cl.cam.ac.uk/teaching/1920/Probablty/materials/Lecture5.pdf for details.

# Property of deterministic/randomized quick-sort




- the pivot is compared with **every** element in the array exactly once.
- the pivot will be excluded from the recursive calls

**property**

- a) If two elements are compared, then one of them is pivot.
- b) If two elements belong to $S_{\text{smaller}}$ and $S_{\text{larger}}$, they will be never compared.
- c) Any two fixed elements are compared at most **once**!
  - because of a) and b)

# Theoretical results

**Theorem**

*Given an array* $A = \{a_1, a_2, \cdots, a_n\}$ *with size* $n$*, denote* $Z$ *as the number of comparisons for randomized quick-sort, then* $\mathbb{E}[Z] \leq 2n \log n$.

# Theoretical results

**Theorem**

*Given an array* $A = \{a_1, a_2, \cdots, a_n\}$ *with size* $n$, *denote* $Z$ *as the number of comparisons for randomized quick-sort, then* $\mathbb{E}[Z] \le 2n \log n$.

- For $i, j \in [n]$ with $i \neq j$, event $R_{ij}$ denotes the element $a_i$ is compared with $a_j$
- $X_{ij}$ is an indicator random variable for $R_{ij}$

$$X_{ij} = \begin{cases} 1 & \text{if } a_i, a_j \text{ are compared} \\ 0 & \text{otherwise.} \end{cases}$$

then we have $Z = \sum_{i<j} X_{ij}$ [using property c)]. This is equivalent to:

# Theoretical results

**Theorem**

*Given an array $A = \{a_1, a_2, \cdots, a_n\}$ with size $n$, denote $Z$ as the number of comparisons for randomized quick-sort, then $\mathbb{E}[Z] \leq 2n \log n$.*

- For $i, j \in [n]$ with $i \neq j$, event $R_{ij}$ denotes the element $a_i$ is compared with $a_j$
- $X_{ij}$ is an indicator random variable for $R_{ij}$

$$X_{ij} = \begin{cases} 1 & \text{if } a_i, a_j \text{ are compared} \\ 0 & \text{otherwise.} \end{cases}$$

then we have $Z = \sum_{i<j} X_{ij}$ [using property c)]. This is equivalent to:
Let $A^* = \{a_1^*, a_2^*, \cdots, a_n^*\}$ be the correctly sorted list. Denote a random variable $Y_{ij}$ with $i, j \in [n]$ as

$$Y_{ij} = \begin{cases} 1 & \text{if } a_i^*, a_j^* \text{ are compared} \\ 0 & \text{otherwise.} \end{cases}$$

then we have $Z = \sum_{i<j} Y_{ij}$ [using property c)].

# Question: what is $\Pr(Y_{ij} = 1)$?

To determine when $a_i^*$ and $a_j^*$ are compared, we need to ensure

# Question: what is $\Pr(Y_{ij} = 1)$?

To determine when $a_i^*$ and $a_j^*$ are compared, we need to ensure

- either $a_i^*$ or $a_j^*$ to be chosen as a pivot [property a)]

# Question: what is $\Pr(Y_{ij} = 1)$?

To determine when $a_i^*$ and $a_j^*$ are compared, we need to ensure

- either $a_i^*$ or $a_j^*$ to be chosen as a pivot [property a)]
- We cannot choose any of $\{a_{i+1}^*, \cdots, a_{j-1}^*\}$ as pivot. Otherwise $a_i^*$ and $a_j^*$ are split into two different sets, and will never be compared. [property b)]

## Question: what is $\Pr(Y_{ij} = 1)$?

To determine when $a_i^*$ and $a_j^*$ are compared, we need to ensure

- either $a_i^*$ or $a_j^*$ to be chosen as a pivot [property a)]
- We cannot choose any of $\{a_{i+1}^*, \cdots, a_{j-1}^*\}$ as pivot. Otherwise $a_i^*$ and $a_j^*$ are split into two different sets, and will never be compared. [property b)]

That means, we are doing a **dart game** over $\{a_i^*, a_{i+1}^*, \cdots, a_{j-1}^*, a_j^*\}$ (if beyond this set, we throw another dart): we throw a dart at random into the array

# Question: what is $\Pr(Y_{ij} = 1)$?

To determine when $a_i^*$ and $a_j^*$ are compared, we need to ensure

- either $a_i^*$ or $a_j^*$ to be chosen as a pivot [property a)]
- We cannot choose any of $\{a_{i+1}^*, \cdots, a_{j-1}^*\}$ as pivot. Otherwise $a_i^*$ and $a_j^*$ are split into two different sets, and will never be compared. [property b)]

