Algorithms for Data Science
CSOR W4246

Eleni Drinea
Computer Science Department

Columbia University

Tuesday, September 11, 2014

Outline

1 Recap

2 Quicksort

Review of the last lecture

- Master theorem for solving recurrences
- Binary Search (sub linear algorithm)
- Karatsuba's integer multiplication
- Strassen's fast matrix multiplication

Today

- Quicksort
- Randomized Quicksort

Quicksort

- Quicksort (QS) is a **divide and conquer** algorithm
- It is the standard algorithm used for sorting
- It is an **in-place** algorithm
- Its worst-case running time is $\Theta(n^2)$ but its average-case running time is $\Theta(n \log n)$
- We will use it to introduce randomized algorithms

Quicksort: main idea

Idea: at every recursive call, pick an element from the input, call it *pivot*. Place *pivot* in its final location in the sorted array as follows:

- re-organize the array so that
  - all items smaller than *pivot* are to the left of *pivot*; and
  - all items greater than *pivot* are to its right.
- recursively sort the left subarray of A (where all items are smaller than *pivot*)
- recursively sort the right subarray of A (where all items are greater than *pivot*).

Remark: I haven't explained yet how to pick *pivot*.

Quicksort pseudocode

Quicksort $(A, left, right)$ (initial call: Quicksort$(A, 1, n)$)

if $|A| = 0$ **then** return $//A$ is empty
end if
$split =$ Partition$(A, left, right)$
Quicksort $(A, left, split - 1)$
Quicksort $(A, split + 1, right)$

Partition

Subroutine Partition

1. picks a *pivot* element
2. re-organizes $A(left, right)$ so that
   - all elements before *pivot* are smaller than it;
   - all elements after *pivot* are greater than it.
3. returns the position of *pivot* in the re-organized array in *split*.

Implementing Partition

1. Pick a *pivot* element that will be used to split the input into two parts.
   - For simplicity, always pick the last element of the array as *pivot*, i.e., $pivot = A[right]$.
   - Hence $A[right]$ will be placed at its final location when the subroutine returns: it will never be used (or move) again until the algorithm terminates.
2. Re-organize A **in place**.
   - *What if we didn't care to implement Partition in place?*

Implementing Partition in place

Intuition: Partition examines the elements in $A(left, right)$ one by one and maintains three 3 regions in A. Specifically, after examining the j-th element for $left \leq j \leq right - 1$, these regions are as follows:

1. The first region, starting at $A[left]$ contains all elements encountered so far that are smaller than *pivot*. A pointer *split* points at the last element of this region.
2. The second region, to the right of the first, contains all elements encountered so far that are greater than *pivot*. This region starts at $A[split + 1]$ and ends at $A[j]$.
3. The third region contains the elements that we have not encountered so far. It starts at element $A[j + 1]$ and ends at $A[right - 1]$.

Implementing Partition in place (cont'd)

At every iteration, compare the first element from the third region, that is, element $A[j+1]$ with $pivot$.

If $A[j+1] < pivot$

- swap $A[j+1]$ with element $A[split+1]$: the latter is the first element of the second region, thus is greater than $pivot$; hence it is ok to move it further into the second region (specifically, to the end of the second region);
- increment $split$ to account for the new element $A[j+1]$ added to the first region (recall that this is the region of elements smaller than $pivot$)

Example: $A = \{1, 3, 7, 2, 6, 4, 5\}$

Pseudocode for Partition

Partition($A, left, right$)

$pivot = A[right]$

$split = left - 1$

for $j = left$ to right-1 **do**

  if $A[j] \leq pivot$ **then**

    swap($A[j], A[split + 1]$)

    $split = split + 1$

  end if

end for

swap($pivot, A[split + 1]$) //Place $pivot$ right after $A[split]$

return $split + 1$ //the position of $pivot$

Analysis of Partition: correctness

Claim

For all j ranging from $left$ to $right - 1$, at the end of loop j, all elements from $A[left, \ldots, j]$ that are less or equal to pivot are located in positions $A[left, \ldots, split]$; and all elements from $A[left, \ldots, j]$ that are greater than pivot are located in positions $A[split + 1, \ldots, j]$.

Remark

If the claim is true, correctness of Partition follows by applying the claim for $j = right - 1$.

Partition: proof of correctness

We will show the claim by induction on j.

1. **Base case:** For $j = left$ (that is, during the first execution of the for loop), there are two possibilities:
   - if $A[left] \leq pivot$, then $A[left]$ is swapped with itself and $split$ is incremented to equal $left$
   - otherwise, nothing happens.

   In both cases, the claim holds.

2. **Induction hypothesis:** Assume that the claim is true for some $left \leq j < right - 1$.

Partition: proof of correctness (cont'd)

3. **Induction step:** We will show the claim for $j+1$.
   - At the beginning of loop $j+1$, by the hypothesis, $A[left, \ldots, split]$ are all less or equal to $pivot$ and $A[split+1, \ldots, j]$ are all greater than $pivot$.
   - Inside loop $j+1$, there are two possibilities:
     1. $A[j+1] \leq pivot$: then $A[j+1]$ is swapped with $A[split+1]$. At this point, $A[left, \ldots, split+1]$ are all less or equal to $pivot$ and $A[split+2, \ldots, j+1]$ are all greater than $pivot$. Incrementing $split$ (the next step in the conditional) yields that the claim holds for $j+1$.
     2. $A[j+1] > pivot$: nothing is done. The truth of the claim follows from the hypothesis.

This completes the proof of the inductive step.

Analysis of Partition: running time and space

- **Running time:** on input size n, Partition goes through each of the $n-1$ leftmost elements once and performs constant amount of work per element

  $\Rightarrow$ Partition requires $\Theta(n)$ time.

