Computer Science 331

Quicksort

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #27

Outline

1. Introduction
2. Partitioning
   - Deterministic Partitioning
   - Randomized Partitioning
3. Quicksort
   - Deterministic Quicksort
   - Randomized Quicksort
4. Analysis
5. References

Introduction

Quicksort:

- A recursive "Divide and Conquer" sorting algorithm
- A simple deterministic version uses
  - $\Theta(n^2)$ operations to sort an array of size n in the worst case
  - $\Theta(n \log n)$ operations on average, assuming all relative orderings of the (distinct) input are equally likely
- The expected number of operations used by a *randomized* version is in $O(n \log n)$ for any input array of size n

Idea

1. Choose an element x and reorder the array as follows:
   - x is in the correct spot if the array was sorted
   - elements $< x$ are to the left of x in the array
   - elements $> x$ are to the right of x in the array
2. Recursively sort subarray of elements to the left of x
3. Recursively sort subarray of elements to the right of x

Step 1 is the key to this method being efficient. Issues:

- speed (can be done in time $\Theta(n)$)
- position of x — want the final position of x to be the *middle*, so the recursive calls are on arrays of size close to half as long as the original

Partitioning

This is the process that will be used to carry out step 1.

Precondition:

- p and r are integers such that $0 \leq p \leq r < A.length$

Postcondition:

- Value returned is an integer q such that $p \leq q \leq r$
- $A[h] \leq A[q]$ for every integer h such that $p \leq h \leq q-1$
- $A[h] > A[q]$ for every integer h such that $q+1 \leq h \leq r$
- If h is an integer such that $0 \leq h < p$ or such that $r+1 \leq h < A.length$ then A[h] has not been changed
- The entries of A have been reordered but are otherwise unchanged

Deterministic Partitioning

Idea:

- Pivot element used is the last element in the part of the array being processed. Other versions of this algorithm use the first element instead.
- Sweep from left to right over the array, exchanging elements as needed, so that values less than or equal to the pivot element are all located before values that are greater than the pivot element, in the part of the array that has been processed

Pseudocode

```
int DPartition(int [] A, int p, int r)
  x = A[r]
  i = p-1
  j = p
  while j < r do
    if A[j] ≤ x then
      i = i + 1
      Swap: tmp = A[i]; A[i] = A[j]; A[j] = tmp
    end if
    j = j + 1
  end while
  Swap: tmp = A[i+1]; A[i+1] = A[r]; A[r] = tmp
  return i + 1
```


Example

Consider the execution of **DPartition**(A, 3, 10) for A as follows:

	3	4	5	6	7	8	9	10	
···	2	6	4	1	7	3	0	5	···

Using $x = A[10] = 5$ as the pivot. Initially $i = 2$, $j = 3$.

$j = 3$: $A[j] = 2 < 5$; increment i and swap $A[3]$ & $A[3]$

$i = 3, j = 3$

	3	4	5	6	7	8	9	10	
···	2	6	4	1	7	3	0	5	···

$j = 4$: no change

$i = 3, j = 4$

	3	4	5	6	7	8	9	10	
···	2	6	4	1	7	3	0	5	···

Example (cont.)

$j = 5$: $A[j] = 4 < 5$; increment i and swap $A[4]$ & $A[5]$

$i = 4, j = 5$

	3	4	5	6	7	8	9	10	
$\cdots$	2	4	6	1	7	3	0	5	$\cdots$

$j = 6$: $A[j] = 1 < 5$; increment i and swap $A[5]$ & $A[6]$

$i = 5, j = 6$

	3	4	5	6	7	8	9	10	
$\cdots$	2	4	1	6	7	3	0	5	$\cdots$

$j = 7$: no change

$i = 5, j = 7$

	3	4	5	6	7	8	9	10	
$\cdots$	2	4	1	6	7	3	0	5	$\cdots$

Example (cont.)

$j = 8$: $A[j] = 3 < 5$; increment i and swap $A[6]$ & $A[8]$

$i = 6, j = 8$

	3	4	5	6	7	8	9	10	
$\cdots$	2	4	1	3	7	6	0	5	$\cdots$

$j = 9$: $A[j] = 0 < 5$; increment i and swap $A[7]$ & $A[9]$

$i = 7, j = 9$

	3	4	5	6	7	8	9	10	
$\cdots$	2	4	1	3	0	6	7	5	$\cdots$

$j = 10$: swap $A[i + 1] = A[8]$ & $A[10]$

$i = 7, j = 10$

	3	4	5	6	7	8	9	10	
$\cdots$	2	4	1	3	0	5	7	6	$\cdots$

Loop Invariant

Suppose p and r are integers such that $0 \leq p \leq r < \text{length}(A)$.

