Analysis of Algorithms, I
CSOR W4231.002

Eleni Drinea
Computer Science Department

Columbia University

Tuesday, January 26, 2016

Outline

1 Recap

2 Binary search

3 Integer multiplication

4 Fast matrix multiplication

5 Quicksort

Today

1. Recap
2. Binary search
3. Integer multiplication
4. Fast matrix multiplication
5. Quicksort

Review of the last lecture

- Asymptotic notation $(O, \Omega, \Theta, o, \omega)$
- The divide & conquer principle
  - **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
  - **Conquer** the subproblems by solving them recursively.
  - **Combine** the solutions to the subproblems into the solution for the original problem.
- Application: `Mergesort`
- Solving recurrences

Mergesort

```
Mergesort (A, left, right)
   if right == left then
      return
   end if
   middle = left + ⌊(right − left)/2⌋
   Mergesort (A, left, middle)
   Mergesort (A, middle + 1, right)
   Merge (A, left, middle, right)
```

- Initial call: `Mergesort`$(A, 1, n)$
- Subroutine `Merge` merges two sorted lists of sizes $\lceil n/2 \rceil$, $\lfloor n/2 \rfloor$ into one sorted list of size n in time $\Theta(n)$.

Running time of `Mergesort`

The running time of `Mergesort` satisfies:

$$T(n) = 2T(n/2) + cn, \text{ for } n \geq 2, \text{ constant } c > 0$$
$$T(1) = c$$

This structure is typical of recurrence relations:

- an inequality or equation bounds $T(n)$ in terms of an expression involving $T(m)$ for $m < n$
- a base case generally says that $T(n)$ is constant for small constant n

Remarks

- We ignore floor and ceiling notations
- A recurrence does **not** provide an asymptotic bound for $T(n)$: to this end, we must solve the recurrence

Solving recurrences, method 1: recursion trees

The technique consists of three steps

1. Analyze the first few levels of the tree of recursive calls
2. Identify a pattern
3. Sum over all levels of recursion

Example: analysis of running time of `Mergesort`

$T(n) = 2T(n/2) + cn, n \geq 2$

$T(1) = c$

A general recurrence and its solution

Often we can express the running time of a recursive algorithm by the following recurrence

$$T(n) = aT(n/b) + cn^k, \text{for } a, c > 0,\ b > 1, k \geq 0$$

What is the recursion tree for this recurrence?

- a is the branching factor
- b is the factor by which the size of each subproblem shrinks

$\Rightarrow$ at level i, there are a^i subproblems, each of size n/b^i

$\Rightarrow$ each subproblem at level i requires $c(n/b^i)^k$ work

- the height of the tree is $\log_b n$ levels

$\Rightarrow$ Total work: $\sum_{i=0}^{\log_b n} a^i c(n/b^i)^k = cn^k \sum\limits_{i=0}^{\log_b n} \left(\frac{a}{b^k}\right)^i$

Solving recurrences, method 2: Master theorem

Theorem 1 (Master theorem).

If $T(n) = aT(\lceil n/b \rceil) + O(n^k)$ *for some constants* $a > 0$, $b > 1$, $k \geq 0$, *then*

$$T(n) = \begin{cases} O(n^{\log_b a}) & , \text{ if } a > b^k \\ O(n^k \log n) & , \text{ if } a = b^k \\ O(n^k) & , \text{ if } a < b^k \end{cases}$$

Example: running time of `Mergesort`

- $T(n) = 2T(n/2) + cn$:
  $a = 2, b = 2, k = 1, b^k = 2 = a \Rightarrow T(n) = O(n \log n)$

Today

1. Recap
2. Binary search
3. Integer multiplication
4. Fast matrix multiplication
5. Quicksort

Searching a sorted array

- **Input: sorted** list A of n integers, integer x
- **Output:**
  1. index j s.t. $1 \leq j \leq n$ and $A[j] = x$; or
  2. **no** if x is not in A

Searching a sorted array

- **Input: sorted** list A of n integers, integer x
- **Output:**
  1. index j s.t. $1 \leq j \leq n$ and $A[j] = x$; or
  2. **no** if x is not in A

Example: $A = \{0, 2, 3, 5, 6, 7, 9, 11, 13\}$, $n = 9$, $x = 7$

Searching a sorted array

- **Input:** **sorted** list A of n integers, integer x
- **Output:**
  1. index j s.t. $1 \leq j \leq n$ and $A[j] = x$; or
  2. **no** if x is not in A

Example: $A = \{0, 2, 3, 5, 6, 7, 9, 11, 13\}$, $n = 9$, $x = 7$

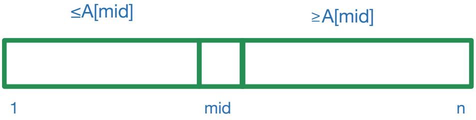
Idea: use the fact that the array is **sorted** and probe specific entries in the array.

