Randomized Algorithms, Quicksort and Randomized Selection

Carola Wenk

Slides courtesy of Charles Leiserson with additions by Carola Wenk
Deterministic Algorithms

Runtime for deterministic algorithms with input size $n$:

- **Best-case runtime**
  - Attained by one input of size $n$

- **Worst-case runtime**
  - Attained by one input of size $n$

- **Average runtime**
  - Averaged *over all possible inputs* of size $n$
Deterministic Algorithms: Insertion Sort

for j=2 to n {
    key = A[j]
    // insert A[j] into sorted sequence A[1..j-1]
    i=j-1
    while(i>0 && A[i]>key) {
        A[i+1]=A[i]
        i--
    }
    A[i+1]=key
}
Deterministic Algorithms: Insertion Sort

Best-case runtime: $O(n)$, input $[1,2,3,\ldots,n]$

- Attained by one input of size $n$

- Worst-case runtime: $O(n^2)$, input $[n, n-1, \ldots, 2, 1]$

- Attained by one input of size $n$

- Average runtime: $O(n^2)$

- Averaged over all possible inputs of size $n$

- What kind of inputs are there?
- How many inputs are there?
Average Runtime

• What kind of inputs are there?
  • Do \([1,2,\ldots,n]\) and \([5,6,\ldots,n+5]\) cause different behavior of Insertion Sort?
  • No. Therefore it suffices to only consider all permutations of \([1,2,\ldots,n]\).

• How many inputs are there?
  • There are \(n!\) different permutations of \([1,2,\ldots,n]\).
## Average Runtime

### Insertion Sort: \( n=4 \)

- **Inputs:** \( 4! = 24 \)

<table>
<thead>
<tr>
<th>Input</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,2,3,4]</td>
<td>[4,1,2,3]</td>
<td>[4,1,3,2]</td>
<td>[4,3,2,1]</td>
<td>[1,2,3,4]</td>
<td>[1,4,2,3]</td>
<td>[1,4,3,2]</td>
<td>[3,4,2,1]</td>
<td>[1,3,2,4]</td>
<td>[1,2,4,3]</td>
<td>[1,3,4,2]</td>
<td>[3,2,4,1]</td>
</tr>
<tr>
<td>[2,1,3,4]</td>
<td>[4,1,2,3]</td>
<td>[4,1,3,2]</td>
<td>[4,3,2,1]</td>
<td>[1,2,3,4]</td>
<td>[1,4,2,3]</td>
<td>[1,4,3,2]</td>
<td>[3,4,2,1]</td>
<td>[1,3,2,4]</td>
<td>[1,2,4,3]</td>
<td>[1,3,4,2]</td>
<td>[3,2,4,1]</td>
</tr>
<tr>
<td>[1,3,2,4]</td>
<td>[1,2,4,3]</td>
<td>[1,3,4,2]</td>
<td>[3,2,4,1]</td>
<td>[1,2,3,4]</td>
<td>[1,4,2,3]</td>
<td>[1,4,3,2]</td>
<td>[3,4,2,1]</td>
<td>[1,3,2,4]</td>
<td>[1,2,4,3]</td>
<td>[1,3,4,2]</td>
<td>[3,2,4,1]</td>
</tr>
<tr>
<td>[3,1,2,4]</td>
<td>[4,2,1,3]</td>
<td>[4,3,1,2]</td>
<td>[4,2,3,1]</td>
<td>[2,1,3,4]</td>
<td>[2,1,4,3]</td>
<td>[2,3,4,1]</td>
<td>[2,4,3,1]</td>
<td>[3,1,2,4]</td>
<td>[2,1,4,3]</td>
<td>[3,1,4,2]</td>
<td>[2,3,4,1]</td>
</tr>
<tr>
<td>[3,2,1,4]</td>
<td>[2,1,4,3]</td>
<td>[3,1,4,2]</td>
<td>[2,3,4,1]</td>
<td>[2,1,3,4]</td>
<td>[2,1,4,3]</td>
<td>[2,3,4,1]</td>
<td>[2,4,3,1]</td>
<td>[3,1,2,4]</td>
<td>[2,1,4,3]</td>
<td>[3,1,4,2]</td>
<td>[2,3,4,1]</td>
</tr>
</tbody>
</table>

- **Runtime is proportional to:** \( 3 + \#\text{times in while loop} \)

- **Best:** \( 3+0 \), **Worst:** \( 3+6=9 \), **Average:** \( 3+72/24 = 6 \)
Average Runtime: Insertion Sort

