Sorting & Growth of Functions

CSci 588: Data Structures, Algorithms and Software Design

Introduction to Algorithms, Cormen et al., Chapter 3

Overview

- Insertion Sort
  - Pseudocode
  - Loop Invariants
- Analyzing Algorithms
  - Summations
  - Best, Worst Expected Cases
  - Order of Growth
- Designing Algorithms
  - Divide and Conquer
  - Mergesort
  - Recursion

Insertion Sort

Pseudocode

Pseudocode is similar to actual programming language code (c, fortran, pascal, java, python, etc) except that it tries to present an algorithm in the most clear and concise way.

Pseudocode

Pseudocode usually ignores software engineering concerns to present the algorithms concisely.

Pseudocode can also use English if that is the easiest way to present a concept.

Pseudocode

You should read up about the book's conventions on pseudocode for more information.

Insertion Sort

Insertion sort solves the *sorting problem*:

Input: A sequence of *n* numbers $<a_1, a_2, ..., a_n>$
Output: A permutation (reordering) $<a'_1, a'_2, ..., a'_n>$ of the input sequence such that $a'_1 <= a'_2 <= ... <= a'_n$.

Insertion Sort

The insertion sort algorithm presented sorts the numbers *in place*, they are rearranged within the array instead of inserted into a new array. At most a constant number of values are stored outside that array at any time.

Insertion Sort

5	*2*	*4*	*6*	*1*	*3*

Start with an initial unordered array.

Insertion Sort

5	2	4	6	1	3

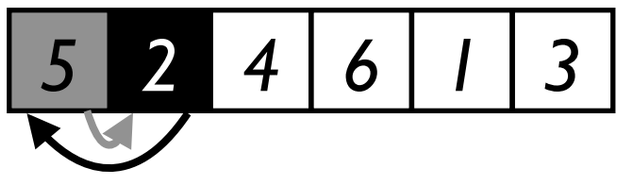
Pick a key (the element in black).

Insertion Sort

5	2	4	6	1	3

The section of the array in grey is already sorted. We can start with a key in the 2nd position because an array of size 1 is always sorted.

Insertion Sort



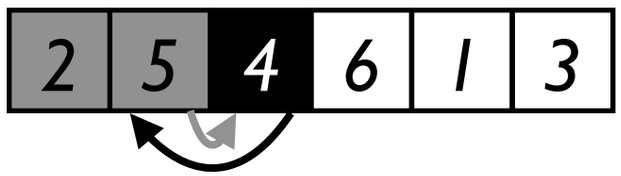
Insertion sort proceeds by inserting the key into the sorted array (hence the name).

Insertion Sort

2	5	4	6	1	3

Now we have a longer sorted array, so we can pick a new key (the element in black).

Insertion Sort



We insert the key (in black) into the sorted array (in grey).

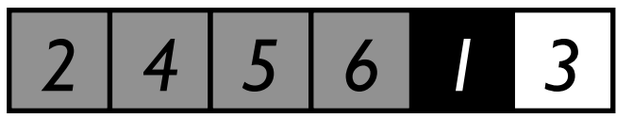
Insertion Sort

Our sorted portion of the array increases, so we select the next element as the key.

Insertion Sort

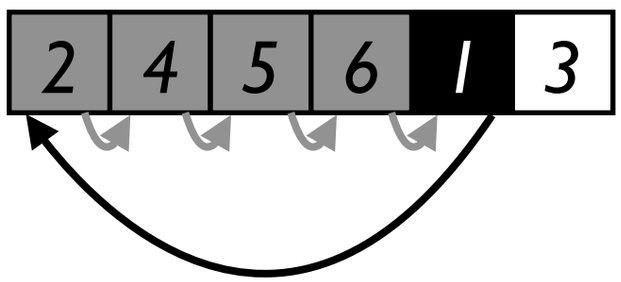
We insert the new key into the sorted array. It's already in the correct position.

Insertion Sort



We select the next element as the key.

Insertion Sort



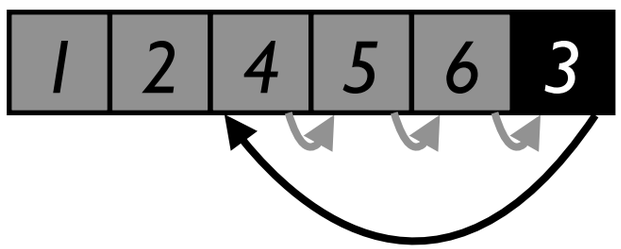
We insert it in the beginning of the sorted array and move everything after it to the right.

Insertion Sort

1	*2*	*4*	*5*	*6*	*3*

The last element of the array is our new key, and everything before it is sorted.

Insertion Sort



Insert the last key into the array and shift everything after it to the right.

Insertion Sort

1	*2*	*3*	*4*	*5*	*6*

We now have a sorted array.

Insertion Sort Pseudocode

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

Some important things to note:

- This pseudocode is slightly different from the book due to character limitations.
- Arrays indices in C++ go from 0 to length -1, the pseudocode in the book have array indices go from 1 to length.

Loop Invariants

```
INSERTION-SORT(A)
1.  for j = 2 to length[A]
2.      do key = A[j]
3.         //insert A[j] into the sorted sequence A[1..j - 1]
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
```

The index *j* indicates the "current card" being inserted into each hand.

At the beginning of the outer for loop (starting on line 1), the subarray A[1 ... *j* - 1] are the "sorted hand," while A[*j*+1 ... *length*] are the pile of unsorted cards.

Loop Invariants

At the beginning of the outer for loop (starting on line 1), the subarray A[1 ... *j* - 1] are the "sorted hand," while A[*j*+1 ... *length*] are the pile of unsorted cards.

We can state these properties of A[1 ... *j* - 1] formally as a *loop invariant*:

> At the start of each iteration of the **for** loop of lines 1-8, the subarray A[1 ... *j* - 1] consists of the elements originally in A[1 .. *j* - 1] but in sorted order.

Loop Invariants

Loop invariants can be used to help understand why an algorithm is correct, and also help us reason about it's speed.

Loop Invariants

Insertion sort loop invariant:

At the start of each iteration of the **for** loop of lines 1-8, the subarray A[1 ... *j* - 1] consists of the elements originally in A[1 .. *j* - 1] but in sorted order.

(This is similar to mathematical induction.)

Initialization: The loop invariant is true on the first iteration of the loop.

Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.

Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

Insertion Sort Loop Invariants

Insertion sort loop invariant:

At the start of each iteration of the **for** loop of lines 1-8, the subarray A[1 ... *j* - 1] consists of the elements originally in A[1 .. *j* - 1] but in sorted order.

Initialization: Show the loop invariant holds before the first iteration. When *j* == 2, the subarray is a single element, the first element of the array. So this subarray is sorted (trivially), which shows the invariant holds.

Insertion Sort Loop Invariants

Insertion sort loop invariant:

At the start of each iteration of the **for** loop of lines 1-8, the subarray A[1 ... *j* - 1] consists of the elements originally in A[1 .. *j* - 1] but in sorted order.

