Analysis of Algorithms, I
CSOR S4231

Eleni Drinea
Computer Science Department

Columbia University

Thursday, January 28, 2016

Outline

Today

1. Recap
2. Quicksort
3. Randomized Quicksort
4. Random variables and linearity of expectation

Review of the last lecture

- More divide & conquer algorithms
  - Binary search
  - Karatsuba's integer multiplication
  - Strassen's matrix multiplication
  - Quicksort (the main idea)

Today

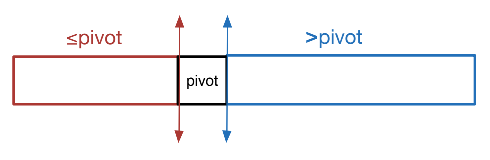
1. Recap
2. Quicksort
3. Randomized Quicksort
4. Random variables and linearity of expectation

Quicksort facts

- Quicksort 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

- Pick an input item, call it *pivot*, and place it in its final location in the sorted array by re-organizing the array:
  - all items $\leq$ *pivot* are placed before *pivot*
  - all items $>$ *pivot* are placed after *pivot*



- Recursively sort the subarray to the left of *pivot* (items $\leq$ *pivot*).
- Recursively sort the subarray to the right of *pivot* (items $>$ *pivot*).

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

Quicksort pseudocode

```
Quicksort(A, left, right)
   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)
```

Initial call: `Quicksort`$(A, 1, n)$

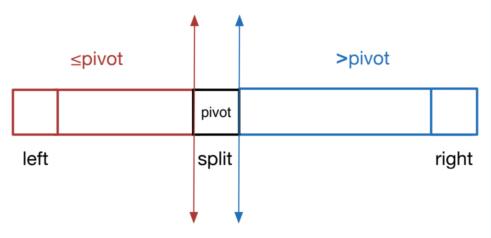
Subroutine `Partition`$(A, left, right)$

Notation: $A[a, b]$ denotes the portion of A starting at position a and ending at position b.

`Partition`

1. picks a *pivot* item
2. re-organizes $A[left, right]$ so that
   - all items before *pivot* are $\leq$ *pivot*
   - all items after *pivot* are $>$ *pivot*
3. returns *split*, the index of *pivot* in the re-organized array

After `Partition`, $A[left, right]$ looks as follows:



Implementing `Partition`

1. Pick a *pivot* item: for simplicity, always pick the last item of the array as *pivot*, i.e., $pivot = A[right]$.
   - Thus $A[right]$ will be placed in its final location when `Partition` returns and will never be used (or moved) again until the algorithm terminates.

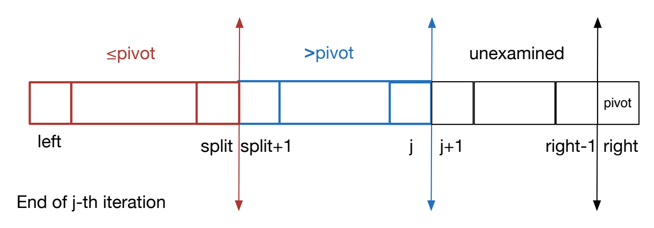
2. Re-organize the input array A in place.

(What if we didn't care to implement `Partition` *in place?)*

Implementing `Partition` in place

`Partition` examines the items in $A[left, right]$ one by one and maintains three regions in A. Specifically, **after** examining the j-th item for $j \in [left, right-1]$, the regions are:

1. First region: starts at $A[left]$ and ends at $A[split]$; it contains all items examined so far that are $< pivot$.
2. Second region: starts at $A[split+1]$ and ends at $A[j]$; it contains all items $> pivot$ examined so far;
3. Third region: starts at $A[j+1]$ and ends at $A[right-1]$; it contains all items yet to be examined.



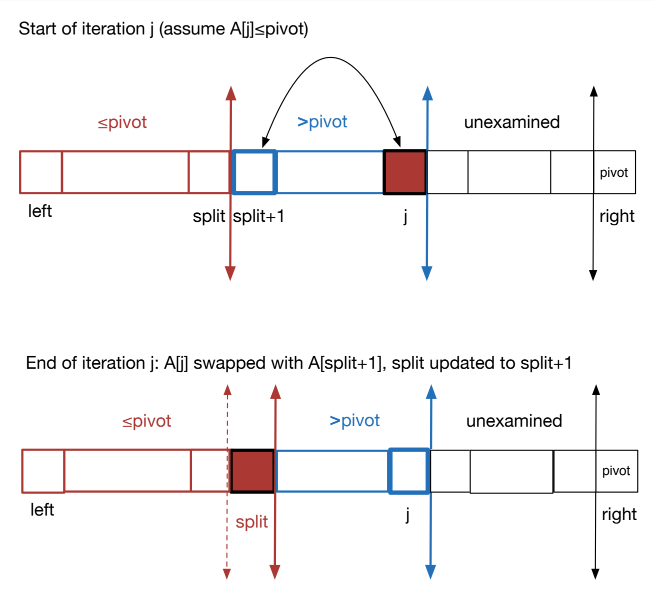
Implementing `Partition` in place (cont'd)