• The average runtime averages runtimes over all $n!$ different input permutations

• Disadvantage of considering average runtime:
  • There are still worst-case inputs that will have the worst-case runtime
  • Are all inputs really equally likely? That depends on the application

⇒ Better: Use a randomized algorithm
Randomized Algorithm: Insertion Sort

• Randomize the order of the input array:
  • Either prior to calling insertion sort,
  • or during insertion sort (insert random element)
  • This makes the runtime depend on a probabilistic experiment (sequence of numbers obtained from random number generator; or random input permutation)

  \[\Rightarrow \text{Runtime is a random variable (maps sequence of random numbers to runtimes)}\]

• Expected runtime = expected value of runtime random variable
Randomized Algorithm: Insertion Sort

- Runtime is independent of input order ([1,2,3,4] may have good or bad runtime, depending on sequence of random numbers)
- No assumptions need to be made about input distribution
- No one specific input elicits worst-case behavior
- The worst case is determined only by the output of a random-number generator.

⇒ When possible use expected runtimes of randomized algorithms instead of average case analysis of deterministic algorithms
Quicksort

- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).
- We are going to perform an expected runtime analysis on randomized quicksort
Quicksort: 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.

   \[
   \begin{array}{c|c|c}
   \leq x & x & \geq x \\
   \end{array}
   \]

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

3. **Combine:** Trivial.

**Key:** *Linear-time partitioning subroutine.*
Partitioning subroutine

\textsc{Partition}(A, p, q) \triangleright A[p \ldots q]

\begin{align*}
x & \leftarrow A[p] \quad \triangleright \text{pivot} = A[p] \\
i & \leftarrow p \\
\text{for } j & \leftarrow p + 1 \text{ to } q \\
\text{do if } A[j] & \leq x \\
& \text{then } i \leftarrow i + 1 \\
& \text{exchange } A[i] \leftrightarrow A[j] \\
\text{exchange } A[p] & \leftrightarrow A[i] \\
\text{return } i
\end{align*}

\textbf{Invariant:}

\begin{tabular}{c|c|c|c|c}
\hline
x & \leq x & \geq x & ? \\
\hline
p & i & j & q \\
\hline
\end{tabular}

Running time

=\(O(n)\) for \(n\) elements.
Example of partitioning

\[
\begin{array}{cccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
\end{array}
\]

\[i \quad j\]
Example of partitioning

\[ \begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
\end{array} \]

\[ i \rightarrow j \]
Example of partitioning

\begin{center}
\begin{tabular}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
\end{tabular}
\end{center}

\begin{center}
\begin{tikzpicture}
\node (i) at (0,0) {$i$};
\node (j) at (2,0) {$j$};
\draw (i) -- (j);
\end{tikzpicture}
\end{center}
Example of partitioning
Example of partitioning

\[ \begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
\end{array} \]

\( i \quad \rightarrow \quad j \)
Example of partitioning

\[ \begin{array}{cccccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
\end{array} \]
Example of partitioning

\begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & \color{blue}{5} & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & \color{red}{3} & 10 & 8 & 13 & 2 & 11 \\
\end{array}

\quad i \quad j
Example of partitioning

\[
\begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
\end{array}
\]
Example of partitioning

\[ \begin{array}{cccccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
6 & 5 & 3 & 2 & 8 & 13 & 10 & 11 \\
\end{array} \]

\[ \rightarrow i \quad j \]
Example of partitioning

\[\begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
6 & 5 & 13 & 10 & 8 & 3 & 2 & 11 \\
6 & 5 & 3 & 10 & 8 & 13 & 2 & 11 \\
6 & 5 & 3 & 2 & 8 & 13 & 10 & 11 \\
\end{array}\]
Example of partitioning

i \quad \rightarrow \quad j
## Example of partitioning

<table>
<thead>
<tr>
<th>6</th>
<th>10</th>
<th>13</th>
<th>5</th>
<th>8</th>
<th>3</th>
<th>2</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>5</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>10</td>
<td>8</td>
<td>13</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>13</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>13</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

$i$
Pseudocode for quicksort

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

Initial call: QUICKSORT(A, 1, n)
Analysis 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.
- Partition around min or max element.
- One side of partition always has no elements.

\[ T(n) = T(0) + T(n - 1) + \Theta(n) \]
\[ = \Theta(1) + T(n - 1) + \Theta(n) \]
\[ = T(n - 1) + \Theta(n) \]
\[ = \Theta(n^2) \quad \text{(arithmetic series)} \]
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

\[ T(n) = T(0) + T(n-1) + cn \]

\[ \Theta \left( \sum_{k=1}^{\text{height}} k \right) \]

height = \( n \)
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]

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

height = \( n \)
Worst-case recursion tree

\[ T(n) = T(0) + T(n-1) + cn \]

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

\[ height = n \]

\[ T(n) = \Theta(n) + \Theta(n^2) = \Theta(n^2) \]
Best-case analysis
(For intuition only!)