Maintenance: The body of the outer **for** loop moves A[*j*-1], A[*j*-2], A[*j*-3] and so on by one position to the right until the proper position for A[j] is found (lines 4 - 7). Informally, this keeps the array sorted to the left and right of where A[j] is inserted, preserving the loop invariant. But formally, we should also have a loop invariant for the inner loop.

Insertion Sort Loop Invariants

Insertion sort loop invariant:

At the start of each iteration of the **for** loop of lines 1-8, the subarray A[1 ... *j* - 1] consists of the elements originally in A[1 .. *j* - 1] but in sorted order.

Termination: The loop terminates when *j* exceeds the *length* of A, or when *j* = *length* + 1. Substituting *length* + 1 for *j* in the invariant, we get that the subarray A[1 ... *length*] consists of the elements originally in A[1 ... *length*] but in sorted order. Since this is the entire array, that proves our algorithm correct.

Analyzing Algorithms

Analyzing Algorithms

Analyzing an algorithm means predicting the resources that the algorithm will require.

This usually means computational time, but it can also apply to memory, communication bandwidth or other computer hardware usage.

Analyzing Algorithms

We will use a *random-access machine* (RAM) model of computation in this course. You can read the details in Chapter 2.2.

In short, it is important to note that in this model:

- Operations occur in sequential order (there aren't concurrent operations).
- Memory hierarchy effects are ignored (for simplicity).
- Each instruction takes a constant amount of time.

Analysis of Insertion Sort

The time taken to sort an array using insertion sort depends on the length of the array. For example, an array of length 3 can be sorted much faster than an array of length 1000.

This is true of most algorithms, so traditionally the runtime of an algorithm is described as a function of the size of it's input.

Analysis of Insertion Sort

The time taken to sort an array using insertion sort depends on the length of the array. For example, an array of length 3 can be sorted much faster than an array of length 1000.

This is true of most algorithms, so traditionally the runtime of an algorithm is described as a function of the size of it's input.

Runtime in Analysis

Runtime in analysis of algorithms on a particular input is typically presented as a number of primitive operations or "steps". It is an abstract count of things done (where doing those things takes equal time).

For the purposes of this course and the text, a single line of pseudocode takes a constant time (assuming it isn't calling another function). Formally, the *i*th line takes time c_i, where c_i is a constant.

In general this is a pretty fair assumption given how our model and most computers operate.

Breaking Down Insertion Sort

```
INSERTION-SORT(A)                                                    cost        times
1.  for j = 2 to length[A]                                           c1          n
2.      do key = A[j]                                                c2          n-1
3.         //insert A[j] into the sorted sequence A[1..j - 1]       0           n-1
4.         i = j - 1                                                 c4          n-1
5.         while i > 0 and A[i] > key                                c5          Σ_{j=2}^{n} t_j
6.           do A[i + 1] = A[i]                                      c6          Σ_{j=2}^{n} (t_j - 1)
7.              i = i - 1                                            c7          Σ_{j=2}^{n} (t_j - 1)
8.         A[i + 1] = key                                            c8          n-1
```

Since we know the cost of each line (and how many times it is executed), we know the total runtime, which is the sum of the cost each line multiplied by the number of times it is executed. So the runtime of INSERTION-SORT with an input of length n, or T(N) is:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \Sigma_{j=2}^{n}(t_j) + c_6 \Sigma_{j=2}^{n}(t_j - 1) + c_7 \Sigma_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

Where t_j is the number of times the inner loop (lines 5-7) is executed for that value of *j*.

Summations

If we take the following summation:

$$\Sigma_{j=2}^{n}(t_j)$$

We can set n = 5 (or any other value). If n = 5, then:

$$\Sigma_{j=2}^{5}(t_j) = t_2 + t_3 + t_4 + t_5$$

Then this is the sum of what comes after the summation for j = 2, 3, 4, 5.

Best Case for Insertion Sort

```
INSERTION-SORT(A)                                                    cost        times
1.  for j = 2 to length[A]                                            c1          n
2.      do key = A[j]                                                 c2          n-1
3.         //insert A[j] into the sorted sequence A[1..j - 1]        0           n-1
4.         i = j - 1                                                  c4          n-1
5.         while i > 0 and A[i] > key                                 c5          Σ_{j=2}^{n} t_j
6.           do A[i + 1] = A[i]                                       c6          Σ_{j=2}^{n} (t_j - 1)
7.              i = i - 1                                             c7          Σ_{j=2}^{n} (t_j - 1)
8.         A[i + 1] = key                                             c8          n-1
```

The best case for insertion sort happens when the array is already sorted. In this case, the statement A[*i*] > *key* on line 6 is always false and the loop immediately terminates. This means t_j = 1, and then:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \Sigma_{j=2}^{n}(t_j) + c_6 \Sigma_{j=2}^{n}(t_j - 1) + c_7 \Sigma_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1)$$

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

From the simplified version of T(n) we can se that it increases in runtime linearly with the size of n (as c_1 ... c_8 are all constants).

Worst Case for Insertion Sort

```
INSERTION-SORT(A)                                                    cost        times
1.  for j = 2 to length[A]                                            c1          n
2.      do key = A[j]                                                 c2          n-1
3.         //insert A[j] into the sorted sequence A[1..j - 1]        0           n-1
4.         i = j - 1                                                  c4          n-1
5.         while i > 0 and A[i] > key                                 c5          Σ_{j=2}^{n} t_j
6.           do A[i + 1] = A[i]                                       c6          Σ_{j=2}^{n} (t_j - 1)
7.              i = i - 1                                             c7          Σ_{j=2}^{n} (t_j - 1)
8.         A[i + 1] = key                                             c8          n-1
```

The worst case for insertion sort happens when the array is in reverse sorted order. In this case, the statement A[*i*] > *key* on line 5 is always true and the loop runs for all values of i. This means $t_j = j$. Since:

$$\Sigma_{j=2}^{n}(j) = \frac{n(n+1)}{2} - 1$$
$$\Sigma_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

We can fill in our original version of T(n):

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5(\frac{n(n+1)}{2} - 1) + c_6(\frac{n(n-1)}{2}) + c_7(\frac{n(n-1)}{2}) + c_8(n-1)$$

$$T(n) = (\frac{c_5+c_6+c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5-c_6-c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Worst Case for Insertion Sort

The worse case scenario has a runtime which increases quadratically:

$$T(n) = (\frac{c_5+c_6+c_7}{2})n^2 + (c_1 + c_2 + c_4 + \frac{c_5-c_6-c_7}{2} + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Because it is similar to the *quadratic function* (remember c_1 ... c_8 are constants, their value doesn't change when *n* changes):

$$an^2 + bn + c$$

Compare this to the best case scenario:

$$T(n) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Which is linear:

$$an + b$$

We can see that not only the size of the data, but also the ordering of the data can be extremely important in determining the runtime of an algorithm.