Binary search

First, probe the middle entry. Let $mid = \lceil n/2 \rceil$.

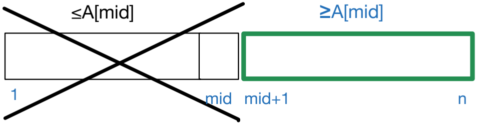
- If $x == A[mid]$, return mid.
- If $x < A[mid]$ then look for x in $A[1, mid - 1]$;
- Else if $x > A[mid]$ look for x in $A[mid + 1, n]$.

Initially, the entire array is "active", that is, x might be anywhere in the array.



Suppose x > A[mid].

Then the active area of the array, where x might be, is to the right of mid.



Binary search pseudocode

```
binarysearch(A, left, right)
   mid = left + ⌈(right − left)/2⌉
   if A[mid] == x then
      return mid
   else if right == left then
      return no
   else if A[mid] < x  then
      left = mid + 1
   else right = mid − 1
   end if
   binarysearch(A, left, right)
```

Binary search running time

Observation: At each step there is a region of A where x could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If n is odd, then we are throwing away $\lceil n/2 \rceil$ elements.
- If n is even, then we are throwing away at least $n/2$ elements.

Binary search running time

Observation: At each step there is a region of A where x could be and we **shrink** the size of this region by a factor of 2 with every probe:

- If n is odd, then we are throwing away $\lceil n/2 \rceil$ elements.
- If n is even, then we are throwing away at least $n/2$ elements.

Hence the recurrence for the running time is

$$T(n) \leq T(n/2) + O(1)$$

Sublinear running time

Here are two ways to argue about the running time:

1. Master theorem: $b = 2, a = 1, k = 0 \Rightarrow T(n) = O(\log n)$.
2. We can reason as follows:
   - Starting with an array of size n, after k probes, we are left with an array of size at most $\frac{n}{2^k}$ (since every time we probe an entry the active portion of the array halves).
   - Hence after $k = \log n$ probes, we are left with an array of **constant** size (i.e., $O(1)$). Now we can search **linearly** for x in the constant size array.

Concluding remarks on binary search

1. The right data structure can improve the running time of the algorithm significantly.
    - *What if we used a* **linked list** *to store the input?*
    - Arrays allow for **random access** of their elements: given an index, we can read any entry in an array in time $O(1)$ (constant time)

2. In general, we obtain running time $O(\log n)$ when the algorithm does a **constant amount of work** to throw away a **constant fraction** of the input.

Today

1. Recap
2. Binary search
3. Integer multiplication
4. Fast matrix multiplication
5. Quicksort

Integer multiplication

- *How do we multiply two integers x and y?*
- Elementary school method: compute a partial product by multiplying every digit of y separately with x and then add up all the partial products.
- Remark: this method works the same in base 10 or base 2.

Examples: $(12)_{10} \cdot (11)_{10}$ and $(1100)_2 \cdot (1011)_2$

```
     12              1100
  ×  11           ×  1011
  -----           -------
     12              1100
  +  12             1100
  -----            0000
    132         + 1100
                ---------
                 10000100
```

Elementary algorithm running time

A more reasonable model of computation: a **single** operation on a pair of digits (bits) is a primitive computational step.

Assume we are multiplying n-digit (bit) numbers.

- $O(n)$ time to compute a partial product.
- $O(n)$ time to combine it in a running sum of all partial products so far.

$\Rightarrow$ There are n partial products, each consisting of n bits, hence total number of operations is $O(n^2)$.

Can we do better?

A first divide & conquer approach

Consider n-digit decimal numbers x, y.

$$
\begin{aligned}
x &= x_{n-1}x_{n-2}\dots x_0 \\
y &= y_{n-1}y_{n-2}\dots y_0
\end{aligned}
$$

Idea: rewrite each number as the sum of the $n/2$ high-order digits and the $n/2$ low-order digits.

$$
\begin{aligned}
x &= \underbrace{x_{n-1}\dots x_{n/2}}_{x_H}\underbrace{x_{n/2-1}\dots x_0}_{x_L} = x_H \cdot 10^{n/2} + x_L \\
y &= \underbrace{y_{n-1}\dots y_{n/2}}_{y_H}\underbrace{y_{n/2-1}\dots y_0}_{y_L} = y_H \cdot 10^{n/2} + y_L
\end{aligned}
$$

where each of x_H, x_L, y_H, y_L is an $n/2$-digit number.