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

\[
T(n) = 2T(n/2) + \Theta(n)
= \Theta(n \log n) \quad \text{(same as merge sort)}
\]

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?
Analysis of “almost-best” case

$T(n)$
Analysis of “almost-best” case

\[ T\left(\frac{1}{10}n\right) \quad cn \quad T\left(\frac{9}{10}n\right) \]
Analysis of “almost-best” case

\[ cn \]

\[ \frac{1}{10} \]

\[ T\left(\frac{1}{100}n\right) T\left(\frac{9}{100}n\right) \]

\[ \frac{9}{10} \]

\[ T\left(\frac{9}{100}n\right) T\left(\frac{81}{100}n\right) \]
Analysis of “almost-best” case

\[ \Theta(1) \quad \Theta(1) \quad \Theta(1) \quad \Theta(1) \quad \Theta(1) \]

\[ O(n) \text{ leaves} \]

\[ \log_{10/9} n \]

\[ \frac{81}{100} \quad \frac{9}{100} \quad \frac{9}{100} \quad \frac{1}{10} \quad \frac{1}{100} \]

\[ cn \quad cn \quad cn \quad cn \quad cn \]

\[ O(n) \]

\[ cn \quad cn \quad cn \quad cn \quad cn \]

\[ \frac{100}{10} \quad \frac{99}{10} \quad \frac{98}{10} \quad \frac{97}{10} \quad \frac{96}{10} \]

\[ \frac{100}{9} \quad \frac{99}{9} \quad \frac{98}{9} \quad \frac{97}{9} \quad \frac{96}{9} \]

\[ \frac{100}{81} \quad \frac{99}{81} \quad \frac{98}{81} \quad \frac{97}{81} \quad \frac{96}{81} \]

\[ \frac{100}{9^2} \quad \frac{99}{9^2} \quad \frac{98}{9^2} \quad \frac{97}{9^2} \quad \frac{96}{9^2} \]

\[ \frac{100}{9^3} \quad \frac{99}{9^3} \quad \frac{98}{9^3} \quad \frac{97}{9^3} \quad \frac{96}{9^3} \]

\[ \frac{100}{9^4} \quad \frac{99}{9^4} \quad \frac{98}{9^4} \quad \frac{97}{9^4} \quad \frac{96}{9^4} \]

\[ \frac{100}{9^5} \quad \frac{99}{9^5} \quad \frac{98}{9^5} \quad \frac{97}{9^5} \quad \frac{96}{9^5} \]

\[ \cdots \]
Analysis of “almost-best” case

\[ cn \leq T(n) \leq cn \log_{10/9} n + O(n) \]

\( \Theta(1) \)

\( \Theta(n \log n) \)

\( O(n) \) leaves
Quicksort Runtimes

- Best case runtime $T_{\text{best}}(n) \in O(n \log n)$
- Worst case runtime $T_{\text{worst}}(n) \in O(n^2)$

- Worse than mergesort? Why is it called quicksort then?
- Its average runtime $T_{\text{avg}}(n) \in O(n \log n)$
- Better even, the expected runtime of randomized quicksort is $O(n \log n)$
Average Runtime

The **average runtime** $T_{\text{avg}}(n)$ for Quicksort is the average runtime over all possible inputs of length $n$.

- $T_{\text{avg}}(n)$ has to average the runtimes over all $n!$ different input permutations.
- There are still worst-case inputs that will have a $O(n^2)$ runtime

$\Rightarrow$ **Better:** Use randomized quicksort
Randomized quicksort

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

- Running time is independent of the input order. It depends only on the sequence \( s \) of random numbers.
- 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 sequence \( s \) of random numbers.
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.
Average Runtime vs. Expected Runtime

• Average runtime is averaged over all inputs of a deterministic algorithm.

• Expected runtime is the expected value of the runtime random variable of a randomized algorithm. It effectively “averages” over all sequences of random numbers.