Worst and Average Case Analysis

Typically we worry about the worst case runtime far more than the best case runtime, for a few reasons:

- We don't know what input we'll pass to an algorithm be seeing before it gets run.
- It gives us an upper bound on the runtime of the algorithm.
- For some algorithms the worst case occurs often.
- The average case is often about as bad as the worst case. For example in insertion sort they both are quadratic functions (just different constants -- in the average case we have to do the inner loop $j/2$ times instead of j times).

Average/Expected Case Analysis

Later we will examine average or expected case analysis, which can be useful. However it does have some problems, mainly determining what the average or expected input would be -- and the average or expected input may be different for different applications.

Order of Growth

In the analysis of insertion sort, we said the worst case scenario was a quadratic function, while the best case scenario was linear. This abstracted the costs of the constants c_i. (Also note that the sum of a set of constants is just a different constant).

In algorithmic analysis, we typically abstract things one bit farther. We're really interested in the *rate* or *order of growth*, and this is dominated by the leading term of an equation.

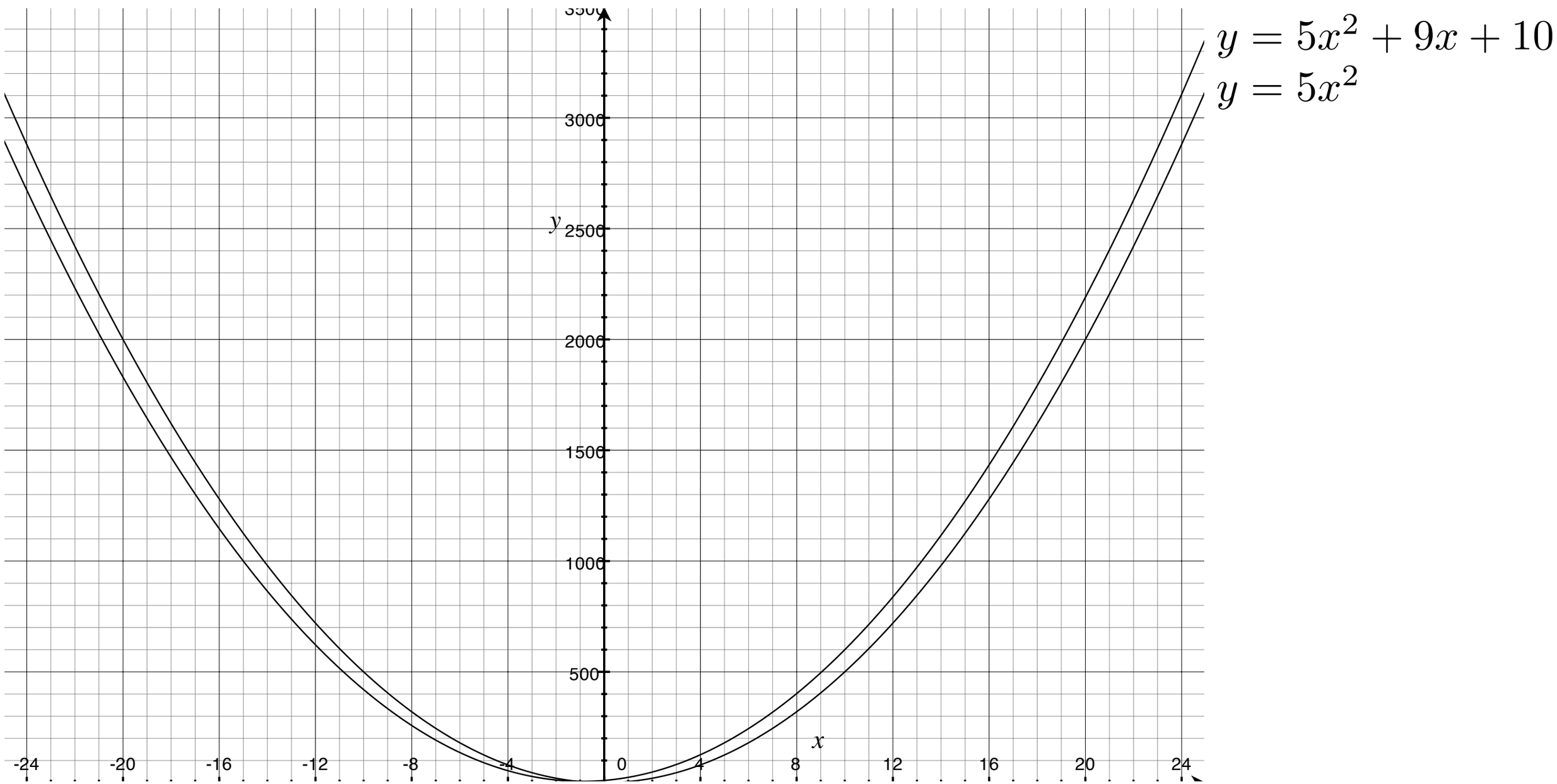
For example, for large values of *n*:

$$an^2 + bn + c \approx an^2$$

Order of Growth

For example, for large values of *n*:

$$an^2 + bn + c \approx an^2$$



Order of Growth

So for insertion sort, we say it's worst-case running time is:

$$O(n^2)$$

"big oh of n-squared" or "oh n-squared"

An algorithm is faster than another algorithm if its worst case running time has a lower order of growth.

Note that for small values of *n*, the 'faster' algorithm may be slower because of constants, but for as *n* increases this constants are overcome.

For example, even (if n is large enough):

$$2398472938472983742 3O(n^2) < O(n^3)$$

Designing Algorithms

Designing Algorithms

We saw with insertion sort that it used an incremental (or iterative) approach to designing the algorithm.

Mergesort uses a different approach, *divide-and-conquer*, which has a much faster worst case scenario than insertion sort.

Divide-and-conquer algorithms often are also easily analyzed with techniques described later in Chapter 4.

Divide-and-Conquer

Has three steps:

Divide the problem into a number of subproblems.

Conquer the subproblems by solving them recursively (divide them again if possible). If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

Mergesort

Has three steps:

Divide the *n*-element array into two subarrays of *n*/2 elements each.
Conquer the subproblems by sorting them using merge sort (which will continue to subdivide the subarrays until they are length 1).
Combine the two now sorted subsequences to produce the sorted result.

Sorting Two Sorted Arrays

The key to the mergesort algorithm is that it is possible to efficiently sort together two sorted arrays. In fact it is possible to merge the sorted subarrays into a sorted array in O(n) time.

