Data Structures & Algorithms

Lecture 3: Sorting

Many slides by Kevin Buchin

Recap

Recap: Analyzing Running Time

Two components:

1. Determine running time as function $T(n)$ of input size n
   - assume elementary operations take constant time
   - focus on worst-case running time

2. Characterize rate of growth of $T(n)$
   - focus on the order of growth
     ignore all but the most dominant terms
   - use O()-notation

     $O(g(n)) = \{ f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

     $T(n) = O(g(n))$: The order of growth of the running time T(n) is at most g(n).
   - also know $\Omega()$ for lower bounds, and $\Theta()$ for tight bounds

Recap: Recursive Algorithms

- recursive algorithms are based on reduction:

  solve a problem based on smaller instances

```
def binary_search(A, v, x=0, y=None):
    if (y == None): y = len(A)
    if (x<y):
        h = (x+y)//2
        if (A[h]<v): return binary_search(A, v, h+1, y)
        else: return binary_search(A, v, x, h)
    else:
        if (A[x] == v): return x
        else: return -1
```

```
binary_search([1,4,5], 4)
```

```
1
```

Recap: Algorithms

- A complete description of an algorithm consists of three parts:

1. the algorithm, *expressed in whatever way is clearest and most concise*
2. a proof of the algorithm’s correctness
   - For recursive algorithms: mathematical induction
   - Base case, induction hypothesis, induction step
   - Standard: $n \rightarrow n+1$ vs. strong induction: $m<n \rightarrow n$
3. a derivation of the algorithm’s running time
   - Find recurrence of running time
   - Solve recurrence (today)

Sorting

The sorting problem

Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: a permutation of the input such that $\langle a_{i1} \le \ldots \le a_{in} \rangle$

8	1	6	4	0	3	9	5

➡

0	1	3	4	5	6	8	9

- The input is typically stored in arrays
- Numbers ≈ Keys
- Additional information (satellite data) may be stored with keys
- We will study several solutions ≈ algorithms for this problem

Selection Sort

Probably the simplest sorting algorithm …

12	9	3	7	14	11

3	9	12	7	14	11

3	7	12	9	14	11

3	7	9	12	14	11

3	7	9	11	14	12

3	7	9	11	12	14

Selection-Sort(A, n)

Input: an array A and the number n of elements in A to sort
Output: the elements of A sorted into non-decreasing order

1. For i = 1 to n-1:
   - A. Set smallest to i
   - B. For j = i + 1 to n
     - i. If A[j] < A[smallest], then set smallest to j
   - C. Swap A[i] with A[smallest]

Selection Sort

Selection-Sort(A, n)

Input: an array A and the number n of elements in A to sort
Output: the elements of A sorted into non-decreasing order

1. For i = 1 to n-1:
   - A. Set smallest to i
   - B. For j = i + 1 to n
     - i. If A[j] < A[smallest], then set smallest to j
   - C. Swap A[i] with A[smallest]

Correctness? Loop Invariant Proof ✔

Running time? $O(n^2)$

Insertion Sort

- Like sorting a hand of playing cards:
  - start with empty left hand, cards on table
  - remove cards one by one, insert into correct position
  - to find position, compare to cards in hand from right to left
  - cards in hand are always sorted

Insertion Sort is
  - a good algorithm to sort a small number of elements
  - an incremental algorithm

Incremental algorithms

process the input elements one-by-one and maintain the solution for the elements processed so far.

Incremental algorithms

Incremental algorithms

process the input elements one-by-one and maintain the solution for the elements processed so far.

- In pseudocode:

IncAlg(A)

// incremental algorithm which computes the solution of a problem with input A = $\{x_1,\dots,x_n\}$

1. initialize: compute the solution for $\{x_1\}$
2. **for** j = 2 **to** n
3. **do** compute the solution for $\{x_1,\dots,x_j\}$ using the (already computed) solution for $\{x_1,\dots,x_{j-1}\}$

Insertion Sort

Insertion-Sort(A)