• De facto both analyses are very similar. However in practice the randomized algorithm ensures that not one single input elicits worst case behavior.
Order statistics

Select the $i$th smallest of $n$ elements (the element with rank $i$).

- $i = 1$: **minimum**;
- $i = n$: **maximum**;
- $i = \lceil (n+1)/2 \rceil$ or $\lfloor (n+1)/2 \rfloor$: **median**.

**Naive algorithm**: Sort and index $i$th element.
Worst-case running time $= \Theta(n \log n + 1) = \Theta(n \log n)$, using merge sort (not quicksort).
Randomized divide-and-conquer algorithm

\[\text{RAND-SELECT}(A, p, q, i) \quad \triangleright \text{i-th smallest of } A[p \ldots q]\]

if \( p = q \) then return \( A[p] \)

\( r \leftarrow \text{RAND-PARTITION}(A, p, q) \)

\( k \leftarrow r - p + 1 \quad \triangleright k = \text{rank}(A[r])\)

if \( i = k \) then return \( A[r] \)

if \( i < k \) then return \( \text{RAND-SELECT}(A, p, r - 1, i) \)

else return \( \text{RAND-SELECT}(A, r + 1, q, i - k) \)
Example

Select the $i = 7$th smallest:

\[
\begin{array}{cccccccc}
6 & 10 & 13 & 5 & 8 & 3 & 2 & 11 \\
\end{array}
\]

\[i = 7\]

**pivot**

Partition:

\[
\begin{array}{cccccccc}
2 & 5 & 3 & 6 & 8 & 13 & 10 & 11 \\
\end{array}
\]

\[k = 4\]

Select the $7 - 4 = 3$rd smallest recursively.
Intuition for analysis

(All our analyses today assume that all elements are distinct.)

Lucky:

\[ T(n) = T(3n/4) + dn \]
\[ = \Theta(n) \]

Unlucky:

\[ T(n) = T(n - 1) + dn \]
\[ = \Theta(n^2) \]

Worse than sorting!

\[ n^{\log_{4/3} 1} = n^0 = 1 \]

Case 3

arithmetic series
Analysis of expected time

• Call a pivot **good** if its rank lies in \([n/4, 3n/4]\).

• How many good pivots are there? \(n/2\)

  \(\Rightarrow\) A random pivot has 50% chance of being good.

• Let \(T(n,s)\) be the runtime random variable

\[
T(n,s) \leq T(3n/4,s) + X(s) \cdot dn
\]

- Time to reduce array size to \(\leq 3/4n\)
- #times it takes to find a good pivot
- Runtime of partition
Analysis of expected time

**Lemma:** A fair coin needs to be tossed an expected number of 2 times until the first “heads” is seen.

**Proof:** Let $E(X)$ be the expected number of tosses until the first “heads” is seen.

- Need at least one toss, if it’s “heads” we are done.
- If it’s “tails” we need to repeat (probability $\frac{1}{2}$).

\[ E(X) = 1 + \frac{1}{2} E(X) \]
\[ \Rightarrow E(X) = 2 \]
Analysis of expected time

\[ T(n,s) \leq T(3n/4,s) + X(s) \cdot dn \]

- Time to reduce array size to \( \leq 3/4n \)
- #times it takes to find a good pivot
- Runtime of partition

\[ \Rightarrow E(T(n,s)) \leq E(T(3n/4,s)) + E(X(s) \cdot dn) \]
\[ \Rightarrow E(T(n,s)) \leq E(T(3n/4,s)) + E(X(s)) \cdot dn \]
\[ \Rightarrow E(T(n,s)) \leq E(T(3n/4,s)) + 2 \cdot dn \]
\[ \Rightarrow T_{\text{exp}}(n) \leq T_{\text{exp}}(3n/4) + \Theta(n) \]
\[ \Rightarrow T_{\text{exp}}(n) \in \Theta(n) \]

Linearity of expectation

Lemma

CMPS 2200 Intro. to Algorithms 53
Summary of randomized order-statistic selection

- Works fast: linear expected time.
- Excellent algorithm in practice.
- But, the worst case is very bad: $\Theta(n^2)$.

**Q.** Is there an algorithm that runs in linear time in the worst case?

**A.** Yes, due to Blum, Floyd, Pratt, Rivest, and Tarjan [1973].

**Idea:** Generate a good pivot recursively. This algorithms large constants though and therefore is not efficient in practice.