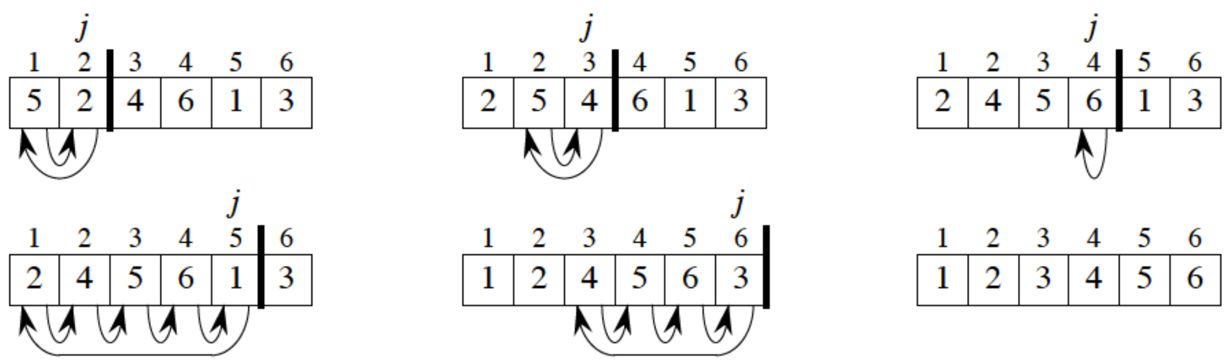
sorting

preliminaries

- input is an array of *n* elements
- sorting key is integer: sorting in the increasing order of the keys
- internal sorting: all elements are stored in main memory
- external sorting: elements are stored on disk or tape
- comparison-based sorting: comparison ($<$ or $>$) is the only operation applied

insertionsort (revisited, Lecture1)

- N-1 passes
- For pass p, move the element in position p left until its correct place is found
- Running time: $O(N^2)$



- All elements on the left of p are sorted
- Best case: array already sorted
- Fast for almost sorted inputs

```
INSERTION-SORT(A, n)
  for j = 2 to n
      key = A[j]
      // Insert A[j] into the sorted sequence A[1 .. j - 1]
      i = j - 1
      while i > 0 and A[i] > key
          A[i + 1] = A[i]
          i = i - 1
      A[i + 1] = key
```

selectionsort (revisited, Lecture2)

First find the smallest element of the array A and exchange it with the element in A[1]. Then find the second smallest element of A and exchange it with A[2]. Continue in the same fashion for the first N-1 elements of A. Cost: $O(N^2)$

j = 1

$\boxed{1}\ 5\ 6\ 3\ 7\ 2$ smallest = 1

j = 2

$1\ \boxed{5}\ 6\ 3\ 7\ 2$ smallest = 2
smallest = 4
smallest = 6

j = 3

$1\ 2\ \boxed{6}\ 3\ 7\ 5$ smallest = 3
smallest = 4

j = 4

$1\ 2\ 3\ \boxed{6}\ 7\ 5$ smallest = 4
smallest = 6

j = 5

$1\ 2\ 3\ 5\ \boxed{7}\ 6$ smallest = 5
smallest = 6

$1\ 2\ 3\ 5\ 6\ 7$

```
SELECTION-SORT(A)
  n = A.length
  for j = 1 to n - 1
      smallest = j
      for i = j + 1 to n
          if A[i] < A[smallest]
              smallest = i
      exchange A[j] with A[smallest]
```

shellsort

- ShellSort was the first algorithm with average running time less than $O(N^2)$
- ShellSort uses a pre-defined **increment sequence**, $h_1, h_2, \ldots, h_t$, (h_1 must be =1)
- After a phase (increment h_k), for every i: $A[i] \le A[i+h_k]$ => all elements spaced by h_k apart are sorted

  and sort sub-arrays A[i + hk] (k = t, t - 1, ..., 1) sequentially (insertionsort)
- shellsort property: *hk-sorted* list is also *hj-sorted* for $j > k$.
- The performance of the algorithm depends on the increment sequence.

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

$h=5$														
		81	94	11	96	12	35	17	95	28	58	41	75	15
	$i=1$	35	94	11	96	12	81	17	95	28	58	41	75	15
	$i=2$	35	17	11	96	12	81	94	95	28	58	41	75	15
	$i=3$	35	17	11	96	12	81	94	95	28	58	41	75	15
	$i=4$	35	17	11	28	12	81	94	95	96	58	41	75	15
	$i=5$	35	17	11	28	12	81	94	95	96	58	41	75	15
	$i=6$	35	17	11	28	12	41	94	95	96	58	81	75	15
	$i=7$	35	17	11	28	12	41	75	95	96	58	81	94	15
	$i=8$	35	17	11	28	12	41	75	15	96	58	81	94	95

shellsort (cnt'd)

- Complexity: $O(N^r)$, with $1 < r < 2$, i.e., better than quadratic
- It depends on the chosen increment sequence (how "evenly" insertionsort is performed in every phase)
- A pass with increment h_k consists of insertionsort of about N/h_k elements
- insertionsort is $O(N^2)$
- therefore a phase costs $O((N/h_k)^2) = O(N^2/h_k)$
- summing over all phases: $O(N^2 \Sigma_k 1/h_k)$
- because $\max\{1/h_k\} = 1$ ($h_1=1$)
- Worst case: $O(N^2)$

heapsort

- Using priority queue to sort in O(NlogN) time:
  1) Build a priority queue from the input array
  2) deleteMin *N* times, generating a sequence in sorted order.

- #2 implies that we use a second array to store what we delete from the heap
- To avoid using an extra array, the return value of deleteMin can be put back into the last place of the heap (emptied by deleteMin).
- This produces a maxheap (decreasing order)
- What if instead of building the (min) heap, we build the (max) heap?

```
BUILD-MAX-HEAP(A, n)
  for i = ⌊n/2⌋ downto 1
      MAX-HEAPIFY(A, i, n)
```