MERGE Pseudocode

```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

MERGE Pseudocode

A is the array.

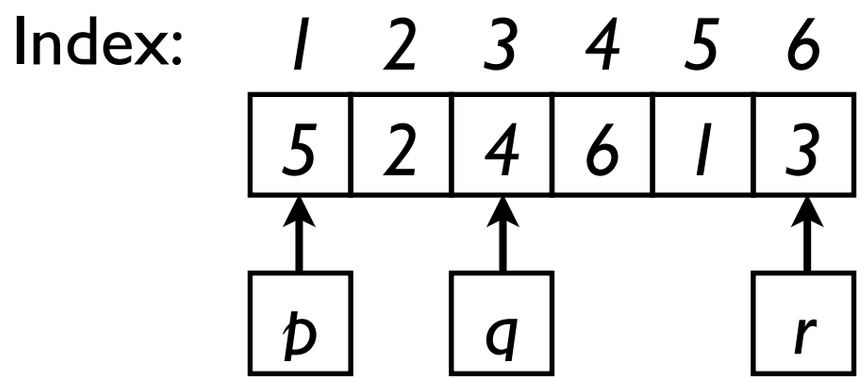
```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

MERGE Pseudocode

p <= q < r are indices in the array. These represent the beginning, middle and end of our subarrays.

```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
        then A[k] = L[i]
             i = i + 1
        else A[k] = R[j]
             j = j + 1
```

Array Indices




```
MERGE(A,p,q,r)
1.  n1 = q - p + 1
2.  n2 = r - q
```

If p = 1, q = 3, r = 6 then:

$n_1 = 3 - 1 + 1 = 3$
$n_2 = 6 - 3 \quad = 3$

MERGE Pseudocode

n_1 is the length of the first subarray
n_2 is the length of the second subarray

```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

MERGE Pseudocode

```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
        then A[k] = L[i]
             i = i + 1
        else A[k] = R[j]
             j = j + 1
```

These two loops create two subarrays and fill them in.

MERGE Pseudocode

```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

Put a placeholder value at the end of the arrays, note that they are 1 larger than the subarrays.

MERGE Pseudocode

```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
        then A[k] = L[i]
             i = i + 1
        else A[k] = R[j]
             j = j + 1
```

p to *r* will iterate over every element in the two subarrays.

MERGE Pseudocode

```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

If the current element in the left subarray is less than the current element in the right subarray, insert it into the current position in A.

MERGE Pseudocode

```
MERGE(A,p,q,r)
n1 = q - p + 1
n2 = r - q
create arrays L[1 .. n1+1] and R[1 .. n2+1]
for i = 1 to n1
    do L[i] = A[p + i - 1]
for j = 1 to n2
    do R[i] = A[q + j]
L[n1 + 1] = infinity //or maximum possible value
R[n1 + 1] = infinity //or maximum possible value
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

Otherwise insert the element at the current right array index in the right subarray.

MERGE Pseudocode

```
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

We'll start somewhere within the mergesort algorithm. The two subarrays have already been sorted, by calling mergesort on them, and they have been placed in the L and R arrays.

	8	9	10	11	12	13	14	15	16	17
A	...	2	4	5	7	1	2	3	6	...
	k									

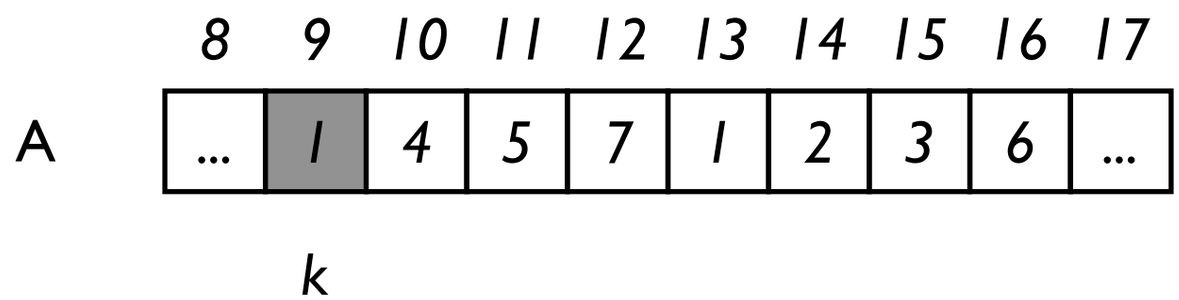
	1	2	3	4	5
L	2	4	5	7	inf
	i				

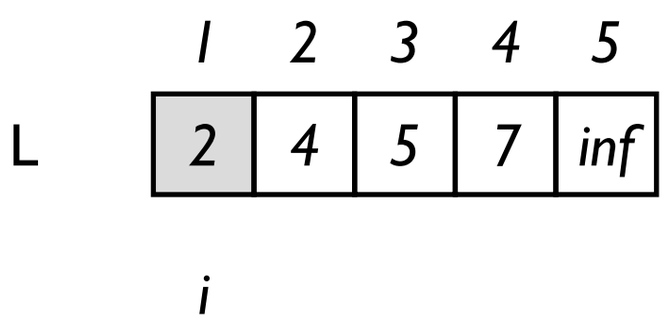
	1	2	3	4	5
R	1	2	3	6	inf
	j				

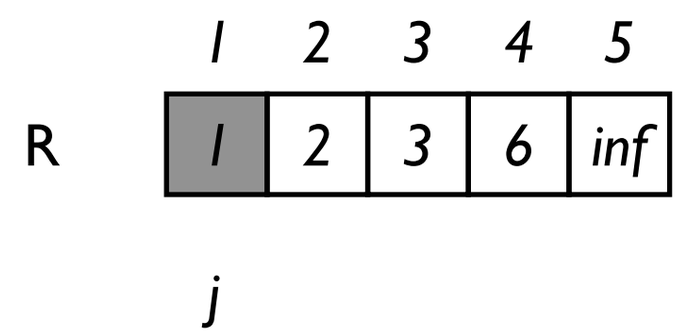
MERGE Pseudocode

```
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

```
i = 1, j = 1, k = 9
L[i] = 2
R[j] = 1

L[i] is greater than R[j], so we set A[k] =
R[j]
and j is incremented
```



