CS4311
Design and Analysis of Algorithms

Lecture 5: Quicksort
About this lecture

- **Introduce Quicksort**
  - Cinderella’s New Problem

- **Running time of Quicksort**
  - Worst-Case
  - Average-Case
Do you remember Cinderella’s Problem?

You have to find the largest bolt and the largest nut

I see...
Cinderella’s New Problem

You have to sort the bolts and sort the nuts

I see...
Fairy Godmother’s Proposal

1. Pick one of the nuts.
2. Compare this nut with all other bolts → Find those which are larger, and find those which are smaller.
Fairy Godmother's Proposal

Bolts smaller than nut

Bolt equal to nut

Bolts larger than nut

picked nut

Done!
Fairy Godmother’s Proposal

3. Pick the bolt that is equal to the selected nut
4. Compare this bolt with all other nuts  ➞ Find those which are larger, and find those which are smaller
Fairy Godmother's Proposal

Done!

Nuts smaller than bolt

Nuts larger than bolt
Fairy Godmother's Proposal

5. Sort left part (recursively)
6. Sort right part (recursively)

^^ This is all of my proposal ^^
Fairy Godmother’s Proposal

• Can you see why Fairy Godmother’s proposal is a correct algorithm?

• What is the running time?
  • Worst-case: $\Theta(n^2)$ comparisons
  • No better than the brute force approach!!

• Though worst-case runs badly, the average case is good: $\Theta(n \log n)$ comparisons
Quicksort uses Partition

The previous algorithm is exactly Quicksort, which makes use of a Partition function:

Partition($A, p, r$)  /* to partition array $A[p..r]$ */
1. Pick an element, say $A[t]$ (called pivot)
2. Let $q$ = #elements less than pivot
3. Put elements less than pivot to $A[p..p+q-1]$
4. Put pivot to $A[p+q]$
5. Put remaining elements to $A[p+q+1..r]$
6. Return $q$
More on Partition

• After \texttt{Partition}(A,p,r), we obtain the value \texttt{q}, and know that
  • Pivot was \texttt{A}[p+q]
  • Before \texttt{A}[p+q]: smaller than pivot
  • After \texttt{A}[p+q]: larger than pivot

• There are many ways to perform \texttt{Partition}. One way is shown in the next slides
  • It will be an \texttt{in-place} algorithm (using \texttt{O}(1) extra space in addition to the input array)
Ideas for In-Place Partition

• Idea 1: Use $A[r]$ (the last element) as pivot
• Idea 2: Process $A[p..r]$ from left to right
  • The prefix (the beginning part) of $A$ stores all elements less than pivot seen so far
  • Use two counters:
    • One for the length of the prefix
    • One for the element we are looking
**In-Place Partition in Action**

**Before running**

Length of prefix = 0

Because next element is less than pivot, we shall extend the prefix by 1
In-Place Partition in Action

Length of prefix = 1

Because next element is smaller than pivot, and is adjacent to the prefix, we extend the prefix
In-Place Partition in Action

Length of prefix = 2

Because next element is larger than pivot, no change to prefix
In-Place Partition in Action

Length of prefix = 2

After 3 steps

Again, next element is larger than pivot, no change to prefix
In-Place Partition in Action

Length of prefix = 2

Because next element is less than pivot, we shall extend the prefix by swapping
In-Place Partition in Action

Length of prefix = 3

1 3 2 8 7 6 4 5

Because next element is larger than pivot, no change to prefix

after 5 steps
In-Place Partition in Action

Length of prefix = 3

Because next element is less than pivot, we shall extend the prefix by swapping
In-Place Partition in Action

Length of prefix = 4

1 3 2 4 7 6 8 5

When next element is the pivot, we put it after the end of the prefix by swapping
In-Place Partition in Action

Length of prefix = 4

Partition is done, and return length of prefix

after 8 steps
The Quicksort algorithm works as follows:

\[
\text{Quicksort}(A, p, r) \quad /* \text{to sort array } A[p..r] */ \\
1. \text{if } (p \geq r) \text{ return;}
2. q = \text{Partition}(A, p, r);
3. \text{Quicksort}(A, p, p+q-1);
4. \text{Quicksort}(A, p+q+1, r);
\]

To sort \(A[1..n]\), we just call \text{Quicksort}(A, 1, n)
Worst-Case Running Time

The worst-case running time of Quicksort can be expressed by:

\[ T(n) = \max_{q=0}^{n-1} \left( T(q) + T(n-q-1) \right) + \Theta(n) \]

We prove \( T(n) = O(n^2) \) by substitution method:
1. Guess \( T(n) \leq cn^2 \) for some constant \( c \)
2. Next, verify our guess by induction
Worst-Case Running Time