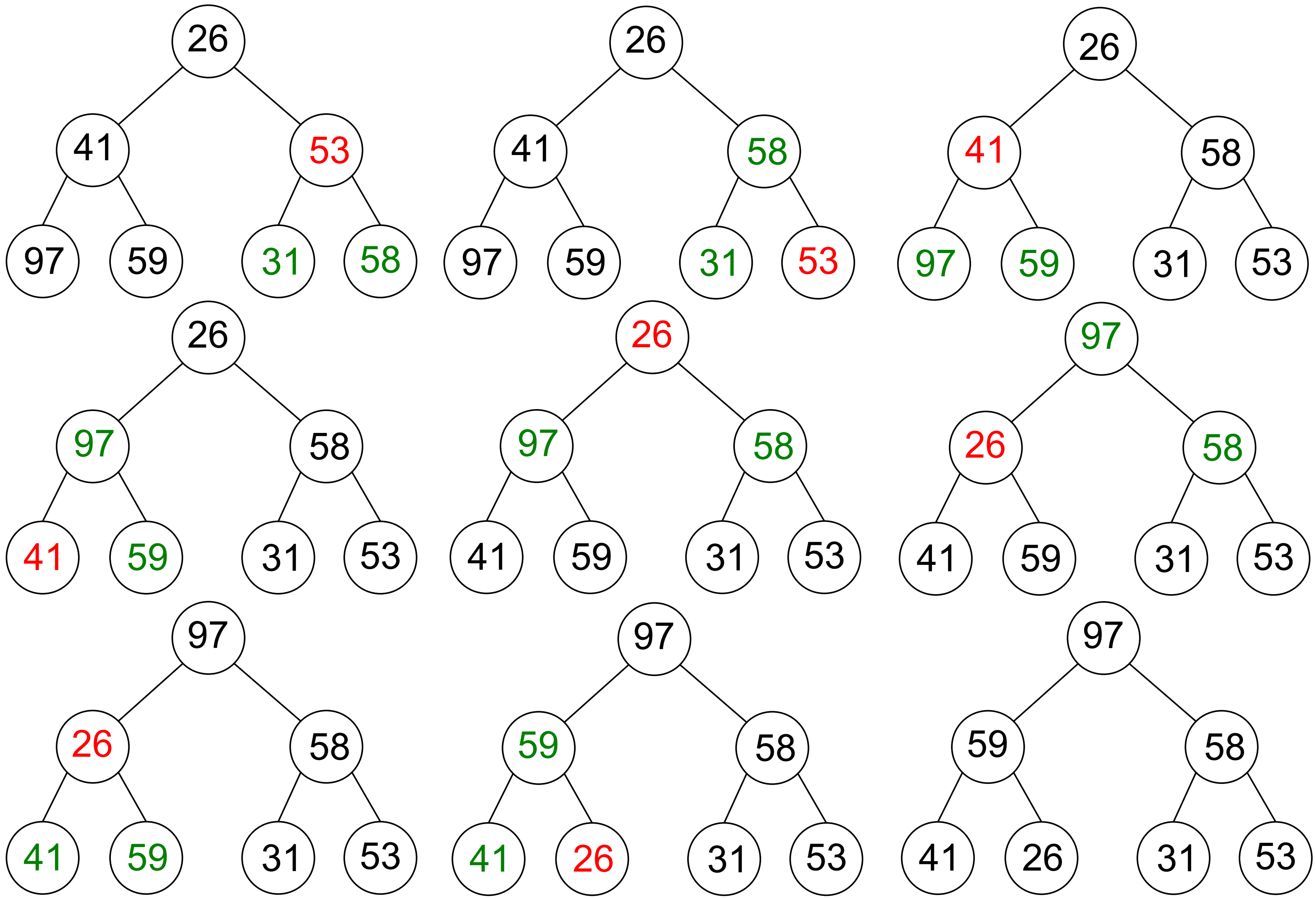
```
MAX-HEAPIFY(A, i, n)
  l = LEFT(i)
  r = RIGHT(i)
  if l ≤ n and A[l] > A[i]
      largest = l
  else largest = i
  if r ≤ n and A[r] > A[largest]
      largest = r
  if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest, n)
```

example

26 41 53 97 59 31 58

Build maxheap:

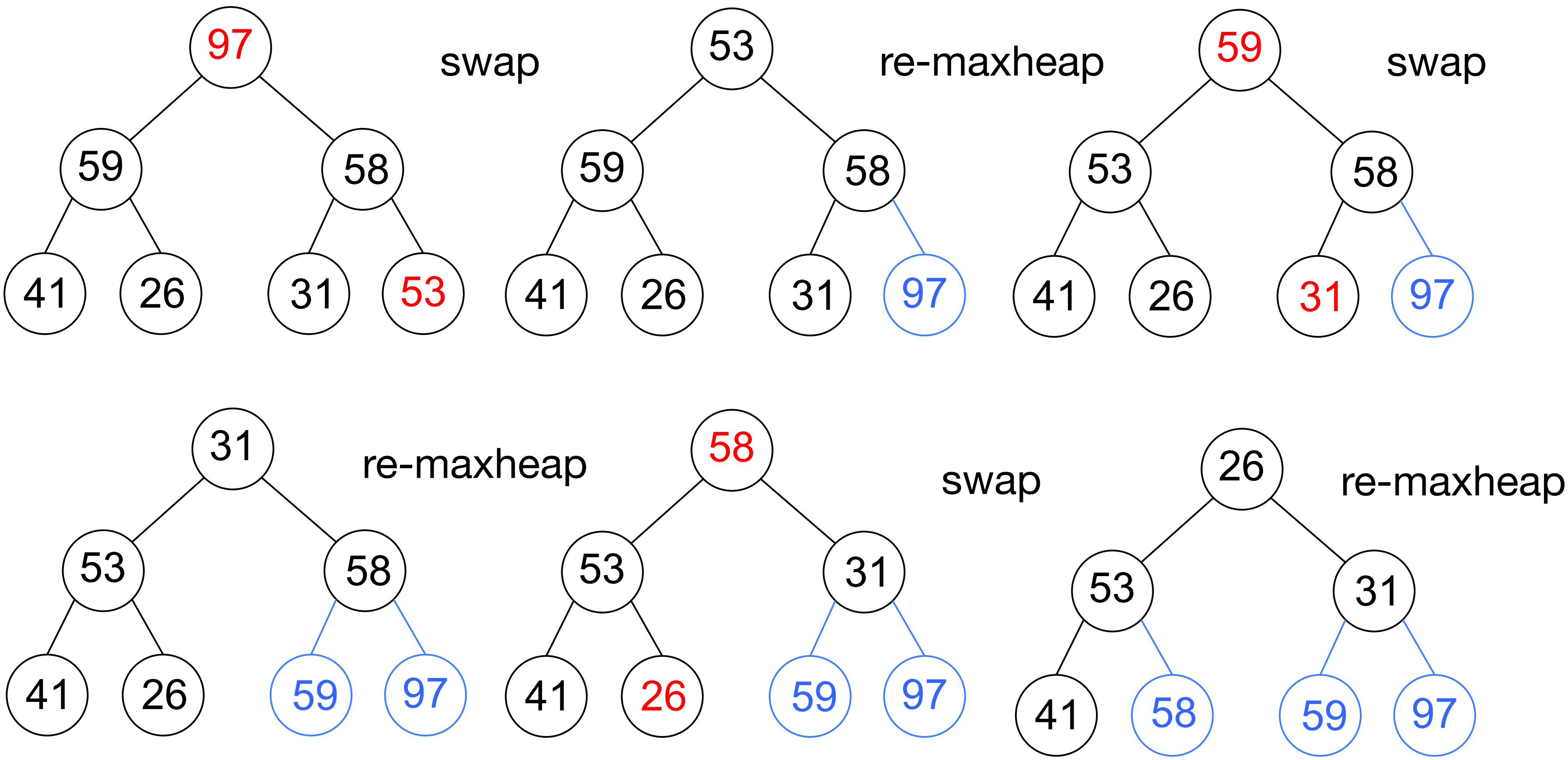
1. Arbitrary order
2. Compare with children and percolateDOWN