- **Space:** in-place algorithm

Analysis of Quicksort: correctness

- Quicksort is a recursive algorithm hence we will prove correctness with an inductive argument.
- We will use a variant of induction, called **strong** induction: the induction step at n requires that the inductive hypothesis holds at all steps $1, 2, \ldots, n-1$ and not just at step $n-1$, as with simple induction.
- Strong induction is equivalent to simple induction but is most useful when several instances of the inductive hypothesis are required to show.

Analysis of Quicksort (QS): correctness

We will use strong induction on the size of the array $n \geq 0$.

- **Base case:** for $n = 0$, QS sorts correctly.
- **Induction hypothesis:** Assume that QS correctly sorts an array of size m, for all $0 \leq m < n$.
- **Induction step:** Show that QS correctly sorts for n.
  - Since Partition$(A, 1, n)$ is correct, it will return an index $split$ and a re-organized array A such that all elements in $A[1, \ldots, split - 1]$ are less or equal to $A[split]$ and all elements in $A[split + 1, \ldots, n]$ are greater than $A[split]$.
  - By the hypothesis, QS$(A, left, split - 1)$ and QS$(A, split + 1, n)$ each returns a permutation of its input subarray that is correctly sorted (since each subarray has fewer than n elements). Thus

$$A[1] \leq \ldots \leq A[split - 1] \text{ and } A[split + 1] \leq \ldots \leq A[n].$$

It follows that the whole array is sorted.

Analysis of Quicksort: space and running time

- Space: in-place algorithm
- Running time: depends on the **distribution** of the input elements
  - the sizes of the inputs to the two recursive calls and therefore the form of the recursion depends on how *pivot* compares to the rest of the elements in the input.

Running time: best case

- **Best case:** Suppose that the pivot element during every call to Partition is the *median* of the input. Then Partition splits its input into two lists of almost equal sizes $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$, hence at most $n/2$.
  - This is a "balanced" partitioning.

  Thus
  $$T(n) = 2T(n/2) + \Theta(n) = O(n \log n).$$
  Example of best case: $A = [1 \;\; 3 \;\; 2 \;\; 5 \;\; 7 \;\; 6 \;\; 4]$
- It can be shown that the running time is $O(n \log n)$ for **any** splitting where the two subarrays have sizes that are constant fractions α and $1 - \alpha$ of the original input, for constant $0 < \alpha < 1$.

Running time: worst case

- Upper bound for worst-case running time: $T(n) = O(n^2)$ since
  - there are at most n calls to Partition (one for each element as pivot)
  - Partition takes at most cn time, for some constant c.
- This worst-case upper bound is tight: if every time Partition is called the pivot element is bigger (or smaller) than every other element in the array, (i.e., the array is sorted), then Partition returns one list of size $n-1$ and one list of size 0. This partitioning is very unbalanced.
  - Let $T(0) = d$ for constant $d > 0$. Then for constant $c > 0$:

    $$T(n) = T(n-1) + T(0) + cn \leq T(n-1) + (c+d)n = \Theta(n^2).$$

- Note that the worst-case input is the sorted input.

Running time: average case analysis

Average case: Our input consists of n numbers. What is an "average" input to sorting?

- This really depends on the application:
  - all inputs might be equally probable, i.e., all $n!$ permutations of the n input numbers equally probable; then an average input is *any* of these permutations.
  - the input might be almost sorted.
- In your book there is intuition why average-case analysis for Quicksort is $O(n \log n)$.
- From now on, we will focus on how to use randomness to provide Quicksort with a **random** input. We start with a discussion on how randomness affects the analysis of the running times of algorithms.

Two ways of viewing randomness in computation

1. Our algorithm is deterministic but the world behaves randomly.

- **Algorithm:** given the same input, our algorithm always produces the same output using the same time. Hence the running time solely depends on the input.
  - This is the kind of algorithms we've encountered so far.
- **Input:** The input to our algorithm is randomly generated: there is some underlying distribution that generates it.
- **Average case** analysis: analyzing the running time of the algorithm on an average input

Two ways of viewing randomness in computation (cont'd)

2. Our algorithm behaves randomly.
   - **Algorithm:** given the same input the algorithm produces the same output but different executions of the algorithm might require different running times because the latter depend on the **random choices** of the algorithm;
     - the algorithm may flip coins or generate random numbers; we assume that our random samples are independent of each other.
   - **Input:** the world provides its worst-case input.
   - **Expected** running time: analysis of the running time of the randomized algorithm on such a worst-case input.

Remarks on randomness in computation

1. Allowing randomness within the algorithm strictly empowers the algorithm: e.g., thanks to randomness a worst-case input could be transformed into a random input, i.e., one that is closer to an "average" input.
2. Deterministic algorithms are a special case of randomized algorithms.

Randomized Quicksort

So how can use randomization so that Quicksort works with a random input even when it receives a worst-case input?

1. We could explicitly permute the input: given the input, generate a random permutation of it.
2. Another way that yields a simpler analysis is to use **random sampling** for choosing *pivot*: instead of always choosing $pivot = A[right]$ we will randomly select an element from $A[left, right]$ as *pivot*.

Idea (intuition behind random sampling)

Essentially, this makes sure that no matter how our input is organized, we won't often pick the largest or smallest element as our pivot element (unless we are really, really unlucky). Thus we expect that most often the two subarrays that Partition returns will be "balanced" in size and we'll be closer to the analysis and the running time for the best case.

Pseudocode for Randomized Partition

Randomized Partition($A, left, right$)

$b =$ random($left, right$)
swap($A[b], A[right]$)
return Partition($A, left, right$)

where random($left, right$) returns a random number between $left$ and $right$ inclusive. In the second hour we will rigorously analyze the expected running time of this Randomized-QS.