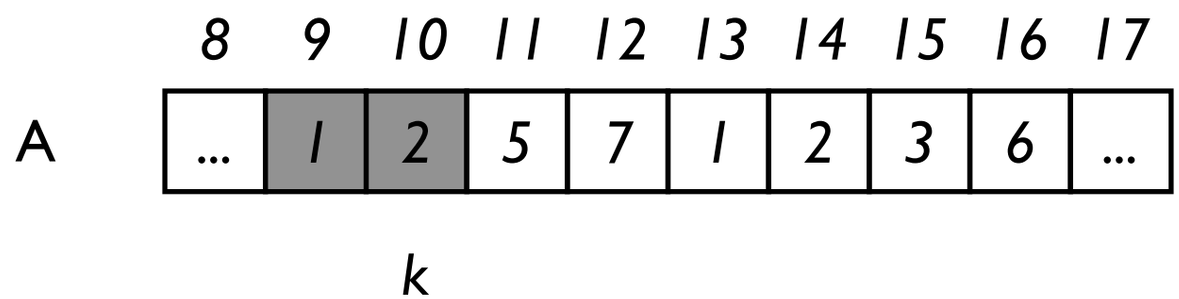


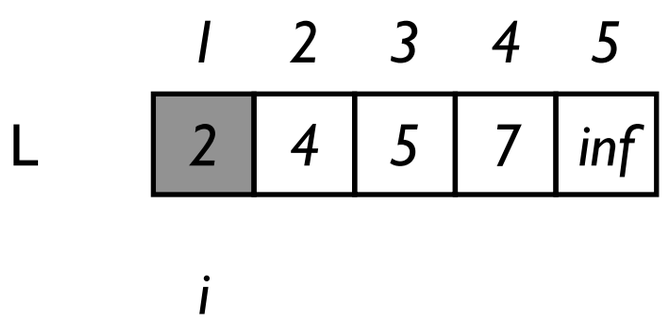


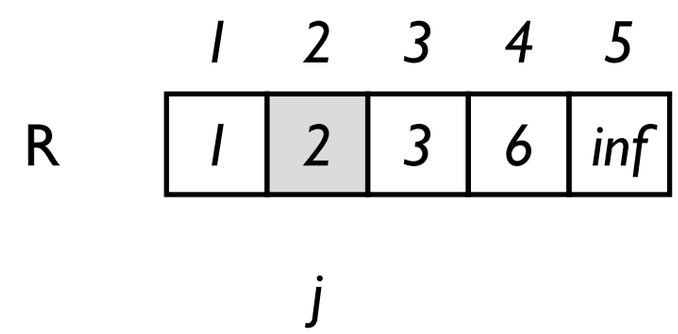
MERGE Pseudocode

```
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

```
i = 1, j = 2, k = 10
L[i] = 2
R[j] = 2

L[i] is less than or equal to R[j], so we set
A[k] = L[i]
and i is incremented
```



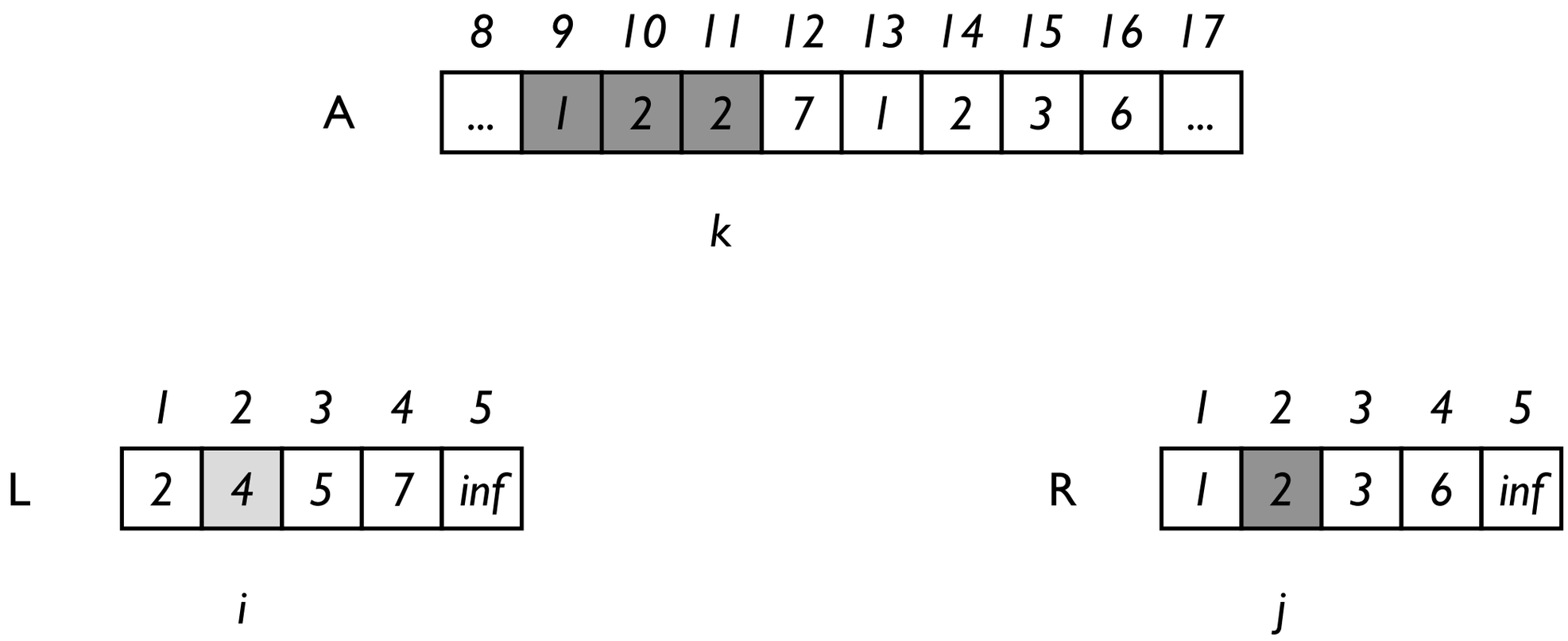




MERGE Pseudocode

```
for k = p to r
    do if L[i] <= R[j]
          then A[k] = L[i]
               i = i + 1
          else A[k] = R[j]
               j = j + 1
```

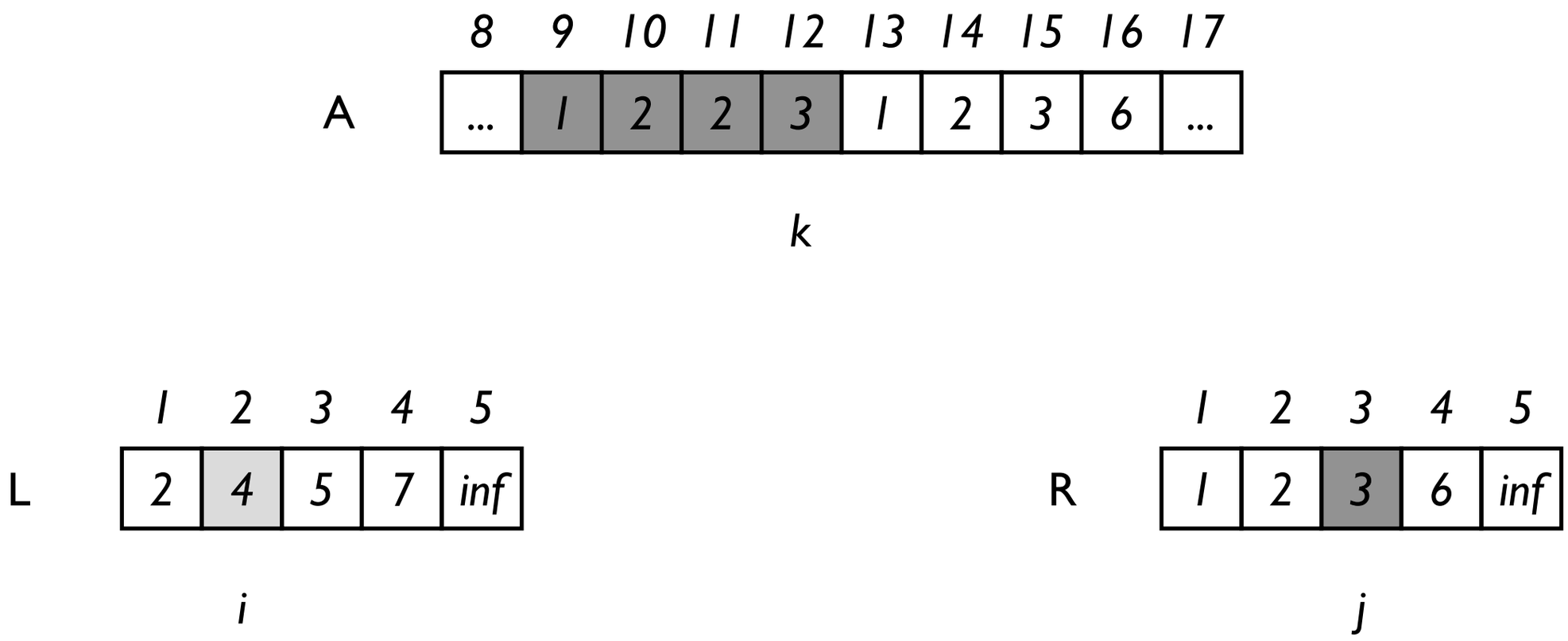
```
i = 2, j = 2, k = 11
L[i] = 4
R[j] = 2