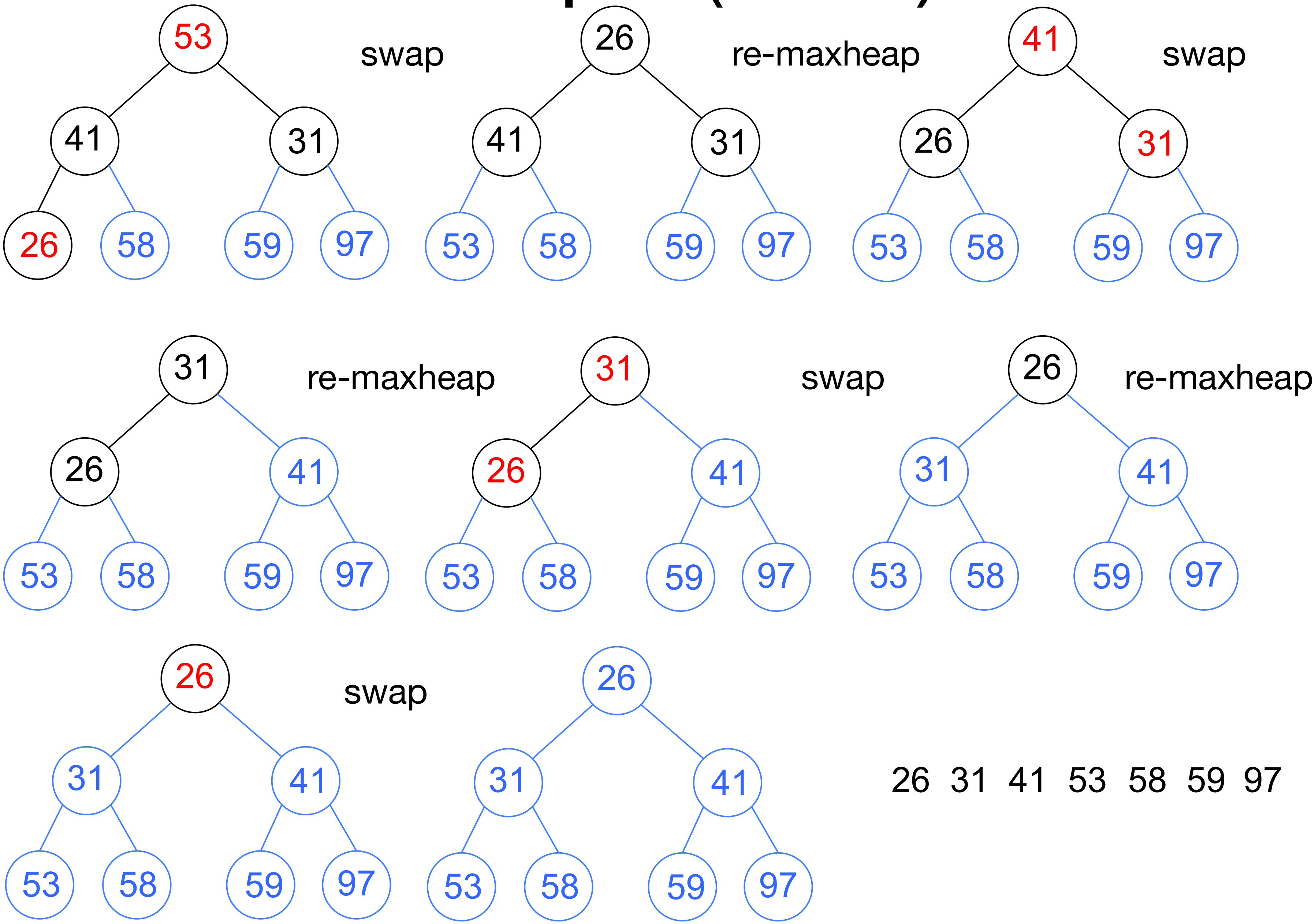
$\textsc{Heapsort}(A, n)$
  $\textsc{Build-Max-Heap}(A, n)$
  for $i = n$ **downto** 2
    exchange $A[1]$ with $A[i]$
    $\textsc{Max-Heapify}(A, 1, i - 1)$

example (cnt'd)

swap first and last, then ignore last, maxheapify, repeat



example (cnt'd)



Mergesort divide-and-conquer

divide and conquer *(Philip II, King of Macedonia, 382-336BC)*

an algorithm design with the following steps:

divide the problems into a number of subproblems that are smaller instances of the same problem

conquer the subproblems by solving them recursively (for small subproblems, a brute-force method can also be used)

combine the solutions of the subproblems into the solution of the original problem

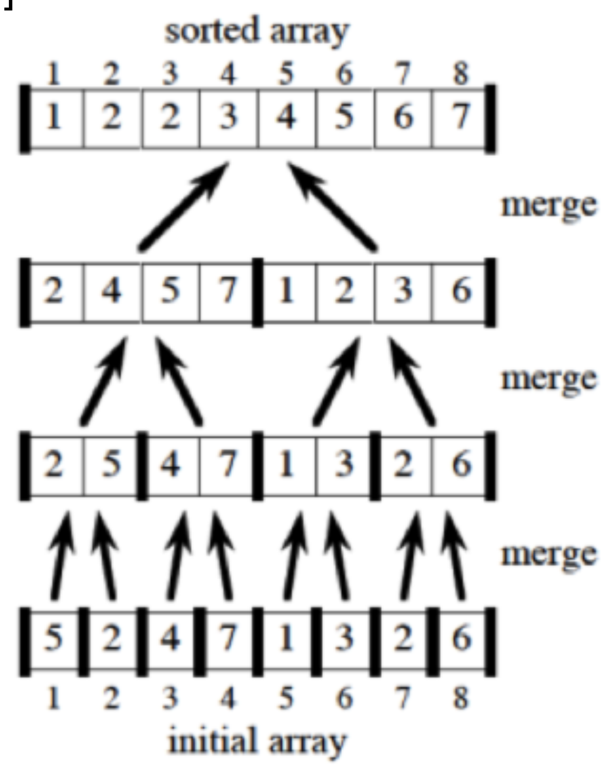
merge sort

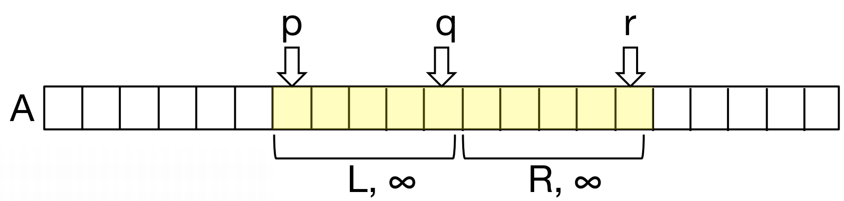
- Divide n-element sequence into two subsequences of n/2 elements each
- Sort the subsequences recursively using merge sort
- Merge the two sorted subsequences to produce the final solution