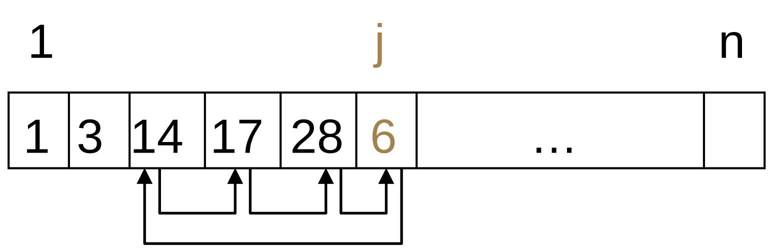
// incremental algorithm that sorts array A[1..n] in non-decreasing order

1. initialize: sort A[1]
2. **for** j = 2 **to** A.length
3. **do** sort A[1..j] using the fact that A[1.. j-1] is already sorted

Insertion Sort

```
Insertion-Sort(A)
 //   incremental algorithm that sorts array A[1..n] in non-decreasing
      order
      initialize: sort A[1]
      for j = 2 to A.length
          do key = A[j]
              i = j -1
              while i > 0  and A[i] > key
                  do  A[i+1] = A[i]
                      i = i -1
              A[i +1] = key
```

Insertion Sort is an in place algorithm: the numbers are rearranged within the array with only constant extra space.



Correctness

Correctness proof

Loop invariant

At the start of each iteration of the “outer” **for** loop (indexed by j) the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

Correctness proof

Insertion-Sort(A)

```
initialize: sort A[1]
for j = 2 to A.length
    do key = A[j]
        i = j -1
        while i > 0  and A[i] > key
            do  A[i+1] = A[i]
                i = i -1
        A[i +1] = key
```

Loop invariant

At the start of each iteration of the “outer” **for** loop (indexed by j) the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

Initialization

Just before the first iteration, j = 2 ➡ A[1..j-1] = A[1], which is the element originally in A[1], and it is trivially sorted.

Correctness proof

Insertion-Sort(A)

```
1.  initialize: sort A[1]
2.  for j = 2 to A.length
3.      do key = A[j]
4.         i = j -1
5.         while i > 0  and A[i] > key
6.             do  A[i+1] = A[i]
7.                 i = i -1
8.         A[i +1] = key
```

Loop invariant

At the start of each iteration of the "outer" **for** loop (indexed by j) the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

Maintenance

Strictly speaking need to prove loop invariant for "inner" **while** loop. Instead, note that body of **while** loop moves A[j-1], A[j-2], A[j-3], and so on, by one position to the right until proper position of key is found (which has value of A[j]) ➡ invariant maintained.

Correctness proof

Insertion-Sort(A)

```
initialize: sort A[1]
for j = 2 to A.length
    do key = A[j]
        i = j -1
        while i > 0  and A[i] > key
            do  A[i+1] = A[i]
                i = i -1
        A[i +1] = key
```

Loop invariant

At the start of each iteration of the "outer" **for** loop (indexed by j) the subarray A[1..j-1] consists of the elements originally in A[1..j-1] but in sorted order.

Termination

The outer **for** loop ends when j > n; this is when j = n+1 $\Rightarrow$ j-1 = n. Plug n for j-1 in the loop invariant $\Rightarrow$ the subarray A[1..n] consists of the elements originally in A[1..n] in sorted order.

Another sorting algorithm

using a different paradigm …

Merge Sort

- A divide-and-conquer sorting algorithm.

Divide
: the problem into a number of subproblems that are smaller instances of the same problem.

Conquer
: the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.

Combine
: the solutions to the subproblem into the solution for the original problem.

Divide-and-conquer

D&CAlg(A)

// divide-and-conquer algorithm that computes the solution of a problem with input $A = \{x_1,\ldots,x_n\}$

1. **if** # elements of A is small enough (for example 1)
2. **then** compute Sol (the solution for A) brute-force
3. **else**
4. split A in, for example, 2 non-empty subsets A_1 and A_2
5. Sol_1 = D&CAlg(A_1)
6. Sol_2 = D&CAlg(A_2)
7. compute Sol (the solution for A) from Sol_1 and Sol_2
8. **return** Sol

Merge Sort

Merge-Sort(A)

// divide-and-conquer algorithm that sorts array A[1..n]

1. **if** A.length = 1
2. **then** compute Sol (the solution for A) brute-force
3. **else**
4. split A in 2 non-empty subsets A_1 and A_2
5. Sol_1 = Merge-Sort(A_1)
6. Sol_2 = Merge-Sort(A_2)
7. compute Sol (the solution for A) from Sol_1 en Sol_2