L[i] is not less than or equal to R[j], so we
set A[k] = R[j]
and j is incremented
```



MERGE Pseudocode

```
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

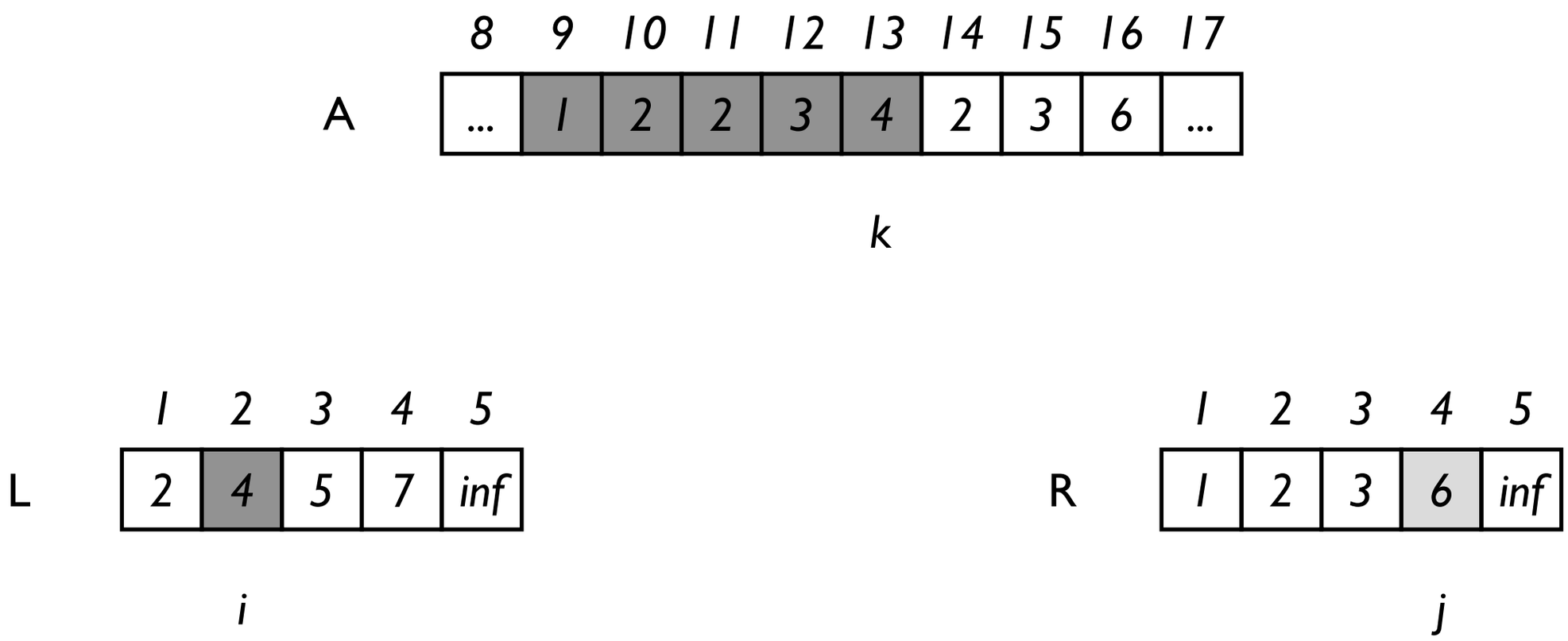
```
i = 2, j = 3, k = 12
L[i] = 4
R[j] = 3

L[i] is not less than or equal to R[j], so we
set A[k] = R[j]
and j is incremented
```



MERGE Pseudocode

```
for k = p to r
    do if L[i] <= R[j]
          then A[k] = L[i]
               i = i + 1
          else A[k] = R[j]
               j = j + 1
```

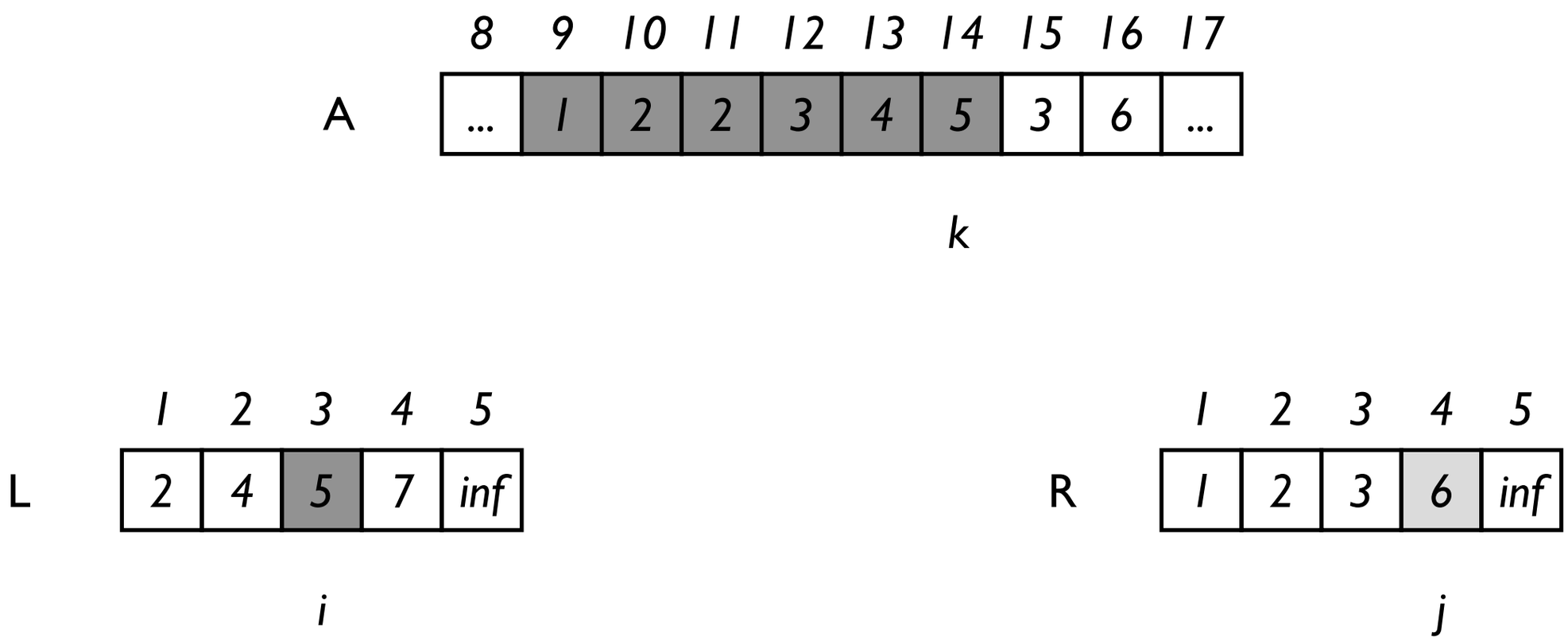
```
i = 2, j = 4, k = 13
L[i] = 4
R[j] = 6

L[i] is less than or equal to R[j], so we set
A[k] = L[i]
and i is incremented
```