Examples

- $n = 2$, $x = 12$, $y = 11$

$$\begin{aligned}\underbrace{12}_{x} &= \underbrace{1}_{x_H} \cdot \underbrace{10^1}_{10^{n/2}} + \underbrace{2}_{x_L} \\ \underbrace{11}_{y} &= \underbrace{1}_{y_H} \cdot \underbrace{10^1}_{10^{n/2}} + \underbrace{1}_{y_L}\end{aligned}$$

- $n = 4$, $x = 1000$, $y = 1110$

$$\begin{aligned}\underbrace{1000}_{x} &= \underbrace{10}_{x_H} \cdot \underbrace{10^2}_{10^{n/2}} + \underbrace{0}_{x_L} \\ \underbrace{1110}_{y} &= \underbrace{11}_{y_H} \cdot \underbrace{10^2}_{10^{n/2}} + \underbrace{10}_{y_L}\end{aligned}$$

A first divide & conquer approach

$$\begin{aligned} x \cdot y &= (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_L) \\ &= x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) 10^{n/2} + x_L y_L \end{aligned}$$

In words, we reduced the problem of solving 1 instance of size n (i.e., one multiplication between two n-digit numbers) to the problem of solving 4 instances, each of size $n/2$ (i.e., computing the products $x_H y_H, x_H y_L, x_L y_H$ and $x_L y_L$).

A first divide & conquer approach

$$\begin{aligned} x \cdot y &= (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_L) \\ &= x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) 10^{n/2} + x_L y_L \end{aligned}$$

In words, we reduced the problem of solving 1 instance of size n (i.e., one multiplication between two n-digit numbers) to the problem of solving 4 instances, each of size $n/2$ (i.e., computing the products $x_H y_H, x_H y_L, x_L y_H$ and $x_L y_L$).

This is a divide and conquer solution!

- Recursively solve the 4 subproblems.
- Multiplication by 10^n is easy (**shifting**): $O(n)$ time.
- Combine the solutions from the 4 subproblems to an overall solution using 3 additions on $O(n)$-digit numbers: $O(n)$ time.

Karatsuba's observation

Running time: $T(n) \leq 4T(n/2) + cn$

- by the Master Theorem: $T(n) = O(n^2)$
- **no** improvement

Karatsuba's observation

Running time: $T(n) \leq 4T(n/2) + cn$

- by the Master Theorem: $T(n) = O(n^2)$
- **no** improvement

However, if we only needed three $n/2$-digit multiplications, then by the Master theorem

$$T(n) \leq 3T(n/2) + cn = O(n^{1.59}) = o(n^2).$$

Karatsuba's observation

Running time: $T(n) \leq 4T(n/2) + cn$

- by the Master Theorem: $T(n) = O(n^2)$
- **no** improvement

However, if we only needed three $n/2$-digit multiplications, then by the Master theorem

$$T(n) \leq 3T(n/2) + cn = O(n^{1.59}) = o(n^2).$$

Recall that

$$x \cdot y \quad = \quad x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) 10^{n/2} + x_L y_L$$

Key observation: we do not need each of $x_H y_L, x_L y_H$.
We only need their sum, $x_H y_L + x_L y_H$.

Gauss's observation on multiplying complex numbers

A similar problem: multiply two complex numbers $a + bi, c + di$

$$(a + bi)(c + di) = ac + (ad + bc)i + bdi^2$$

Gauss's observation on multiplying complex numbers

A similar problem: multiply two complex numbers $a + bi, c + di$

$$(a + bi)(c + di) = ac + (ad + bc)i + bdi^2$$

Gauss's observation: can be done with just 3 multiplications

$$(a + bi)(c + di) = ac + ((a + b)(c + d) - ac - bd)i + bdi^2,$$

at the cost of few extra additions and subtractions.

$*$ Unlike multiplications, additions and subtractions of n-digit numbers are cheap: $O(n)$ time!