That means, we are doing a **dart game** over $\{a_i^*, a_{i+1}^*, \cdots, a_{j-1}^*, a_j^*\}$ (if beyond this set, we throw another dart): we throw a dart at random into the array

- if we hit $a_i^*$ or $a_j^*$, then $Y_{ij} = 1$
- if we hit $a_{i+1}^*, \cdots, a_{j-1}^*$, then $Y_{ij} = 0$

## Question: what is $\Pr(Y_{ij} = 1)$?

To determine when $a_i^*$ and $a_j^*$ are compared, we need to ensure

- either $a_i^*$ or $a_j^*$ to be chosen as a pivot [property a)]
- We cannot choose any of $\{a_{i+1}^*, \cdots, a_{j-1}^*\}$ as pivot. Otherwise $a_i^*$ and $a_j^*$ are split into two different sets, and will never be compared. [property b)]

That means, we are doing a **dart game** over $\{a_i^*, a_{i+1}^*, \cdots, a_{j-1}^*, a_j^*\}$ (if beyond this set, we throw another dart): we throw a dart at random into the array

- if we hit $a_i^*$ or $a_j^*$, then $Y_{ij} = 1$
- if we hit $a_{i+1}^*, \cdots, a_{j-1}^*$, then $Y_{ij} = 0$

Accordingly, we have

$$\Pr(Y_{ij} = 1) = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1},$$

and $\mathbb{E}(Y_{ij}) = \Pr(Y_{ij} = 1)$.

# Results

$$\mathbb{E}[Z] = \sum_{i<j} \mathbb{E}[Y_{ij}] = 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{1}{j-i+1} = 2\sum_{i=1}^{n-1} \frac{n-i}{i+1} = 2\sum_{k=2}^{n} \frac{n}{k} - 2\sum_{i=1}^{n-1} \frac{i}{i+1}$$

where we observe

$$\begin{aligned}
\text{if } i = 1, &\quad \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-2} + \frac{1}{n-1} + \frac{1}{n} \\
\text{if } i = 2, &\quad \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-2} + \frac{1}{n-1} \\
\text{if } i = 3, &\quad \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-2} \\
&\cdots \\
\text{if } i = n-1, &\quad \frac{1}{2}
\end{aligned}$$

where we use $\sum_{i=1}^{n-1} \frac{i}{i+1} \geq \frac{n-1}{2}$.

# Results

$$\mathbb{E}[Z] = \sum_{i<j} \mathbb{E}[Y_{ij}] = 2\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1} = 2\sum_{i=1}^{n-1} \frac{n-i}{i+1} = 2\sum_{k=2}^{n} \frac{n}{k} - 2\sum_{i=1}^{n-1} \frac{i}{i+1}$$

Recall the definition of harmonic numbers,

$$H_n = \sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n) = \log n + \gamma + \frac{1}{2n} + \mathcal{O}(\frac{1}{n^2}).$$

Then we have

$$\mathbb{E}[Z] \leq 2\left(\sum_{k=1}^{n} \frac{n}{k} - \frac{n-1}{2}\right) \leq 2n\log n,$$

where we use $\sum_{i=1}^{n-1} \frac{i}{i+1} \geq \frac{n-1}{2}$.

# Numerical validations[2]

- setting (left and middle): 1000 arrays with size 1000, run 50 times.
- setting (right): a fixed reverse-sorted input array with size 1000

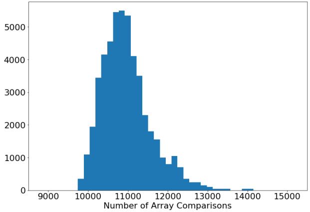



Figure: Distribution of run-time of deterministic Quick-sort over random array inputs.

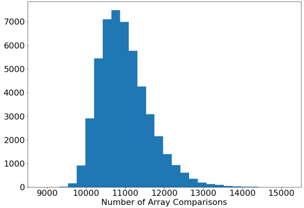



Figure: Distribution of run-time of randomized Quick-sort over random array inputs.

