CSc445 Algorithms

Quick Sort and median selection

Alon Efrat
Based on slides curacy of
Piotr Indyk and Carola Wenk

QuickSort –
example of the
divide-and-concourse paradigm

• Sorts “in place” (no need for extra space).
  Like insertion sort, but not like merge sort.
• Very practical (with tuning).

Divide and conquer

Quick sort an n-element array:
1. Divide: Partition the array into two subarrays
   around a pivot x such that elements in lower
   subarray ≤ x ≤ elements in upper subarray.

   ≤ x ≤

2. Conquer: Recursively sort the two subarrays.

   Key: Linear-time partitioning subroutine.
Partitioning subroutine

\[
\text{PARTITION}(A, p, q) \rightarrow A[p \ldots q]
\]

\[
x \leftarrow A[p] \quad \text{pivot} = A[p]
\]

\[
i \leftarrow p
\]

\[
\text{for } j \leftarrow p + 1 \text{ to } q
\]

\[
do \text{ if } A[j] \leq x
\]

\[
\text{then}
\]

\[
i \leftarrow i + 1
\]

\[
\text{exchange } A[i] \leftrightarrow A[j]
\]

\[
\text{Now } A[i] > x
\]

\[
\text{exchange } A[p] \leftrightarrow A[i]
\]

\[
\text{return } i
\]

\[
\text{Invariant: } x \leq x > x ?
\]

\[
\text{Running time } = O(n)
\]

for \( n \) elements.

Example of partitioning

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

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

Example of partitioning

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

Example of partitioning

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

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
<td>13</td>
<td>5</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

\[i \quad j\]
Example of partitioning

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

Example of partitioning

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

Example of partitioning

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
</tbody>
</table>
Example of partitioning

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

Pseudocode for quicksort

\[
\text{QUICKSORT}(A, p, r) \\
\text{if } p < r \\
\quad \text{then } q \leftarrow \text{PARTITION}(A, p, r) \\
\quad \text{QUICKSORT}(A, p, q-1) \\
\quad \text{QUICKSORT}(A, q+1, r) \\
\]

\textit{Initial call:} \text{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

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

\[ \Theta(1) \]
Worst-case recursion tree

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

\[ T(0) \quad c(n-1) \quad \Theta \left( \sum_{k=1}^{n} k \right) = \Theta(n^2) \]

\[ T(0) \quad c(n-2) \quad \Theta(1) \quad \cdots \quad \Theta(1) \]

Best-case and almost best-case analysis

If we are lucky, \textsc{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 \( \frac{1}{10} : \frac{9}{10} \)?

\[ T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \Theta(n) \]

What is the solution to this recurrence?
Analysis of “almost-best” case

\[ T(n) \]

Analysis of “almost-best” case

\[ T\left(\frac{1}{m}n\right) \quad T\left(\frac{2}{m}n\right) \]

Analysis of “almost-best” case

\[ T\left(\frac{1}{100}n\right) T\left(\frac{2}{100}n\right) \quad T\left(\frac{3}{100}n\right) \]
Analysis of “almost-best” case

\[ T(n) = cn + 2T(n/2) \leq cn \log_{10/9} n + O(n) \leq 8c \log_2 n \]

Randomized quicksort

How can find a pivot that guarantees partitions with good ratios for \( A[1..n] \)?

We say that \( q \) is a good pivot if:

- at least 10% of the elements of \( A[1..n] \) are smaller than \( q \), and
- at least 10% of the elements of \( A[1..n] \) are larger than \( q \).

**Best pivot:** Pick the median of \( A[1..n] \) as pivot.

(median – an element that is larger than half of the elements)

Then the time would obey \( T(n) = cn + 2T(n/2) \)

**Problem:** need to work too hard to find the median (best pivot), so we will do with (only) a good pivot.
Finding a good pivot for \( A[1..n] \)

**5-random-elements method:**
- Pick the indices of 5 elements at random from \( A[1..n] \).
- For \( k=1 \) to 5
  \[ X[k] = A[n \text{ rand}()] \]
- Set \( q \) to be the median of \( X[1..5] \)

---

Finding a good pivot for \( A[1..n] \)

**5-random-elements method:**
- Pick 5 elements at random from \( A[1..n] \), and set \( q \) to be their median.
- What is the probability that \( q \) is not a good pivot?
- Let \( S \) be the elements of \( A[1..n] \) which are the 10% smallest.
- The probability that an elements picked at random is in \( S \) is 0.1.
- \( q \) is in \( S \) only if at least 3 of the 5 elements that we pick are in \( S \).
- The probability that this happens is
  \[ 0.1^3 + 5 \times 0.1^4 \times 0.9 + \]
  \[ \text{all in } S \quad 4 \text{ in } S, \text{one not in } S \quad 2 \text{ not in } S \]
  \[ = 0.00001 + 0.00045 + 0.00810 = 0.00856 \]
- This is also the probability that \( q \) is in the 10% largest elements.
- In other words: with probability \( \geq 98\% \), \( q \) is a good pivot.

---

Randomized quicksort – cont

**Finding good pivots**

Putting it together, during QS, each time that we need to find a pivot, we use the "5 random elements" method.

With probability 98\%, we find a good pivot.

The overall time that we spend on good partitions is much smaller than the time we spent on bad partitions.

(note – bad partitions are not harmful – they are just not helpful)

So the recursion formula \( T(n) = cn + T(n/10) + T(9n/10) \) still apply, leading to running time \( \Theta(n \log n) \).

This is expected running time – there is a chance that the actual running time is \( \Theta(n^2) \), but the probability that it happens is very slim.
Quicksort in practice

• Quicksort is a great general-purpose sorting algorithm.
• Quicksort is typically over twice as fast as merge sort.
• Quicksort behaves well even with caching and virtual memory.

Median Selection

• (CLRS Section 9.2, page 185).
• For \( A[1..n] \) (all different elements) we say that the rank of \( x \) is \( i \) if exactly \( i-1 \) elements in \( A \) are smaller than \( x \).
• In particular, the median is the \( \lceil \frac{n}{2} \rceil \)-smallest.
• To find the median, we could sort and pick \( A[\lceil \frac{n}{2} \rceil] \) (taken \( O(n \log n) \)).
• We can do better.

Median Selection-cont

```c
RS(A, p, r, i){
    //Randomize Selection: Returns i’st smallest element in A[p..r].
    //Assumption: Input is valid and elements are different.
    • If p==r return A[p]
    • q=PARTITION(A,p,r);
      *//Partition using the 5-random element method
    • k=q-p
    • If i==k+1 return A[q]
    • If i<k return RS(A, p, q-1, i) // Note the difference from QS
    • Else return RS(A, q+1, r, i-k-1)
}
```

13
**Time analysis**

- Recall: With high probability, we pick a good pivot:
  - Not in the 10% smallest or largest:
- Hence, we get rid of at least 10% of the elements of $A$
- So, $T(n) = cn + T(0.9\, n)$.
  - $T(n) = cn + 0.9n + 0.9^2n + 0.9^3n + \ldots$
  - $cn(1 + 0.9 + 0.9^2 + 0.9^3 + \ldots) = cn(1/(1-0.9)) = O(n)$.
- So the expected time is linear. (yuppie)

As in the case of QS, partitions which are not good are not harmful, just not helpful.