Inductive Case:

\[ T(n) = \max_{q=0 \text{ to } n-1} \left( T(q) + T(n-q-1) \right) + \Theta(n) \]

\[ \leq \max_{q=0 \text{ to } n-1} \left( cq^2 + c(n-q-1)^2 \right) + \Theta(n) \]

\[ \leq c(n-1)^2 + \Theta(n) \]

\[ = cn^2 - 2cn + c + \Theta(n) \]

\[ \leq cn^2 \quad \text{when } c \text{ is large enough} \]

Maximized when \( q = 0 \) or when \( q = n-1 \)

Inductive Case is OK now. How about Base Case?
Worst-Case Running Time

Conclusion:

1. $T(n) = O(n^2)$

2. However, we can also show $T(n) = \Omega(n^2)$ by finding a worst-case input

$\Rightarrow T(n) = \Theta(n^2)$
Average-Case Running Time

So, Quicksort runs badly for some input...

But suppose that when we store a set of $n$ numbers into the input array, each of the $n!$ permutations are equally likely

$\Rightarrow$ Running time varies on input

What will be the “average” running time?
Average Running Time

Let $X = \# \text{ comparisons in all Partition}

Later, we will show that

Running time = $O(n + X)$, varies on input

Finding average of $X$ (i.e. #comparisons) gives average running time

Our first target: Compute average of $X$
Average # of Comparisons

We define some notation to help the analysis:

• Let $a_1, a_2, ..., a_n$ denote the set of $n$ numbers initially placed in the array

• Further, we assume $a_1 < a_2 < ... < a_n$ (So, $a_1$ may not be the element in $A[1]$ originally)

• Let $X_{ij} = \#$ comparisons between $a_i$ and $a_j$ in all Partition calls
Average # of Comparisons

Then, \( X = \# \) comparisons in all Partition calls
\[
= X_{12} + X_{13} + \ldots + X_{n-1,n}
\]

\[ \Rightarrow \text{Average # comparisons} \]
\[
= \mathbb{E}[X]
\]
\[
= \mathbb{E}[X_{12} + X_{13} + \ldots + X_{n-1,n}]
\]
\[
= \mathbb{E}[X_{12}] + \mathbb{E}[X_{13}] + \ldots + \mathbb{E}[X_{n-1,n}]
\]
Average # of Comparisons

The next slides will prove:  \( E[X_{ij}] = \frac{2}{(j-i+1)} \)

Using this result,

\[
E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{(j-i+1)}
\]

\[
= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{(k+1)}
\]

\[
< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}
\]

\[
= \sum_{i=1}^{n-1} \mathcal{O}(\log n) = \mathcal{O}(n \log n)
\]
Comparison between $a_i$ and $a_j$

Question: # times $a_i$ be compared with $a_j$?

Answer: At most once, which happens only if $a_i$ or $a_j$ are chosen as pivot.

After that, the pivot is fixed and is never compared with the others.
Comparison between $a_i$ and $a_j$

Question: Will $a_i$ always be compared with $a_j$?

Answer: No. E.g., after Partition in Page 14:

1 3 2 4 5 6 8 7

pivot

we will separately Quicksort the first 4 elements, and then the last 3 elements

$\Rightarrow$ 3 is never compared with 8
Comparison between $a_i$ and $a_j$

Observation:

Consider the elements $a_i, a_{i+1}, \ldots, a_{j-1}, a_j$

(i) If $a_i$ or $a_j$ is first chosen as a pivot, then $a_i$ is compared with $a_j$

(ii) Else, if any element of $a_{i+1}, \ldots, a_{j-1}$ is first chosen as a pivot, then $a_i$ is never compared with $a_j$
Comparison between $a_i$ and $a_j$

When the $n!$ permutations are equally likely to be the input,

\[
\Pr(a_i \text{ compared with } a_j \text{ once}) = \frac{2}{j-i+1} \\
\Pr(a_i \text{ not compared with } a_j) = \frac{j-i-1}{j-i+1}
\]

\[\Rightarrow \mathbb{E}[X_{ij}] = 1 \times \frac{2}{j-i+1} + 0 \times \frac{j-i-1}{j-i+1} = \frac{2}{j-i+1}\]

Consider $a_i, a_{i+1}, ..., a_{j-1}, a_j$. Given a permutation, if $a_i$ is chosen a pivot first, then by exchanging $a_i$ with $a_{i+1}$ initially, $a_{i+1}$ will be chosen as a pivot first.
Proof: Running time = \( O(n+X) \)

Observe that in the Quicksort algorithm:

- Each Partition fixes the position of pivot
  - exactly \( n \) Partition calls
- After each Partition, we have 2 Quicksort
- Also, all Quicksort (except 1st one: Quicksort(A,1,n)) are invoked after a Partition
  - total \( \Theta(n) \) Quicksort calls
Proof: Running time = \(O(n+X)\)

So, if we ignore the comparison time in all Partition calls, the time used = \(O(n)\)

Thus, we include back the comparison time in all Partition calls,

Running time = \(O(n + X)\)