At the beginning of iteration j, $A[j]$ is compared with $pivot$.

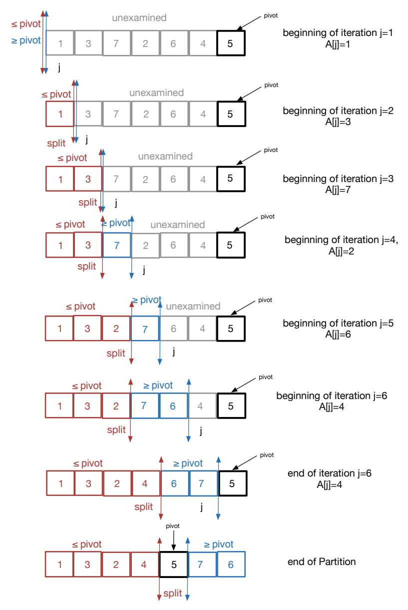
If $A[j] \leq pivot$

1. swap $A[j]$ with $A[split+1]$, the first element of the second region: since $A[split+1] > pivot$, it's "safe" to move it to the end of the second region (recall that the second region ends at position j after examining the j-th item);
2. increment $split$ to include $A[j]$ in the first region, where all items are $\leq pivot$ (recall that the first region ends at $split$).

Iteration j: when $A[j] \leq pivot$



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



Pseudocode for Partition

```
Partition(A, left, right)
    pivot = A[right]
    split = left − 1
    for  j = left to right − 1  do
        if  A[j] ≤ pivot  then
            swap(A[j], A[split + 1])
            split = split + 1
        end if
    end for
    swap(pivot, A[split + 1])      //place pivot after A[split] (why?)
    return split + 1               //the final position of pivot
```

Analysis of `Partition`: correctness

Notation: $A[a, \ldots, b]$ denotes the portion of A starting at position a and ending at position b.

Claim 1.

For $left \leq j \leq right - 1$, at the end of loop j,

1. *all items in $A[left, split]$ are $\leq pivot$; and*
2. *all items in $A[split + 1, j]$ are $> pivot$*

Remark: If the claim is true, correctness of `Partition` follows *(why?)*.

Proof of Claim 1

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 for $j = left$.

2. **Hypothesis:** Assume that the claim is true for some $left \leq j < right - 1$.
   - That is, at the end of loop j, all items in $A[left, split]$ are $\leq pivot$ and all items in $A[split + 1, j]$ are $> pivot$.

Proof of Claim 1 (cont'd)

3. **Step:** We will show the claim for $j+1$. That is, we will show that after loop $j+1$, all items in $A[left, split]$ are $\leq pivot$ and all items in $A[split+1, j+1]$ are $> pivot$.

- At the beginning of loop $j+1$, by the hypothesis, items in $A[left, split]$ are $\leq pivot$ and items in $A[split+1, j]$ are $> pivot$.
- 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, items in $A[left, split+1]$ are $\leq pivot$ and items in $A[split+2, j+1]$ are $> pivot$. Incrementing $split$ (the next step in the pseudocode) 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; we will prove correctness by induction on the input size n.
- We will use **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.

Analysis of `Quicksort`: correctness

- **Base case:** for $n = 0$, `Quicksort` sorts correctly.
- **Hypothesis:** for all $0 \leq m < n$, `Quicksort` correctly sorts on input size m.
- **Step:** show that `Quicksort` correctly sorts on input size n.
  - `Partition`$(A, 1, n)$ re-organizes A so that all items
    - in $A[1, \ldots, split - 1]$ are $\leq A[split]$;
    - in $A[split + 1, \ldots, n]$ are $> A[split]$.
  - Next, `Quicksort`$(A, 1, split - 1)$, `Quicksort`$(A, split + 1, n)$ will correctly sort their inputs (by the hypothesis). Hence

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

At this point, `Quicksort` returns and A is sorted.