Merge Sort

Merge-Sort(A)

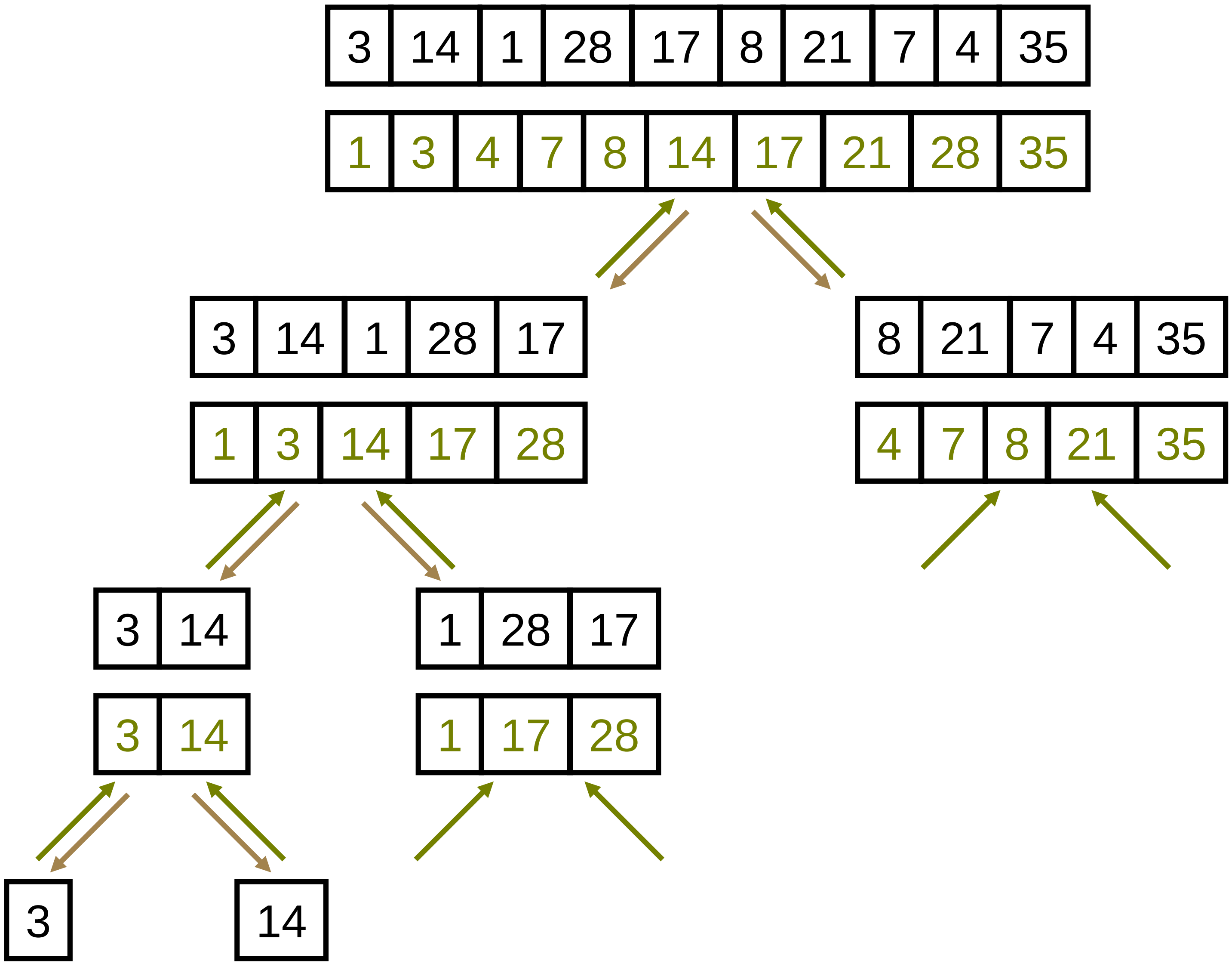
// divide-and-conquer algorithm that sorts array A[1..n]