in a nutshell

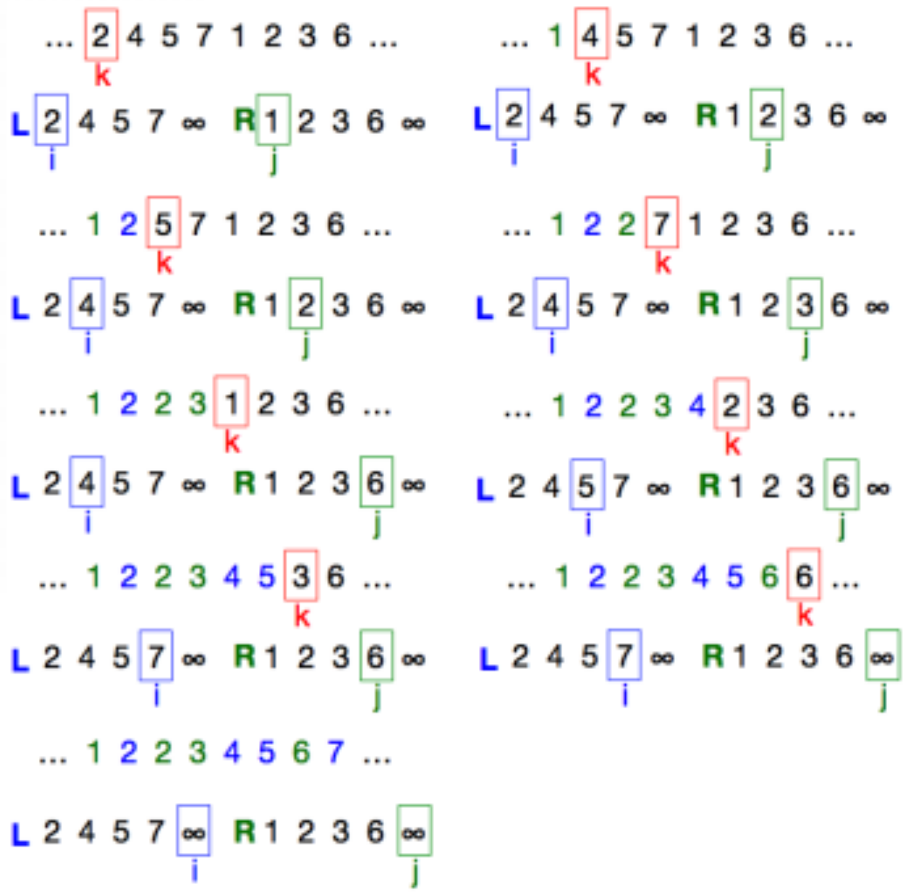
- Consider an array A[1…N] and an instance (subarray) A[p…r] where sorting will be solved
- Divide A[p…r] into A[p…q] and A[q+1…r] where q = (r+p)/2
- Conquer by sorting A[p…q] and A[q+1…r] separately
- Merge the sorted subarrays in A[p…r]

- So…merging does the sorting
- O(NlogN)



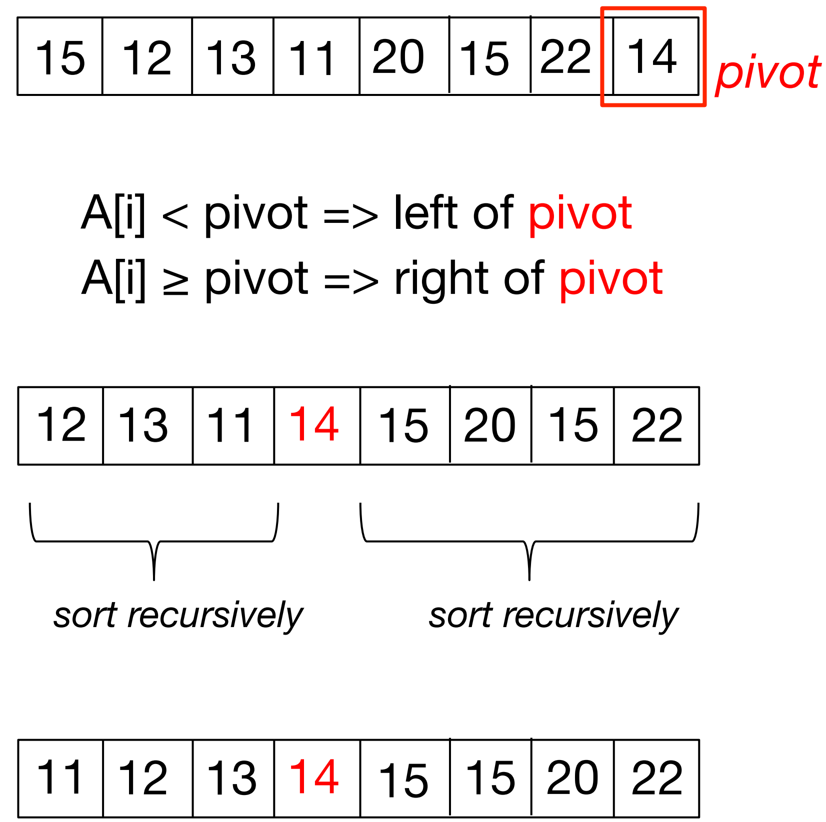



```
MERGE(A, p, q, r)
  n1 = q - p + 1
  n2 = r - q
  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
  for i = 1 to n1
      L[i] = A[p + i - 1]
  for j = 1 to n2
      R[j] = A[q + j]
  L[n1 + 1] = ∞
  R[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r
      if L[i] ≤ R[j]
          A[k] = L[i]
          i = i + 1
      else A[k] = R[j]
          j = j + 1
```



Quicksort divide-and-conquer

- A divide-and-conquer algorithm
- worst-case running time: $O(N^2)$
- average-case running time: $O(N \log N)$
- In practice, quicksort is the fastest sorting algorithm for large input arrays
- For small arrays, insertionsort is better
- So, in the recursion process, when the resulting subarrays are small, use insertionsort



the idea

- **Divide:**

  Partition (rearrange) the array A[p…r] into two subarrays A[p…q -1] and A[q+1…r] such that each element of A[p…q -1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1…r]. Compute the index q as part of this partitioning procedure.