Karatsuba's algorithm

$$
\begin{aligned}
x \cdot y &= (x_H 10^{n/2} + x_L) \cdot (y_H 10^{n/2} + y_H) \\
&= x_H y_H \cdot 10^n + (x_H y_L + x_L y_H) 10^{n/2} + x_L y_L
\end{aligned}
$$

Similarly to Gauss's method for multiplying two complex numbers, compute only the three products

$$x_H y_H, \; x_L y_L, \; (x_H + x_L)(y_H + y_L)$$

and obtain the sum $x_H y_L + x_L y_H$ from

$$(x_H + x_L)(y_H + y_L) - x_H y_H - x_L y_L = x_H y_L + x_L y_H.$$

Combining requires $O(n)$ time hence

$$T(n) \leq 3T(n/2) + cn = O(n^{\log_2 3}) = O(n^{1.59})$$

Pseudocode

Let k be a small constant.

```
Integer-Multiply(x, y)
   if n == k then
       return xy
   end if
   write x = x_H 10^{n/2} + x_L, y = y_H 10^{n/2} + y_L
   compute x_H + x_L, y_H + y_L
   product = Integer-Multiply(x_H + x_L, y_H + y_L)
   x_H y_H = Integer-Multiply(x_H, y_H)
   x_L y_L = Integer-Multiply(x_L, y_L)
   return x_H y_H 10^n + (product − x_H y_H − x_L y_L) 10^{n/2} + x_L y_L
```

Concluding remarks

- To reduce the number of multiplications we do few more additions/subtractions: these are fast compared to multiplications.
- There is no reason to continue with recursion once n is small enough: the conventional algorithm is probably more efficient since it uses fewer additions.
- When we recursively compute $(x_H + x_L)(y_H + y_L)$, each of $x_H + x_L$, $y_H + y_L$ might be $(n/2 + 1)$-digit integers. This does not affect the asymptotics.

Today

1 Recap

2 Binary search

3 Integer multiplication

4 Fast matrix multiplication

5 Quicksort

Fast matrix multiplication

Matrix multiplication: a fundamental primitive in numerical linear algebra, scientific computing, machine learning and large-scale data analysis.

- Input: $m \times n$ matrix A, $n \times p$ matrix B
- Output: $m \times p$ matrix $C = AB$

Example: $A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, B = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, C = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

Lower bounds on matrix multiplication algorithms for $m, p = \Theta(n)$*?*

Conventional matrix multiplication

for $1 \leq i \leq m$ **do**
    for $1 \leq j \leq p$ **do**
        $c_{i,j} = 0$
        for $1 \leq k \leq n$ **do**
            $c_{i,j} += a_{i,k} \cdot b_{k,j}$
        end for
    end for
end for

- *Running time?*
- *Can we do better?*

A first divide & conquer approach: 8 subproblems

Assume square A, B where $n = 2^k$ for some $k > 0$.
Idea: express A, B as 2×2 block matrices and use the conventional algorithm to multiply the two block matrices.

$$\begin{pmatrix} \overbrace{A_{11}}^{n/2 \times n/2} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

Running time?

Strassen's breakthrough: 7 subproblems suffice (part 1)

Compute the following ten $n/2 \times n/2$ matrices.

1. $S_1 = B_{11} - B_{22}$
2. $S_2 = A_{11} + A_{12}$
3. $S_3 = A_{21} + A_{22}$
4. $S_4 = B_{21} - B_{11}$
5. $S_5 = A_{11} + A_{22}$
6. $S_6 = B_{11} + B_{22}$
7. $S_7 = A_{12} - A_{22}$
8. $S_8 = B_{21} + B_{22}$
9. $S_9 = A_{11} - A_{21}$
10. $S_{10} = B_{11} + B_{12}$

Running time?

Strassen's breakthrough: 7 subproblems suffice (part 2)

Compute the following seven products of $n/2 \times n/2$ matrices.

1. $P_1 = A_{11}S_1$
2. $P_2 = S_2B_{22}$
3. $P_3 = S_3B_{11}$
4. $P_4 = A_{22}S_4$
5. $P_5 = S_5S_6$
6. $P_6 = S_7S_8$
7. $P_7 = S_9S_{10}$

Compute C as follows:

1. $C_{11} = P_4 + P_5 + P_6 - P_2$
2. $C_{12} = P_1 + P_2$
3. $C_{21} = P_3 + P_4$
4. $C_{22} = P_1 + P_5 - P_3 - P_7$

Running time?

Strassen's running time and concluding remarks

- Recurrence: $T(n) = 7T(n/2) + cn^2$
- By the Master theorem:

$$T(n) = O(n^{\log_2 7}) = O(n^{2.81})$$

- Recently, there is renewed interest in Strassen's algorithm for **high-performance computing**: thanks to its lower communication cost (number of bits exchanged between machines in the network or data center), it is better suited than the traditional algorithm for multi-core processors.