1. **if** A.length == 1
2. **then skip**
3. **else**
4. n = A.length ; $n_1 = \lfloor n/2 \rfloor$; $n_2 = \lceil n/2 \rceil$;

   copy A[1.. n_1] to auxiliary array A_1[1.. n_1]

   copy A[n_1+1..n] to auxiliary array A_2[1.. n_2]
5. Merge-Sort(A_1)
6. Merge-Sort(A_2)
7. Merge(A, A_1, A_2)

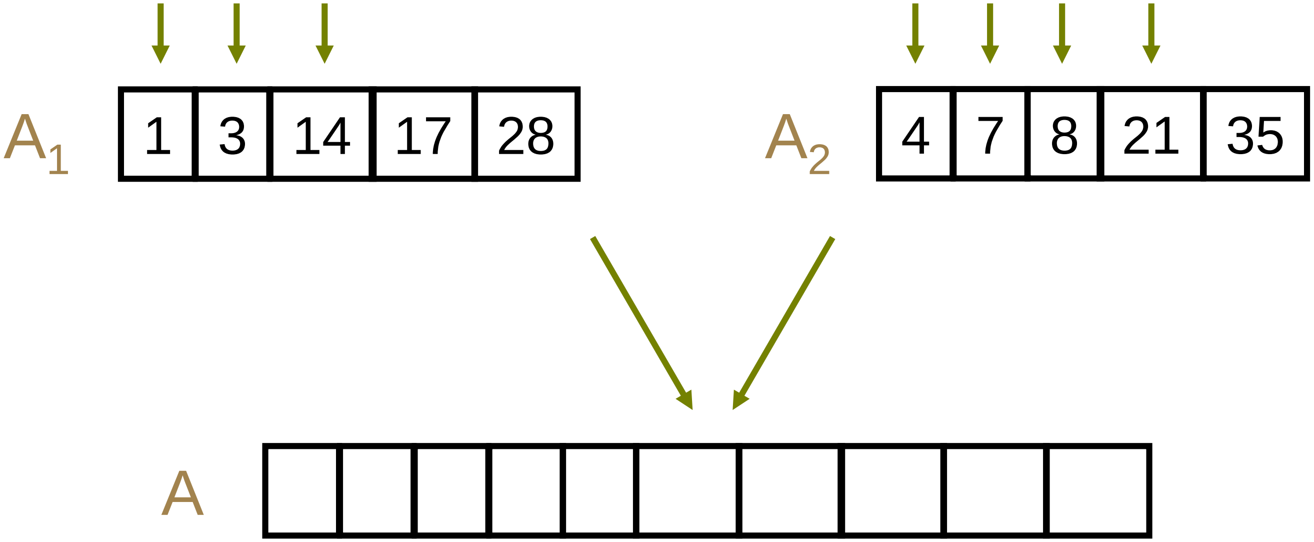
Merge Sort



3 14 1 28 17 8 21 7 4 35
1 3 4 7 8 14 17 21 28 35
3 14 1 28 17
8 21 7 4 35
1 3 14 17 28
4 7 8 21 35
3 14
1 28 17
3 14
1 17 28
3
14

Merge Sort

- Merging



Correctness?

Efficiency

Analysis of Insertion Sort

```
Insertion-Sort(A)
    initialize: sort A[1]
    for j = 2 to A.length
        do key = A[j]
            i = j -1
            while i > 0  and A[i] > key
                do  A[i+1] = A[i]
                    i = i -1
            A[i +1] = key
```

- Get as tight a bound as possible on the worst case running time.
  - ➡ lower and upper bound for worst case running time

Upper bound: Analyze worst case number of elementary operations

Lower bound: Give “bad” input example

Analysis of Insertion Sort

```
Insertion-Sort(A)
initialize: sort A[1]                          O(1)
for j = 2 to A.length
    do key = A[j]                              } O(1)
       i = j -1                                }
       while i > 0  and A[i] > key             } worst case:
           do  A[i+1] = A[i]                   } (j-1) · O(1)
               i = i -1                        }
       A[i +1] = key                           O(1)
```

Upper bound: Let $T(n)$ be the worst case running time of InsertionSort on an array of length n. We have

$$T(n) = O(1) + \sum_{j=2}^{n} \{ O(1) + (j-1)\cdot O(1) + O(1) \} = \sum_{j=2}^{n} O(j) = O(n^2)$$

Lower bound: Array sorted in de-creasing order ➡ $\Omega(n^2)$

The worst case running time of InsertionSort is $\Theta(n^2)$.