[2] figure credit: https://balaramdb.com/2021/08/analysis-of-randomized-quicksort/

# Numerical validations[2]

- setting (left and middle): 1000 arrays with size 1000, run 50 times.
- setting (right): a fixed reverse-sorted input array with size 1000

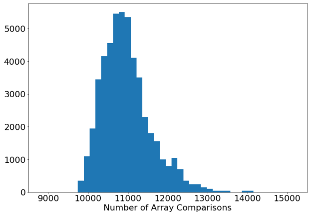



Figure: Distribution of run-time of deterministic Quick-sort over random array inputs.

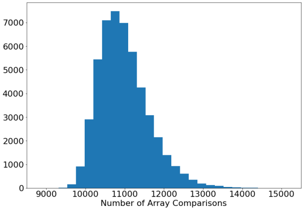



Figure: Distribution of run-time of randomized Quick-sort over random array inputs.

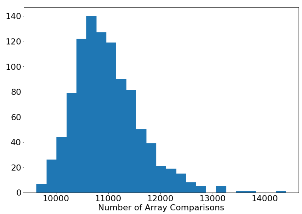



Figure: Distribution of run-time of randomized Quick-sort over a fixed reverse-sorted input array. Deterministic quick-sort takes 499,500 comparisons.

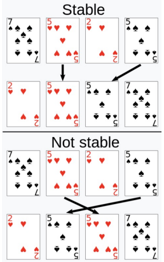
[2]figure credit: https://balaramdb.com/2021/08/analysis-of-randomized-quicksort/

# Comparison of sorting algorithms

check more details if you're interested in
https://en.wikipedia.org/wiki/Sorting_algorithm

| Algorithm | Best case | Average case | Worst case | Stable |
|---|---|---|---|---|
| Bubble-sort | $\mathcal{O}(n)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(n^2)$ | ✓ |
| Merge-sort | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n \log n)$ | ✓ |
| Quick-sort (deterministic) | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n^2)$ | ✗ |
| Quick-sort (randomized) | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n \log n)$ | $\mathcal{O}(n^2)$ | ✗ |

**Remark:** the worst-case expected-time complexity for randomized quick-sort is $\mathcal{O}(n \log n)$.



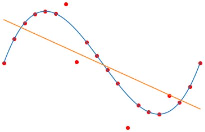

◦ stable: sort equal elements in the same order that they appear in the input

# Thanks for your attention!

# Q & A

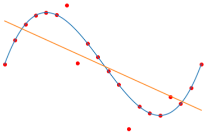
my homepage www.lfhsgre.org for more information!

# Curve fitting from under-fitting to benign overfitting



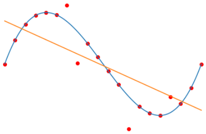
(a) $y_i = f_\rho(\boldsymbol{x}_i) + \epsilon$

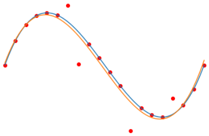
# Curve fitting from under-fitting to benign overfitting



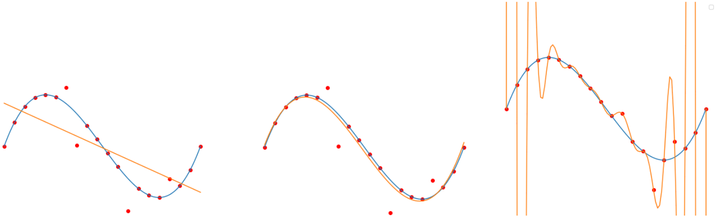
(a) under-fitting

# Curve fitting from under-fitting to benign overfitting


(a) under-fitting

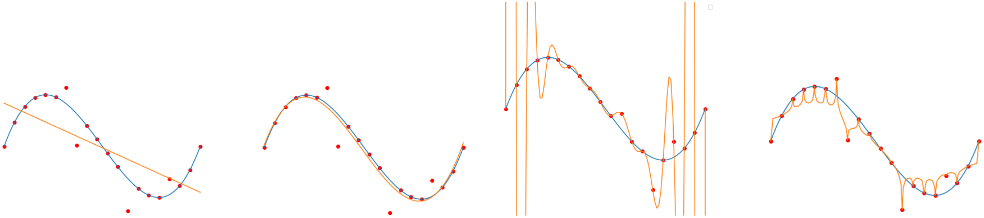

(b) sweet spot

# Curve fitting from under-fitting to benign overfitting



(a) under-fitting　　(b) sweet spot　　(c) overfitting

# Curve fitting from under-fitting to benign overfitting



(a) under-fitting (b) sweet spot (c) overfitting (d) benign overfitting

Figure: Test performance on curve fitting: source from Open AI.