Computer Science 331

Quicksort

Mike Jacobson

Department of Computer Science
University of Calgary

Lecture #29

Outline

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

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

- speed (can be done in time $\Theta(n)$)
- position of p — want the final position of p 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:

- `low` and `high` are integers such that $0 \leq$ `low` $\leq$ `high` $<$ `A.length`

Postcondition:

- Value returned is an integer `q` such that `low` $\leq$ `q` $\leq$ `high`
- `A[h]` $\leq$ `A[q]` for every integer `h` such that `low` $\leq$ `h` $\leq$ `q` $- 1$
- `A[h]` $\geq$ `A[q]` for every integer `h` such that `q` $+ 1 \leq$ `h` $\leq$ `high`
- If `h` is an integer such that $0 \leq$ `h` $<$ `low` or such that `high` $<$ `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 low, int high)
  p = A[high]; i = low; j = high-1
  while i ≤ j do
    while i ≤ j and A[i] ≤ p do
      i = i + 1
    end while
    while j ≥ i and A[j] ≥ p do
      j = j - 1
    end while
    if i < j then
      Swap(A[i], A[j])
    end if
  end while
  Swap(A[i], A[high])
  return i
```


Example

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

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

Using $p = A[10] = 5$ as the pivot. Initially $i = 3$, $j = 9$.

increment i until $i = 4$, decrement j until $j = 9$

$i < j$: swap $A[4]$ & $A[9]$

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

increment i until $i = 7$, decrement j until $j = 8$

$i < j$: swap $A[7]$ & $A[8]$

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

Example (cont.)

increment i until $i = 8$, decrement j until $j = 7$

$i \not< j$: no swap

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

$i = 8$ and $j = 7$: loop terminates

swap $A[8]$ & $A[10]$

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

Loop Invariant (Main Loop)

Suppose `low` and `high` are integers such that $0 \leq$ `low` $\leq$ `high` $<$ length(A) and $p =$ `A[high]`.

The following properties identify a loop invariant for the while loop:

1. `low` $\leq i \leq$ `high` and `low` $- 1 \leq j \leq$ `high` $- 1$
2. `i` $\leq$ `j+1`
3. `A[`ℓ`]` $\leq$ `p` for `low` $\leq \ell \leq$ `i`
4. `A[`ℓ`]` $\geq$ `p` for `j` $\leq \ell \leq$ `high`
5. `A[h]` has been unchanged for each integer `h` such that $0 \leq$ `h` $< \ell$ or `r` $<$ `h` $<$ `A.length`.
6. Entries of `A` are reordered but otherwise unchanged

Partial Correctness

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

Termination and Efficiency

Loop Variant: `j` $+ 1 -$ `i`

Justification: decreases, if ≤ 0 loop terminates ($i = j + 1$)

Efficiency:

- Each location of the array is examined exactly 1 time.
- Each examination requires (at most) a constant number of operations.
- Therefore the cost to execute the loop is in $O($`high` $-$ `low`$)$.
- Since the rest of the program only uses a constant number of operations, it is clear that the program terminates and that it uses $O($`high` $-$ `low`$)$ operations in the worst case.

Quicksort Algorithm

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

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

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


Performance of Deterministic Quicksort

Performance of Quicksort depends on whether the partitioning is **balanced** or **unbalanced**.

- **Worst-case Partitioning:**
  Partition produces two subarrays of size 0 and $n-1$
- **Best-case Partitioning:**
  Partition produces two subarrays of size no more than $n/2$ (one is of size $\lfloor n/2 \rfloor$ and one of size $\lceil n/2 \rceil - 1$)
- **Balanced Partitioning:**
  Partition produces two subarrays of proportional size $\alpha(n-1)$ and $(1-\alpha)(n-1)$ for $0 < \alpha < 1$

Worst-Case Performance of Deterministic Quicksort

Let $T(n)$ be the number of steps used by Quicksort to sort an array of length n in the worst case. Then, for worst-case partitioning

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

Theorem 1

$T(n) \leq nc_0 + \frac{n(n+1)}{2} c_1 - c_1$.

Method of Proof: Induction on n.

Application: Deterministic Quicksort takes $O(n^2)$ steps to sort an array of length n in the worst case.

Worst-Case Performance of Deterministic Quicksort

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

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

Best-Case Performance of Deterministic Quicksort

Let $T(n)$ be the number of steps used by Quicksort to sort an array of length n in the best case. Then, for best-case partitioning

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

Theorem 2

$T(n) \leq nc_0 + (n \log_2 n)c_1$.

Method of Proof: Induction on n.

Application: Deterministic Quicksort takes $O(n \log_2 n)$ steps in the best case to sort an array of length n.

Corollary: Deterministic Quicksort takes $O(n \log_c n)$ steps with balanced partitioning, where $c = \min\{\frac{1}{\alpha}, \frac{1}{1-\alpha}\}$.

Average-Case Analysis of Deterministic Quicksort

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

Result: It can be established that the expected cost of Quicksort is in $O(n \log n)$ if the above assumption for analysis is valid (using heights of binary search trees!).

Intuition:

- In the average case, partition produces a mix of balanced and unbalanced partitions.
- On a random input array, partition is more likely to produce a balanced partition than an unbalanced partition.
- It is unlikely that the partitioning always happens in the same way at every recursion. Thus, unbalanced partitions will result in balanced partitions in subsequent partitions.

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

Efficiency: This algorithm terminates using $O(\texttt{high} - \texttt{low})$ operations.

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 low, int high)
  if low < high then
    q = RPartition(A, low, high)
    RQuickSort(A, low, q-1)
    RQuickSort(A, q+1, high)
  end if
```

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!

References

Further Reading:

- **Introduction to Algorithms**, Chapter 7