Analysis of Merge Sort

Merge-Sort(A)
// divide-and-conquer algorithm that sorts array A[1..n]

if A.length = 1 — O(1)
  then skip
  else
    n = A.length ; n_1 = floor(n/2); n_2 = ceil(n/2); — O(1)
    copy A[1.. n_1] to auxiliary array A_1[1.. n_1] — O(n)
    copy A[n_1+1..n] to auxiliary array A_2[1.. n_2] — O(n)
    Merge-Sort(A_1); Merge-Sort(A_2) — ??
    Merge(A, A_1, A_2) — O(n)

$T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)$

MergeSort is a recursive algorithm
➡ running time analysis leads to recursion

Analysis of Merge Sort

- Let $T(n)$ be the worst case running time of MergeSort on an array of length n. We have

$$T(n) = \begin{cases} \boxed{O(1) \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \text{if } n = 1} \\ \boxed{T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil)} + \Theta(n) \quad \text{if } n > 1 \end{cases}$$

often written as $2T(n/2)$

frequently omitted since it (nearly) always holds

➡ $T(n) = 2\,T(n/2) + \Theta(n)$

Master theorem ➡ $T(n) = \Theta(n \log n)$

Tips

- Analysis of recursive algorithms:
  find the recursion and solve

- Analysis of loops: summations

- Some standard recurrences and sums:
  - $T(n) = 2T(n/2) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$
  - $\sum_{i=1}^{n} i = \frac{1}{2} n(n+1) = \Theta(n^2)$
  - $\sum_{i=1}^{n} i^2 = \Theta(n^3)$

Rate of growth

		n=10	n=100	n=1000
Insertion Sort:	$15 n^2 + 7n - 2$	1568	150698	1.5×10^7
Merge Sort:	$300 n \lg n + 50 n$	10466	204316	3.0×10^6

Insertion Sort
6 x faster

Insertion Sort
1.35 x faster

Merge Sort
5 x faster

The rate of growth of the running time as a function of the input is essential!

n = 1,000,000

Insertion Sort	1.5×10^{13}	
Merge Sort	6×10^9	2500 x faster !

Sorting algorithms

- We focus on running time
- Storage?
  - all sorting algorithms discussed use O(n) storage
  - in place: only constant amount of extra storage

	worst case running time	in place
Selection Sort	$\Theta(n^2)$	yes
Insertion Sort	$\Theta(n^2)$	yes
Merge Sort	$\Theta(n \log n)$	no

  - in place & O(n log n)? … later in the course

QuickSort

another divide-and-conquer sorting algorithm…

QuickSort

- QuickSort is a divide-and-conquer algorithm

To sort the subarray A[p..r]:

Divide

Partition A[p..r] into two subarrays A[p..q-1] and A[q+1..r], such that each element in A[p..q-1] is ≤ A[q] and A[q] is < each element in A[q+1..r].

Conquer

Sort the two subarrays by recursive calls to QuickSort

Combine

No work is needed to combine the subarrays, since they are sorted in place.

- Divide using a procedure Partition which returns q.

QuickSort

```
QuickSort(A, p, r)
if p < r
   then q = Partition(A, p, r)
        QuickSort(A, p, q-1)
        QuickSort(A, q+1, r)
```