The the following properties are satisfied at the beginning of each execution of the body of the while-loop:

1. $\mathtt{j}$ is an integer such that $\mathtt{p} \leq \mathtt{j} < \mathtt{r}$.
2. $\mathtt{i}$ is an integer such that $\mathtt{p} - 1 \leq \mathtt{i} \leq \mathtt{j} - 1$.
3. The following hold for each integer ℓ such that $\mathtt{p} \leq \ell \leq \mathtt{r}$:
   - if $\mathtt{p} \leq \ell \leq \mathtt{i}$ then $\mathtt{A}[\ell] \leq \mathtt{x}$,
   - if $\mathtt{i} + 1 \leq \ell \leq \mathtt{j} - 1$ then $\mathtt{A}[\ell] > \mathtt{x}$,
   - if $\ell = \mathtt{r}$ then $\mathtt{A}[\ell] = \mathtt{x}$.
4. $\mathtt{A}[\mathtt{h}]$ has been unchanged for each integer $\mathtt{h}$ such that $0 \leq \mathtt{h} < \mathtt{p}$ or $\mathtt{r} < \mathtt{h} < \mathtt{A.length}$.
5. Entries of $\mathtt{A}$ are reordered but otherwise unchanged

Application of the Loop Invariant

Suppose again that $\mathtt{p}$ and $\mathtt{r}$ are integers such that $0 \leq \mathtt{p} \leq \mathtt{r} < \mathtt{A.length}$. Then the following properties are satisfied at the *end* of each execution of the body of the `while` loop:

- $\mathtt{j}$ is an integer such that $\mathtt{p} \leq \mathtt{j} \leq \mathtt{r}$
- Parts 2–5 of the loop invariant hold once again

The following properties hold on *termination* of this loop:

- $\mathtt{j} = \mathtt{r}$
- Parts 2–5 of the loop invariant are satisfied

Partial Correctness

Once again, suppose that p and r are integers such that $0 \leq \mathtt{p} \leq \mathtt{r} < \mathtt{A.length}$.

If the program halts then the following conditions are satisfied on termination, if q is the value that is returned:

1. q is an integer such that $\mathtt{p} \leq \mathtt{q} \leq \mathtt{r}$.
2. The following relationships hold for each integer ℓ such that $\mathtt{p} \leq \ell \leq \mathtt{r}$:
   - if $\mathtt{p} \leq \ell < \mathtt{q}$ then $\mathtt{A}[\ell] \leq \mathtt{A}[\mathtt{q}]$,
   - if $\mathtt{q} < \ell \leq \mathtt{r}$ then $\mathtt{A}[\ell] > \mathtt{A}[\mathtt{q}]$.
3. If h is an integer such that $0 \leq \mathtt{h} < \mathtt{p}$ or $\mathtt{r} < \mathtt{h} < \mathtt{A.length}$ then A[h] has not been changed.
4. Entries of A are reordered but otherwise unchanged

Termination and Efficiency

Loop Variant: $r - j$

Justification: decreases, if ≤ 0 loop terminates ($j = r$)

Application:

- The initial value of the loop variant is $r - p$

  Therefore the loop body is executed exactly $r - p$ times.

- Each execution of the loop body requires (at most) a constant number of operations.

  Therefore the cost to execute the loop is in $O(r - p)$.

- Since the rest of the program only uses a constant number of operations, it is clear that the program terminates and that it uses $O(r - p)$ operations in the worst case.

Randomized Partitioning

Idea: Choose the pivot element randomly from the set of values in the part of the array to be processed. Then proceed as before.

`int` **RPartition**(`int [] A, int p, int r`)

Choose i randomly and uniformly from the set of integers between p and r (inclusive).
Swap: $tmp = A[i]$; $A[i] = A[r]$; $A[r] = tmp$
return **DPartition**(A, p, r)

Randomized Partitioning: Analysis

Suppose p and r are integers such that $0 \leq \mathtt{p} \leq \mathtt{r} < \mathtt{A.length}$ and that RPartition is called with inputs A, p and r:

- This algorithm terminates using $O(\mathtt{r} - \mathtt{p})$ operations.
- Let q be the value that is returned on termination. Then the following conditions hold on termination:
  - q is an integer such that $\mathtt{p} \leq \mathtt{q} \leq \mathtt{r}$.
  - If $\ell \in \mathbb{Z}$ and $\mathtt{p} \leq \ell < \mathtt{q}$ then $\mathtt{A}[\ell] \leq \mathtt{A}[\mathtt{q}]$.
  - If $\ell \in \mathbb{Z}$ and $\mathtt{q} < \ell \leq \mathtt{r}$ then $\mathtt{A}[\ell] > \mathtt{A}[\mathtt{q}]$.