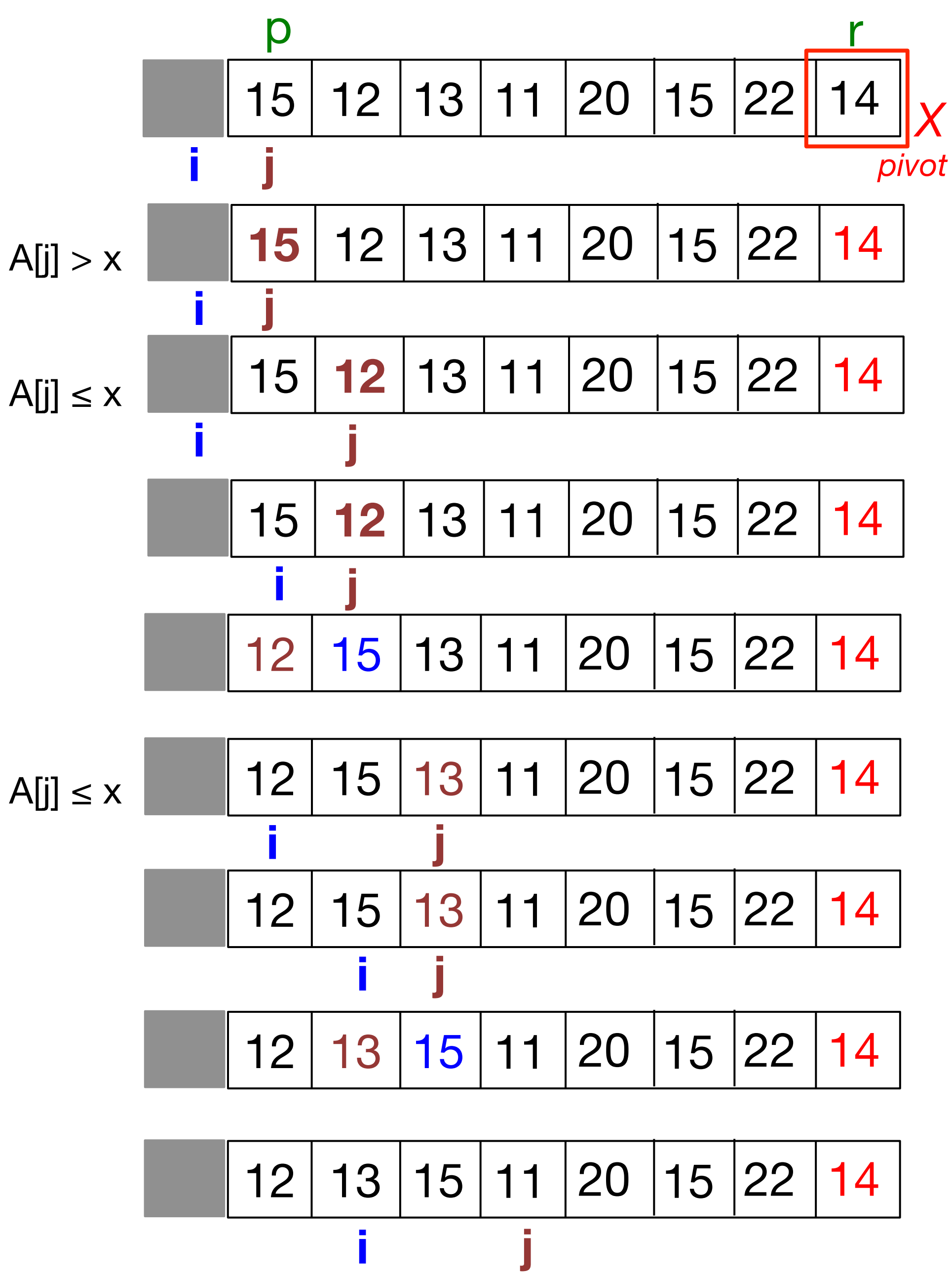
- **Conquer:**

  Sort the two subarrays A[p…q-1] and A[q+1…r] by recursive calls to quicksort.