- Initial call: QuickSort(A, 1, n)

```
Partition(A, p, r)
x = A[r]
i = p-1
for j = p to r-1
   do if A[j] ≤ x
        then i = i+1
             exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

- Partition always selects A[r] as the pivot (the element around which to partition)

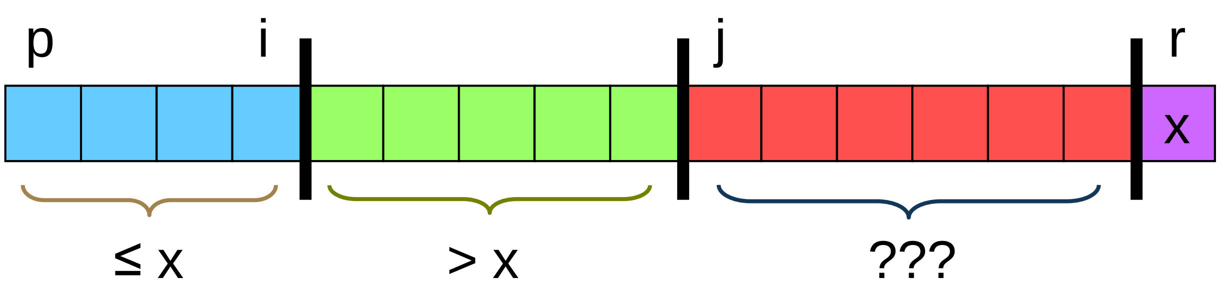
Partition

- As Partition executes, the array is partitioned into four regions (some may be empty)

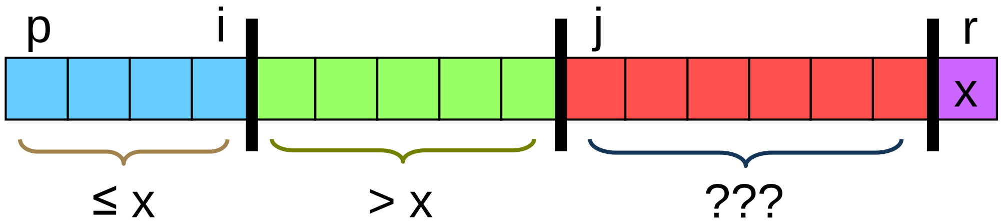
```
Partition(A, p, r)
x = A[r]
i = p-1
  for j = p to r-1
    do if A[j] ≤ x
          then i = i+1
               exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

Loop invariant

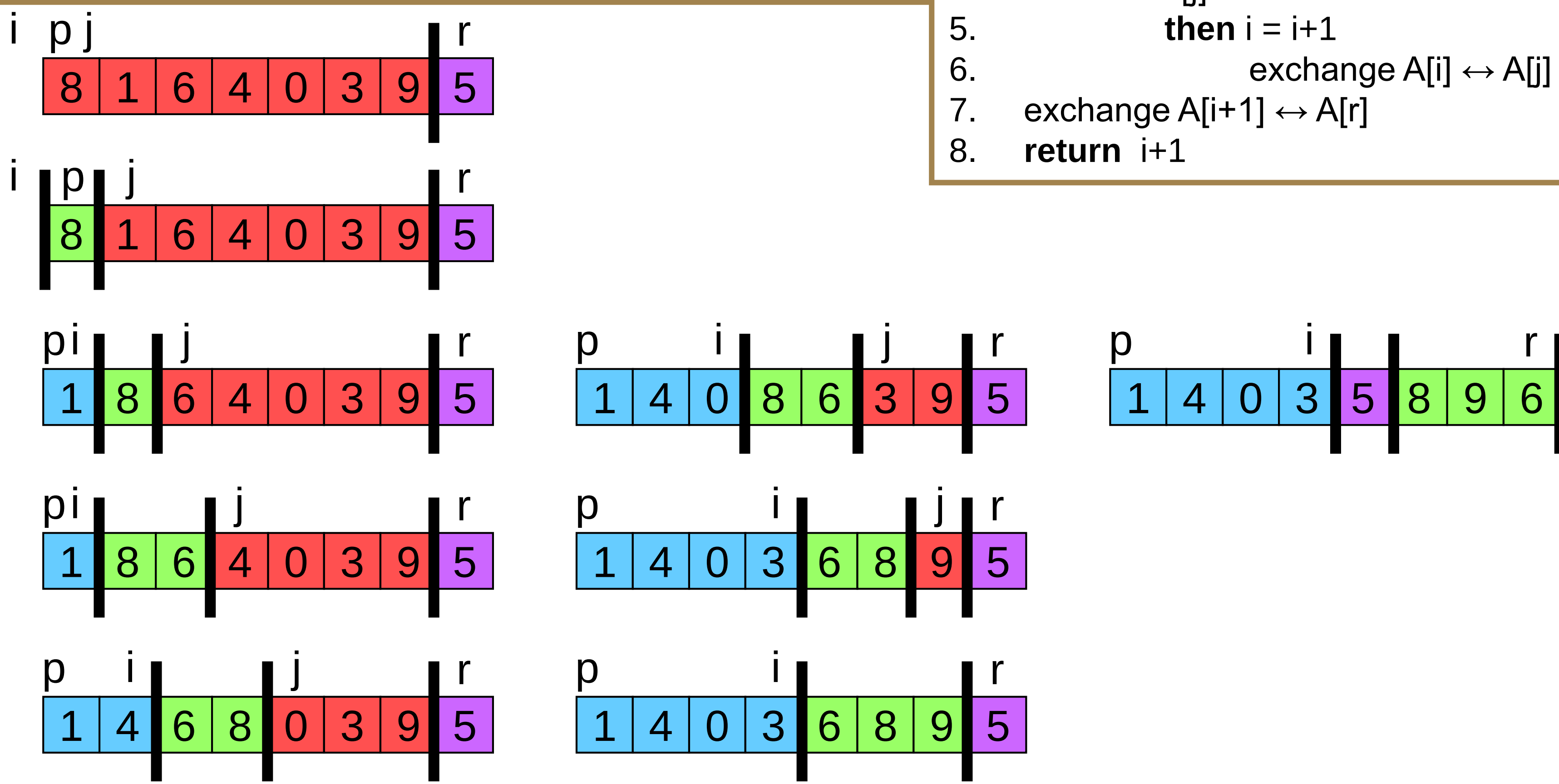
1. all entries in A[p..i] are ≤ pivot
2. all entries in A[i+1..j-1] are > pivot
3. A[r] = pivot