Today

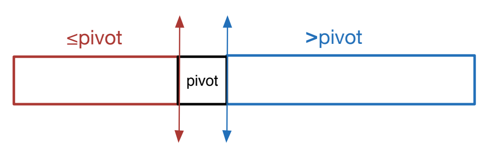
1. Recap
2. Binary search
3. Integer multiplication
4. Fast matrix multiplication
5. Quicksort

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

$\texttt{Quicksort}(A, left, right)$

  if $|A| = 0$ **then** return //A is empty

  end if

  $split = \texttt{Partition}(A, left, right)$

  $\texttt{Quicksort}(A, left, split - 1)$

  $\texttt{Quicksort}(A, split + 1, right)$

Initial call: $\texttt{Quicksort}(A, 1, n)$

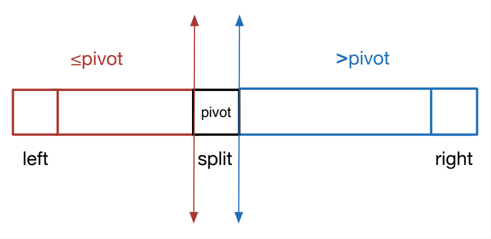
Subroutine $\texttt{Partition}(A, left, right)$

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

`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.

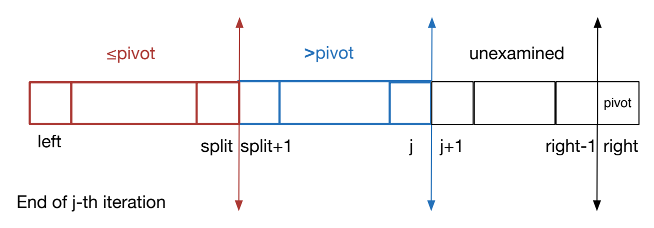
2. Re-organize the input array A in place.

(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 $\leq pivot$.
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)

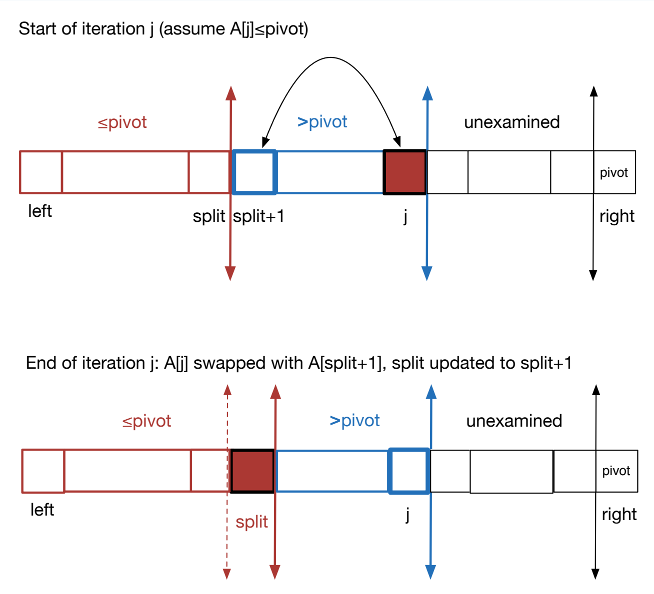
At the beginning of iteration j, $A[j]$ is compared with $pivot$.

If $A[j] \leq pivot$

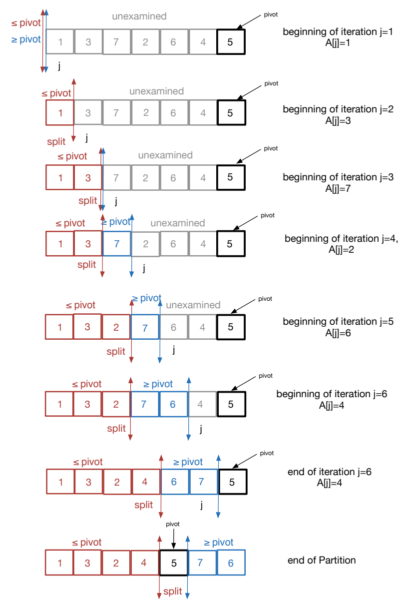
1. swap $A[j]$ with $A[split + 1]$, the first element of the second region: since $A[split + 1] > pivot$, it's "safe" to move it to the end of the second region (recall that the second region ends at position j after examining the j-th item);

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*, all items from* $A[left, j]$ *that appear in*

1. $A[left, split]$ *are less or equal to pivot*
2. $A[split + 1, j]$ *are greater than 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 $j+1$.

Thus 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