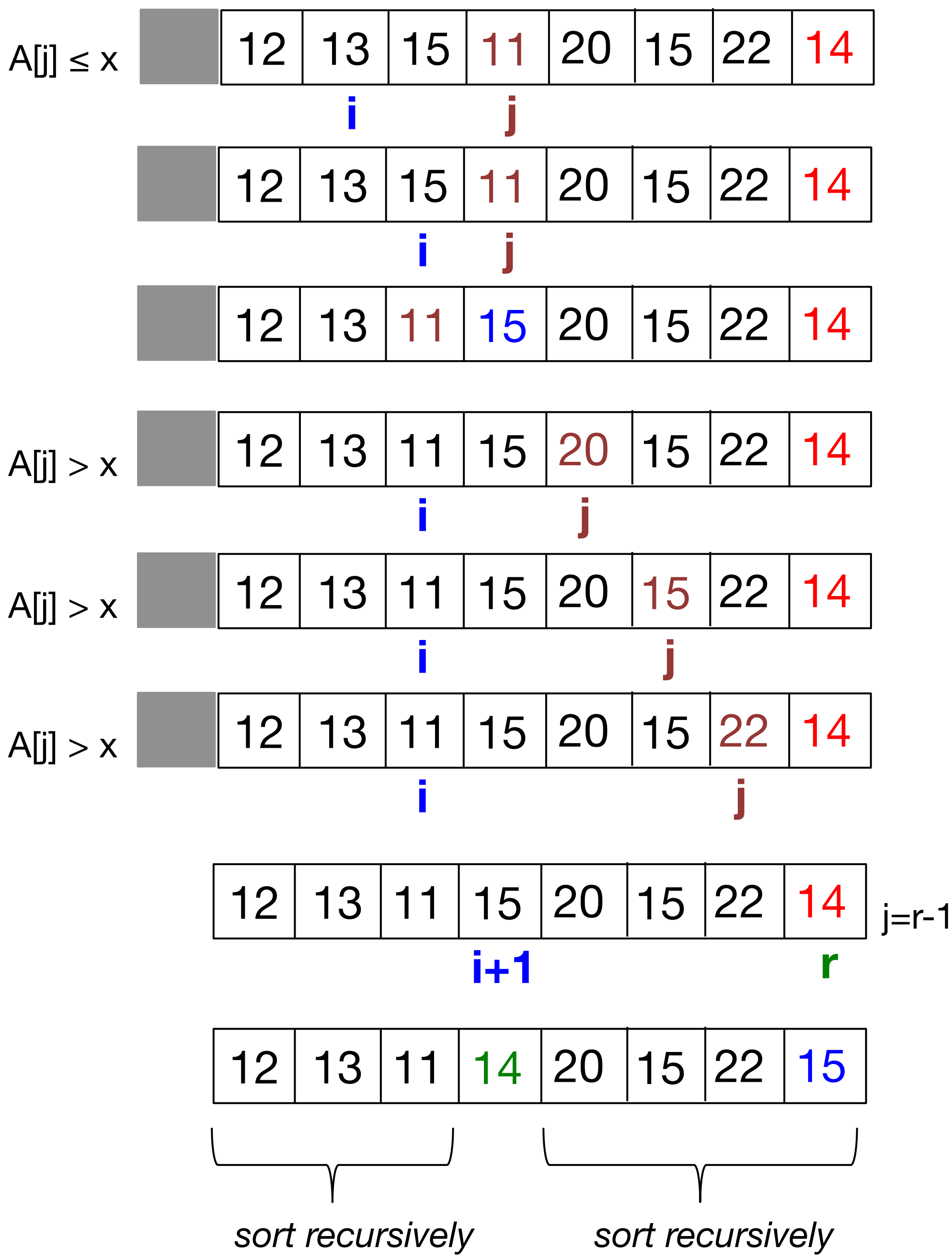
pivot

- Pick the *1st* element: can lead to $O(N^2)$ worst case for pre-sorted arrays
- Pick at a random position: a generally good and safe pick, but need to use a random generator.
- Pick the median of the three elements at *0*, *N-1* and $\lceil N/2 \rceil$ positions: a good choice in general.

find the pivot position




```
PARTITION(A, p, r)
    x = A[r]
    i = p − 1
    for j = p to r − 1
        if A[j] ≤ x
            i = i + 1
            exchange A[i] with A[j]
    exchange A[i + 1] with A[r]
    return i + 1
```

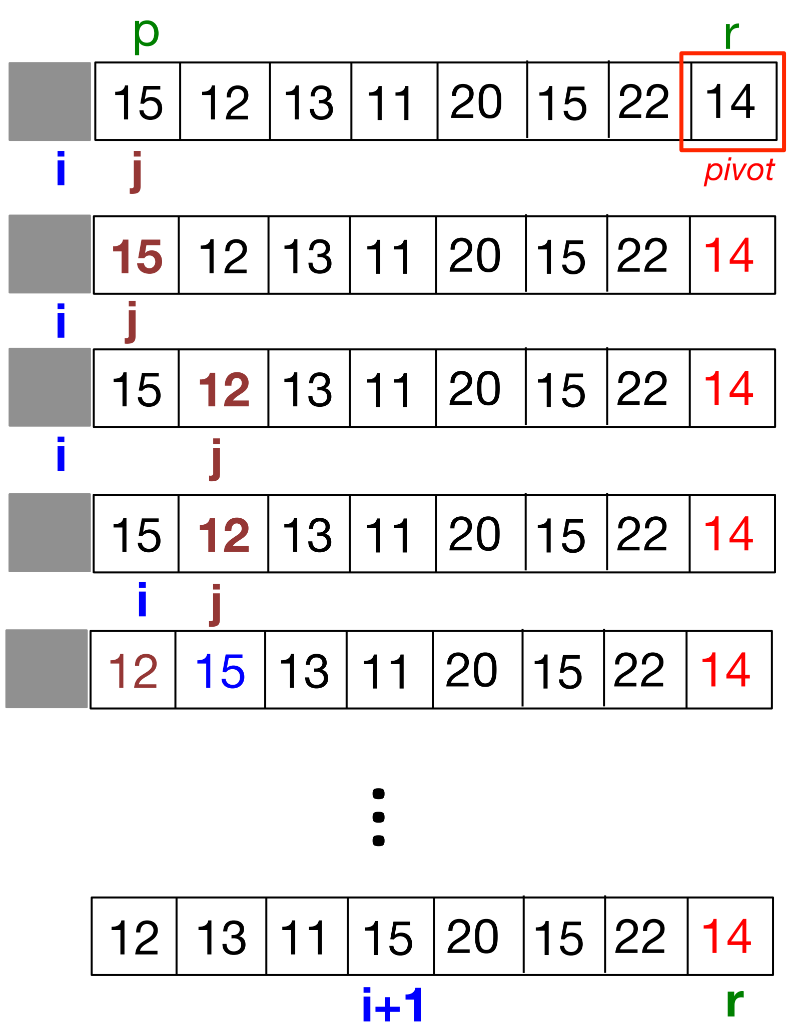


PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \le x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$

QUICKSORT(A, p, r)