Partition



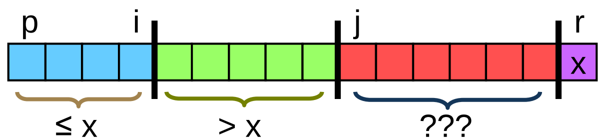

```
Partition(A, p, r)
x = A[r]
i = p-1
  for j = p to r-1
    do if A[j] ≤ x
          then i = i+1
                exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```



Partition - Correctness

Loop invariant

1. all entries in A[p..i] are ≤ pivot
2. all entries in A[i+1..j-1] are > pivot
3. A[r] = pivot



Partition(A, p, r)

```
x = A[r]
i = p-1
   for j = p to r-1
      do if A[j] ≤ x
              then i = i+1
                      exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return  i+1
```

Initialization

before the loop starts, all conditions are satisfied, since r is the pivot and the two subarrays A[p..i] and A[i+1..j-1] are empty

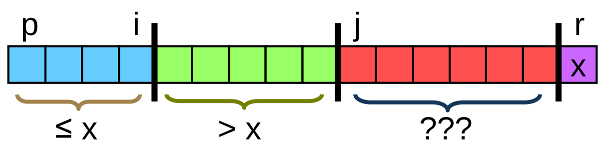
Maintenance

while the loop is running, if A[j] ≤ pivot, then A[j] and A[i+1] are swapped and then i and j are incremented ➡ 1. and 2. hold.
If A[j] > pivot, then increment only j ➡ 1. and 2. hold.

Partition - Correctness

Loop invariant

1. all entries in A[p..i] are ≤ pivot
2. all entries in A[i+1..j-1] are > pivot
3. A[r] = pivot




```
Partition(A, p, r)
1.  x = A[r]
2.  i = p-1
3.    for j = p to r-1
4.      do if A[j] ≤ x
5.           then i = i+1
6.                exchange A[i] ↔ A[j]
7.  exchange A[i+1] ↔ A[r]
8.  return  i+1
```

Termination

when the loop terminates, j = r, so all elements in A are partitioned into one of three cases:

A[p..i] ≤ pivot, A[i+1..r-1] > pivot, and A[r] = pivot

- Lines 7 and 8 move the pivot between the two subarrays

Running time: Θ(n) for an n-element subarray

Running Time of Partition and Quicksort

- Partition takes $O(n)$ time
- Quicksort takes $O(n^2)$ time in the worst case
- Picking a random pivot results in reasonably balanced split on average
  - ➡ Randomized Quicksort takes $O(n \log n)$ expected time
- Quicksort is fast in practice
- The idea of Partition can be used to find the median of an unsorted sequence of numbers in $O(n)$ time

Recap and preview

Today

- Sorting Algorithms
- Incremental Algorithms, Divide & Conquer Algorithms

Next lecture

- Does sorting take $\Theta(n \log n)$ time?