Analysis of `Quicksort`: space and running time

- **Space**: in-place algorithm
- **Running time**: depends on the arrangement of the input elements
  - the sizes of the inputs to the two recursive calls –hence the form of the recurrence– depend on how *pivot* compares to the rest of the input items

Notation: we will denote the running time of `Quicksort` by $T(n)$.

Running time of `Quicksort`: best case

Suppose that in every call to `Partition` the pivot item is the median of the input.

Then every `Partition` splits its input into two lists of almost equal sizes, thus

$$T(n) = 2T(n/2) + \Theta(n) = O(n \log n).$$

This is a "balanced" partitioning.

- Example of best case: $A = [1 \;\; 3 \;\; 2 \quad 5 \;\; 7 \;\; 6 \quad 4]$

Remark: Can show that $T(n) = O(n \log n)$ for any splitting where the two subarrays have sizes αn, $(1-\alpha)n$ respectively, for constant $0 < \alpha < 1$.

Running time of `Quicksort`: worst case

- Upper bound for worst-case running time: $T(n) = O(n^2)$
  - at most n calls to Partition (one for each item as pivot)
  - `Partition` requires $O(n)$ time
- This worst-case upper bound is tight:
  - If every time `Partition` is called *pivot* is greater (or smaller) than every other item, then its input is split into two lists, one of which has size 0.
    - This partitioning is very "unbalanced": let $c, d > 0$ be constants, where $T(0) = d$; then

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

△ The worst-case input is the sorted input!

Running time: average case analysis

Average case: what is an "average" input to sorting?

- Depends on the application.
- In your textbook 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.

Today

1. Recap
2. Quicksort
3. Randomized Quicksort
4. Random variables and linearity of expectation

Two views of randomness in computation

1. Deterministic algorithm, randomness over the inputs
   - On the same input, the algorithm always produces the same output using the same time.
     - So far, we have only encountered such algorithms.
   - The input is randomly generated according to some underlying distribution.
   - Average case analysis: analysis of the running time of the algorithm on an average input.

Two views of randomness in computation (cont'd)

2. Randomized algorithm, worst-case (deterministic) input
    - On the same input, the algorithm produces the same output but different executions may require different running times.
        - The latter depend on the random choices of the algorithm (e.g., coin flips, random numbers).
        - Random samples are assumed independent of each other.
    - Worst-case input
    - Expected running time analysis: analysis of the running time of the randomized algorithm on a worst-case input.

Remarks on randomness in computation

1. Deterministic algorithms are a special case of randomized algorithms.
2. Randomized algorithms are more powerful than deterministic ones.

Randomized Quicksort

Can we use randomization so that Quicksort works with an "average" input even when it receives a worst-case input?

1. Explicitly permute the input.
2. Use random sampling to choose *pivot*: instead of using $A[right]$ as *pivot*, select *pivot* randomly.

Idea 1 (intuition behind random sampling).

No matter how the input is organized, we won't often pick the largest or smallest item as pivot (unless we are really, really unlucky). Thus most often the partitioning will be "balanced".