if $p < r$
  $q =$ PARTITION(A, p, r)
  QUICKSORT$(A, p, q - 1)$
  QUICKSORT$(A, q + 1, r)$




```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

correctness

Loop invariant:

(i) all entries in A[p…i] ≤ pivot; (ii) all entries in A[i+1…j-1] > pivot; (iii)A[r] = pivot

Initialization:

Before the loop starts, all the conditions of the loop invariant are satisfied, because r is the pivot and the subarrays A[p…i] and A[i+1...j-1] are empty.

Maintenance:

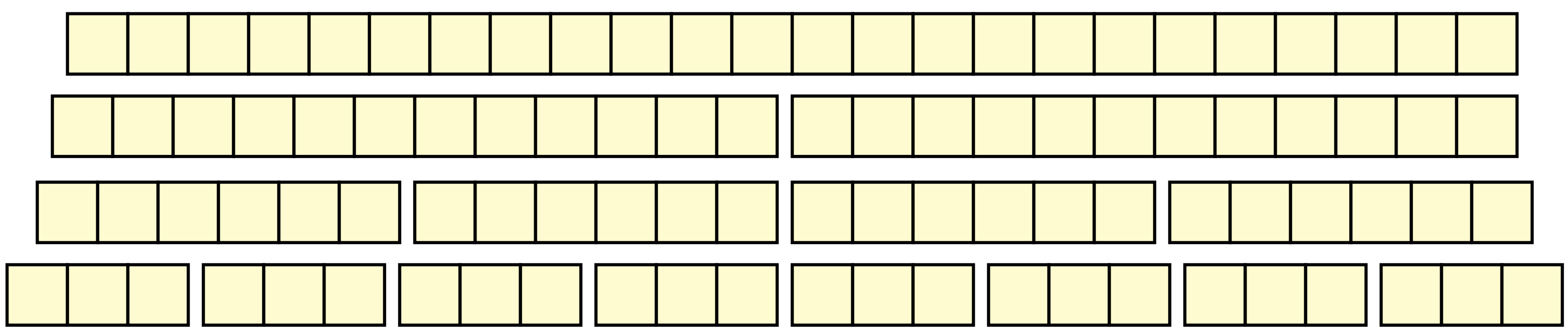
While the loop is running: If A[j] > pivot, then increment only j; if A[j] ≤ pivot, then A[j] and A[i+1] are swapped and i and j are incremented.

Termination:

When the loop terminates, j=r, so all elements in A are partitioned into one of the three cases: A[p…i] ≤ pivot, A[i+1…r-1] > pivot, and A[r] = pivot.

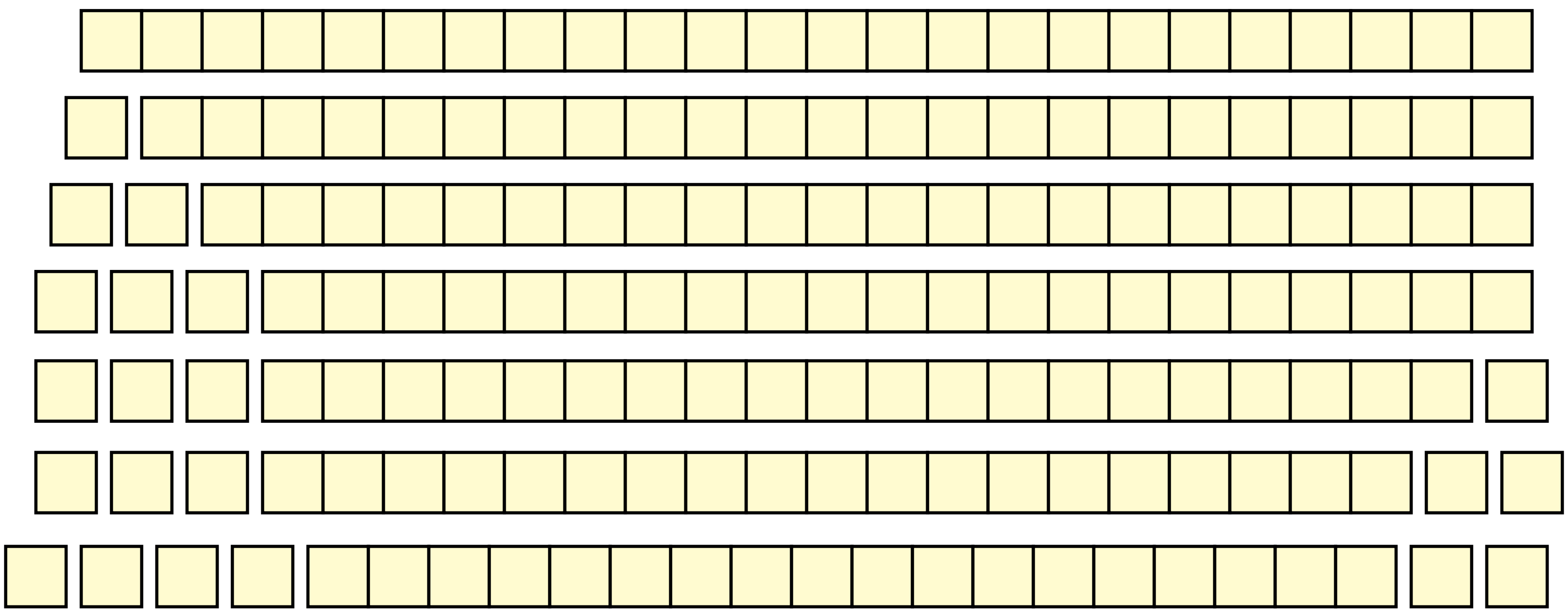
average case

- We cut the array size in half each time
- So the depth of the recursion in logN
- At each level of the recursion, all the partitions at that level do work that is linear in N
- $O(\log_2 N) * O(N) = O(N\log N)$
- Hence in the average case, quicksort has time complexity $O(N\log N)$
- What about the worst case?



Worst case

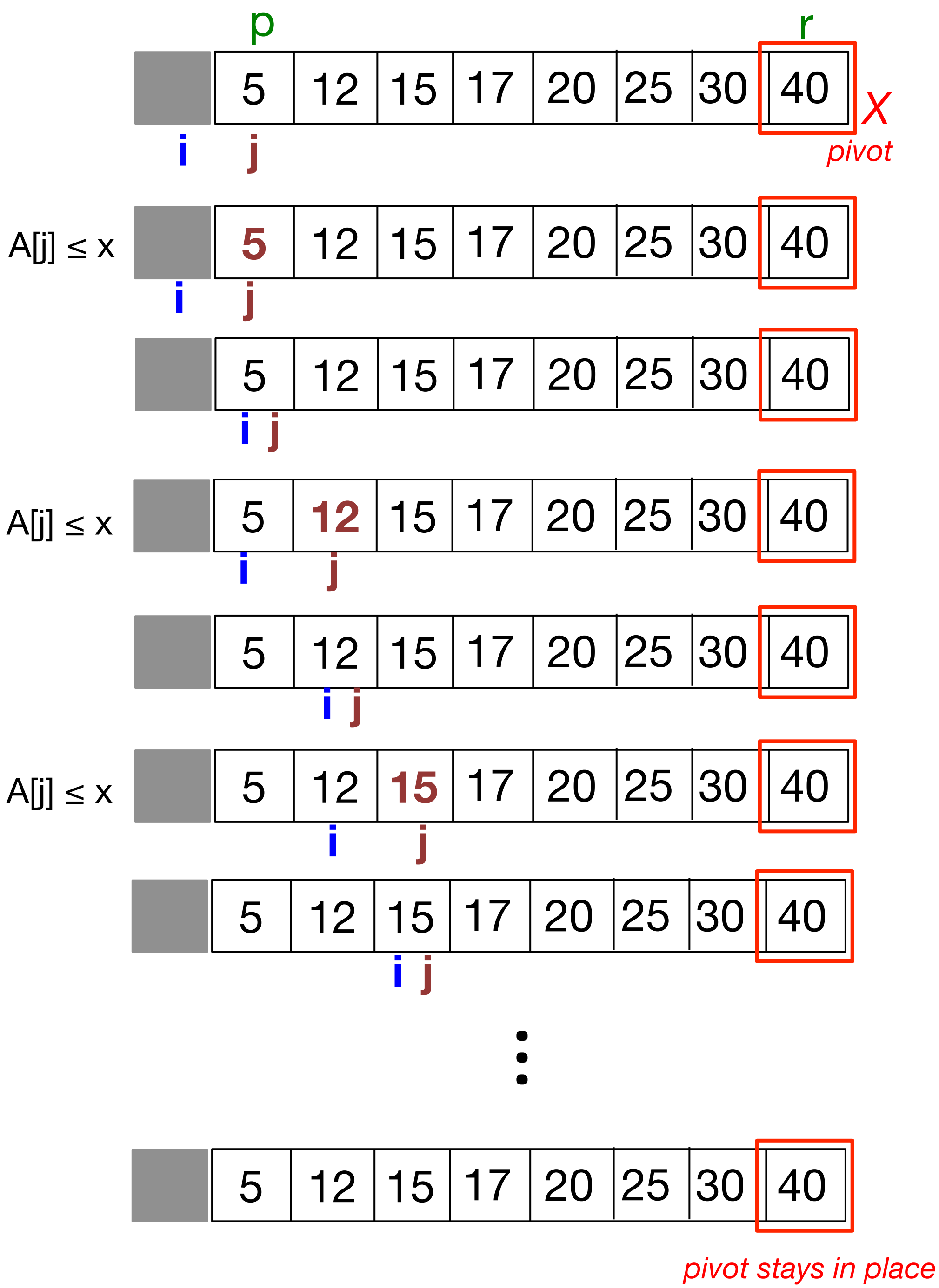
- In the worst case, partitioning always divides the size `N` array into these three parts:
  - A length-one part, containing the pivot itself
  - A length-zero part, and
  - A length-`(N-1)` part, containing everything else
- We don't recur on the zero-length part
- Recurring on the length `N-1` part requires (in the worst case) recurring to depth `N-1`



Worst case for quicksort

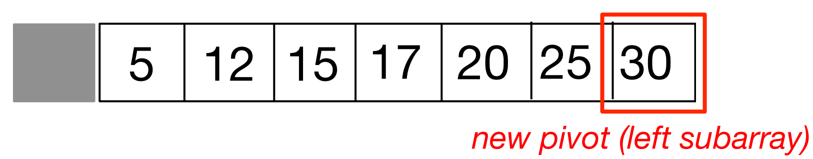