MERGE Pseudocode

```
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

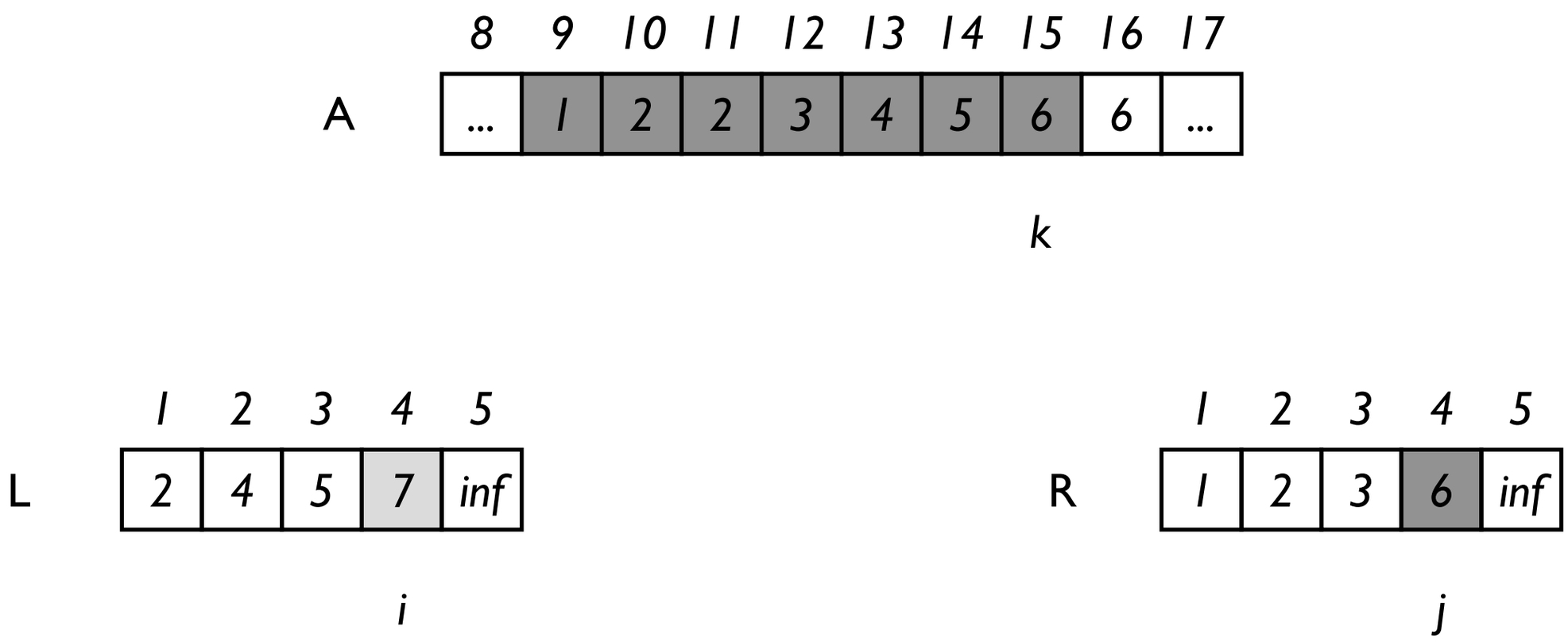
```
i = 3, j = 4, k = 14
L[i] = 5
R[j] = 6

L[i] is less than or equal to R[j], so we set
A[k] = L[i]
and i is incremented
```



MERGE Pseudocode

```
for k = p to r
    do if L[i] <= R[j]
       then A[k] = L[i]
            i = i + 1
       else A[k] = R[j]
            j = j + 1
```

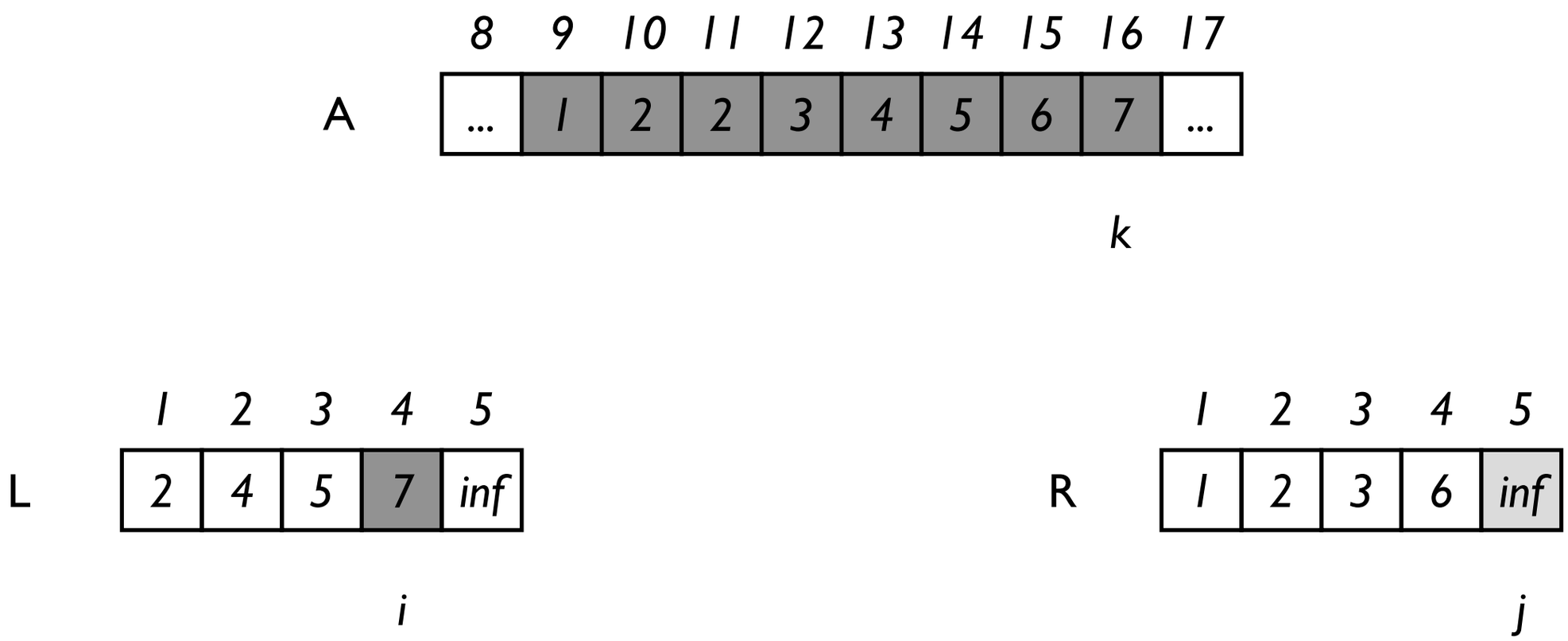
```
i = 4, j = 4, k = 15
L[i] = 7
R[j] = 6