Pseudocode for randomized Quicksort

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

Randomized-Partition(A, left, right)
  b = random(left, right)
  swap(A[b], A[right])
  return Partition(A, left, right)
```

Subroutine `random`(i, j) returns a random number between $left$ and $right$ inclusive.

Today

1 Recap

2 Quicksort

3 Randomized Quicksort

4 Random variables and linearity of expectation

Discrete random variables

- To analyze the expected running time of a randomized algorithm we keep track of certain parameters and their expected size over the random choices of the algorithm.
- To this end, we use random variables.
- A discrete random variable X takes on a finite number of values, each with some probability. We're interested in its expectation
$$E[X] = \sum_j j \cdot \Pr[X = j].$$

Example 1: Bernoulli trial

Experiment 1: flip a biased coin which comes up

- *heads* with probability p
- *tails* with probability $1 - p$

Question: what is the expected number of *heads*?

Example 1: Bernoulli trial

Experiment 1: flip a biased coin which comes up

- *heads* with probability p
- *tails* with probability $1 - p$

Question: what is the expected number of *heads*?

Let X be a random variable such that

$$X = \begin{cases} 1 & \text{, if coin flip comes } \textit{heads} \\ 0 & \text{, if coin flip comes } \textit{tails} \end{cases}$$

Example 1: Bernoulli trial

Experiment 1: flip a biased coin which comes up

- *heads* with probability p
- *tails* with probability $1 - p$

Question: what is the expected number of *heads*?

Let X be a random variable such that

$$X = \begin{cases} 1 & \text{, if coin flip comes } \textit{heads} \\ 0 & \text{, if coin flip comes } \textit{tails} \end{cases}$$

Then

$$\Pr[X = 1] = p$$
$$\Pr[X = 0] = 1 - p$$
$$E[X] = 1 \cdot \Pr[X = 1] + 0 \cdot \Pr[X = 0] = p$$

Indicator random variables

- Indicator random variable: a discrete random variable that only takes on values 0 and 1.
- Indicator random variables are used to denote occurrence (or not) of an event.

  Example: in the biased coin flip example, X is an indicator random variable that denotes the occurrence of *heads*.

Fact 1.

If X is an indicator random variable, then $E[X] = \Pr[X = 1]$.

Example 2: Bernoulli trials

Experiment 2: flip the biased coin n times

Question: what is the expected number of *heads*?

Example 2: Bernoulli trials

Experiment 2: flip the biased coin n times

Question: what is the expected number of *heads*?

Answer 1: Let X be the random variable counting the number of times *heads* appears.

$$E[X] = \sum_{j=0}^{n} j \cdot \Pr[X = j].$$

$\Pr[X = j]$?

Example 2: Bernoulli trials

Experiment 2: flip the biased coin n times

Question: what is the expected number of *heads*?

Answer 1: Let X be the random variable counting the number of times *heads* appears.

$$E[X] = \sum_{j=0}^{n} j \cdot \Pr[X = j].$$

$\Pr[X = j]$?

X follows the binomial distribution $B(n, p)$, thus

$$\Pr[X = j] = \binom{n}{j} p^j (1-p)^{n-j}$$

Example 2: Bernoulli trials

A different way to think about X:

Answer 2: for $1 \leq i \leq n$, let X_i be an indicator random variable such that

$$X_i = \begin{cases} 1 & \text{, if } i\text{-th coin flip comes } \textit{heads} \\ 0 & \text{, if } i\text{-th coin flip comes } \textit{tails} \end{cases}$$

Example 2: Bernoulli trials

A different way to think about X:

Answer 2: for $1 \leq i \leq n$, let X_i be an indicator random variable such that

$$X_i = \begin{cases} 1 & \text{, if } i\text{-th coin flip comes } \textit{heads} \\ 0 & \text{, if } i\text{-th coin flip comes } \textit{tails} \end{cases}$$

Define the random variable

$$X = \sum_{i=1}^{n} X_i$$

By Fact 1, $E[X_i] = p$, for all i. We want $E[X]$.

Linearity of expectation

$$X = \sum_{i=1}^{n} X_i, \quad E[X_i] = p, \quad E[X] = ?$$

Linearity of expectation

$$X = \sum_{i=1}^{n} X_i, \quad E[X_i] = p, \quad E[X] = ?$$

Remark 1: X is a complicated random variable defined as the sum of simpler random variables whose expectation is known.

Linearity of expectation

$$X = \sum_{i=1}^{n} X_i, \quad E[X_i] = p, \quad E[X] = ?$$

Remark 1: X is a complicated random variable defined as the sum of simpler random variables whose expectation is known.

Proposition 1 (Linearity of expectation).

Let $X_1, \ldots, X_k$ be arbitrary random variables. Then

$$E[X_1 + X_2 + \ldots + X_k] = E[X_1] + E[X_2] + \ldots + E[X_k]$$

Linearity of expectation

$$X = \sum_{i=1}^{n} X_i, \quad E[X_i] = p, \quad E[X] = ?$$

Remark 1: X is a complicated random variable defined as the sum of simpler random variables whose expectation is known.

Proposition 1 (Linearity of expectation).

Let $X_1, \ldots, X_k$ be arbitrary random variables. Then

$$E[X_1 + X_2 + \ldots + X_k] = E[X_1] + E[X_2] + \ldots + E[X_k]$$

Remark 2: We made no assumptions on the random variables. For example, they do not need be independent.

Back to example 2: Bernoulli trials

Answer 2: for $1 \leq i \leq n$, let X_i be an indicator random variable such that

$$X_i = \begin{cases} 1 & \text{, if } i\text{-th coin flip comes } \textit{heads} \\ 0 & \text{, if } i\text{-th coin flip comes } \textit{tails} \end{cases}$$

Define the random variable

$$X = \sum_{i=1}^{n} X_i$$

By Fact 1, $E[X_i] = p$, for all i. By linearity of expectation,

$$E[X] = E\Big[\sum_{i=1}^{n} X_i\Big] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} p = np.$$