- In the worst case, recursion may be N levels deep (for an array of size N)
- But the partitioning work done at each level is still N
- $O(N) * O(N) = O(N^2)$
- So worst case for Quicksort is $O(N^2)$
- When does this happen?
  - When the array is sorted to begin with!

find the pivot position



PARTITION(A, p, r)

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



Bucketsort

- Suppose the values in the list to be sorted can repeat but they have a limit (e.g., values are digits from 0 to 9)
- Sorting, in this case, appears easier
- Is it possible to come up with an algorithm better than O(NlogN)?
- Yes, without comparisons

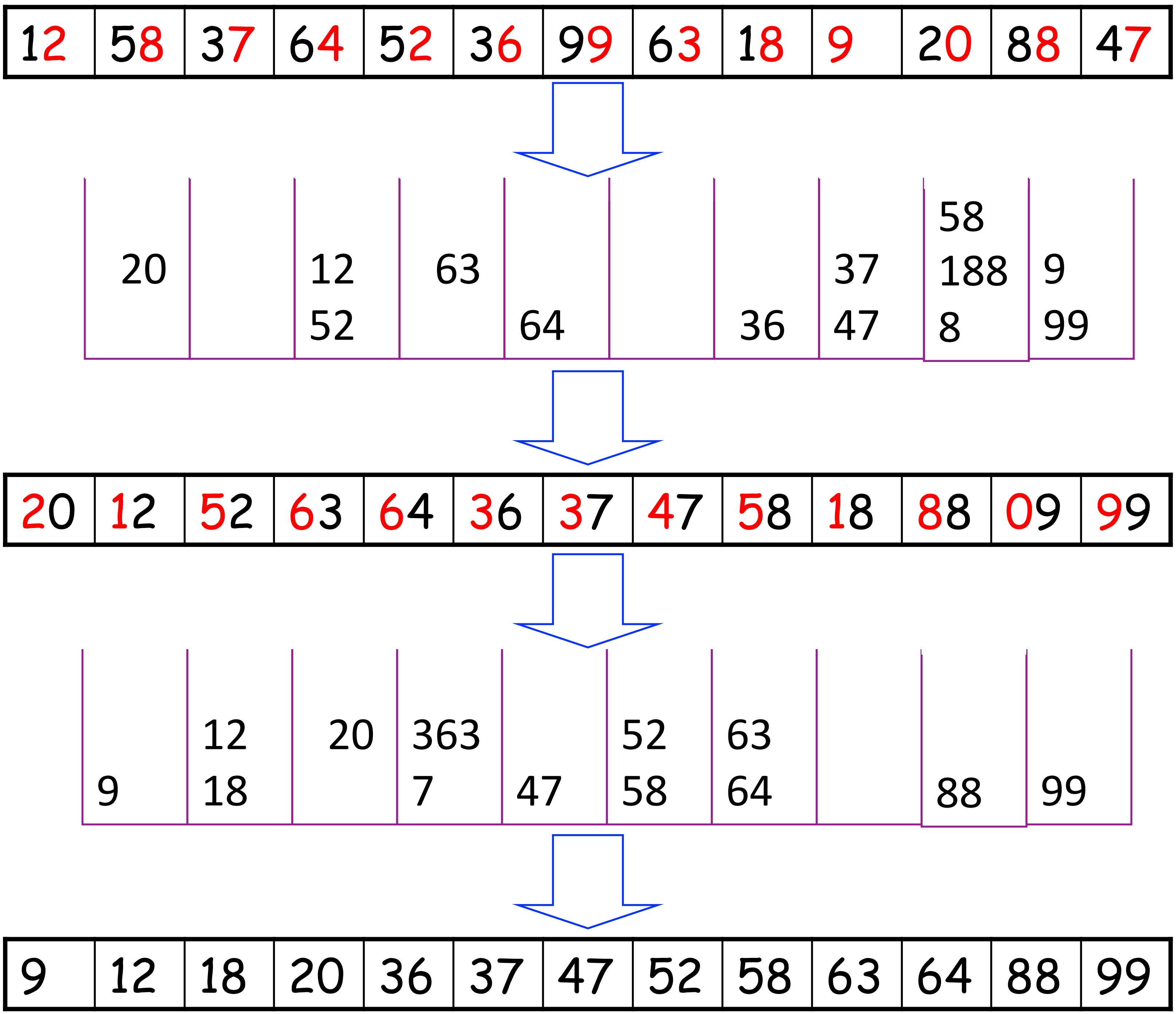
Idea

- suppose the values are in the range 0..m-1; start with m empty *buckets* numbered 0 to m-1
- scan the list and place element A[i] in bucket M[A[i]], and then output the buckets in order
- will need an array of buckets, and the values in the list to be sorted will be the indexes to the buckets
- no comparisons will be necessary
- each bucket can be an array or queue (to be placed back in array)
- Complexity: $O(m + N) = O(N)$, for $m << n$

radixsort

- If you are sorting 1000 integers and the maximum value is 999999, you will need 1 million buckets!
  - Time complexity increases dramatically to O(m)
- Can we do better?

- Idea:
- Perform successive bucketsorts by digit, starting with the rightmost
- In the example above, we need 10 buckets for each bucketsort
- Complexity: O(Np), with p=number of digits


12 58 37 64 52 36 99 63 18 9 20 88 47
20
12 52
63
64
36
37 47
58 188 8
9 99
20 12 52 63 64 36 37 47 58 18 88 09 99
9
12 18
20
363 7
47
52 58
63 64
88
99
9 12 18 20 36 37 47 52 58 63 64 88 99