  If the entries of A are *distinct* then $\mathtt{q} = \mathtt{i}$ with probability $1/(\mathtt{r} - \mathtt{p} + 1)$ for each integer i between p and r (inclusive).
- If h is an integer such that $0 \leq \mathtt{h} < \mathtt{p}$ or $\mathtt{r} < \mathtt{h} < \mathtt{A.length}$ then A[h] has not been changed.
- Entries of A are reordered but otherwise unchanged

Deterministic Quicksort

Idea: Partition the array, then recursively sort the pieces before and after the pivot element.

Call **quickSort**(A, 0, A.length-1) to sort A:

```
void quickSort(int [] A, int p, int r)
  if p < r then
    q = DPartition(A, p, r)
    quickSort(A, p, q-1)
    quickSort(A, q+1, r)
  end if
```


Randomized Quicksort

Idea: Same as deterministic Quicksort, except that randomized partitioning is used.

Call **RQuickSort**(A, 0, A.length-1) to sort A:

```
void RQuickSort(int [] A, int p, int r)
  if p < r then
    q = RPartition(A, p, r)
    RQuickSort(A, p, q-1)
    RQuickSort(A, q+1, r)
  end if
```


Worst-Case Analysis of Deterministic Quicksort

Theorem 1

Let $T(n)$ be the number of steps used by QuickSort to sort an array of length n in the worst case. Then

$$T(n) \leq \begin{cases} c_0 & \text{if } n \leq 1, \\ c_1 n + \max\limits_{0 \leq k \leq n-1} (T(k) + T(n-1-k)) & \text{if } n \geq 2. \end{cases}$$

Justification: Base case requires constant # of steps. General case:

- constant times n steps for **DPartition**
- maximum of all possible subarray sizes for the recursive calls

Application: This recurrence can be used to prove that $T(n) \in O(n^2)$ using induction on k.

Worst-Case Analysis of Deterministic Quicksort

Theorem 2

If Deterministic Quicksort is applied to an array of length n whose entries are already sorted then this algorithm uses $\Omega(n^2)$ steps.

Method of Proof: Induction on n, once again.

Conclusion: Deterministic Quicksort uses $\Theta(n^2)$ to sort an array of length n in the worst case.

Average-Case Analysis of Deterministic Quicksort

Consider an application of this algorithm when the input is an array A with n **distinct** entries

Consider a binary search tree T storing the same values, with

- the "partition" element at the root
- the left subtree formed by considering the application of the algorithm to the left subarray
- the right subtree formed by considering the application of the algorithm to the right subarray

Useful Property: The number of steps used by the algorithm is at most

$$cn(\text{height}(T) + 1)$$

for some positive constant c

Average-Case Analysis of Deterministic Quicksort

Assumption for Analysis: Entries of A are distinct and all $n!$ relative orderings of these inputs are equally likely

Useful Property: The corresponding binary search trees T are generated with the probability distribution discussed in the "Average Case Analysis of Binary Search Trees."

Bounds on expected height of trees from those notes can now be applied.

Conclusion: The expected cost of Quicksort is in $O(n \log n)$ if the above assumption for analysis is valid.

Analysis of Randomized Quicksort

The previous analysis can be modified to establish that the "worst-case expected cost" of Randomized Quicksort to sort an array **with distinct entries** is in $O(n \log n)$ as well.

Note: it is possible to obtain a *worst-case* running time of $\Theta(n \log n)$

- careful (but deterministic) selection of the pivot (see *Introduction to Algorithms*, Chapter 9.3)

An Annoying Problem

An Annoying Problem: Both versions of Quicksort, given above, use $\Theta(n^2)$ operations to "sort" an array of length n if the array contains n copies of the same value!

The different version of Quicksort found in the textbook has the same problem!

References

References:

- Cormen, Leiserson, Rivest and Stein
  Introduction to Algorithms, Second Edition

  Chapter 7 includes more details, including a complete analysis of the version of Quicksort presented here

- Textbook, Section 11.2