L[i] is not less than or equal to R[j], so we
set A[k] = R[j]
and j is incremented
```



MERGE Pseudocode

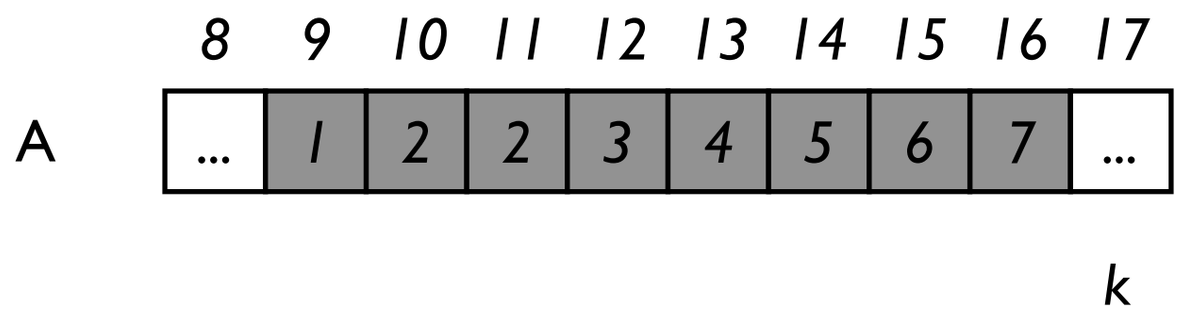
```
for k = p to r
    do if L[i] <= R[j]
          then A[k] = L[i]
               i = i + 1
          else A[k] = R[j]
               j = j + 1
```

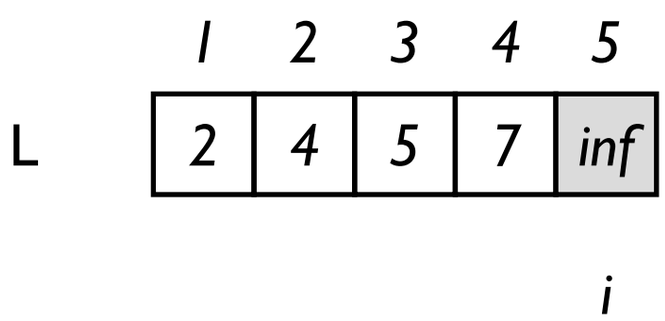
```
i = 4, j = 5, k = 16
L[i] = 7
R[j] = inf

L[i] is less than or equal to R[j], so we set
A[k] = L[i]
and i is incremented
```

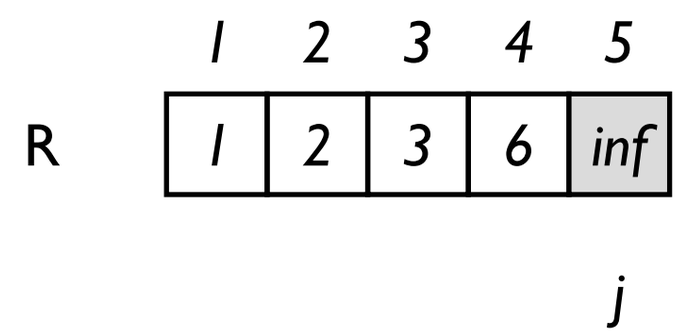


MERGE Pseudocode

```
for k = p to r                    i = 5, j = 5, k = 16
    do if L[i] <= R[j]            L[i] = inf
       then A[k] = L[i]           R[j] = inf
            i = i + 1
       else A[k] = R[j]           k is greater than r, so the for loop
            j = j + 1             terminates and we are finished.
```







MERGE Loop Invariant

```
MERGE(A,p,q,r)
1.  n1 = q - p + 1
2.  n2 = r - q
3.  create arrays L[1 .. n1+1] and R[1 .. n2+1]
4.  for i = 1 to n1
5.      do L[i] = A[p + i - 1]
6.  for j = 1 to n2
7.      do R[i] = A[q + j]
8.  L[n1 + 1] = infinity //or maximum possible value
9.  R[n2 + 1] = infinity //or maximum possible value
10. i = 1
11. j = 1
12. for k = p to r
13.     do if L[i] <= R[j]
14.           then A[k] = L[i]
15.                i = i + 1
16.           else A[k] = R[j]
17.                j = j + 1
```

At the start of each iteration of the `for` loop of lines 12-17, the subarray `A[`p`..`k`–1]` contains the $k - p$ smallest elements of `L[1..`n_1 `+ 1]` and `R[1..`n_2 `+ 1]`, in sorted order. Moreover, `L[`i`]` and `R[`j`]` are the smallest elements of their arrays that have not been copied back into `A`.

MERGE Loop Invariant

```
MERGE(A,p,q,r)
1.  n1 = q - p + 1
2.  n2 = r - q
3.  create arrays L[1 .. n1+1] and R[1 .. n2+1]
4.  for i = 1 to n1
5.      do L[i] = A[p + i - 1]
6.  for j = 1 to n2
7.      do R[i] = A[q + j]
8.  L[n1 + 1] = infinity //or maximum possible value
9.  R[n2 + 1] = infinity //or maximum possible value
10. i = 1
11. j = 1
12. for k = p to r
13.     do if L[i] <= R[j]
14.           then A[k] = L[i]
15.                i = i + 1
16.           else A[k] = R[j]
17.                j = j + 1
```

Important to note that they are the smallest elements because we added infinity into the L and R subarrays.

At the start of each iteration of the `for` loop of lines 12-17, the subarray `A[`p`..`k`–1]` contains the $k - p$ <u>smallest elements</u> of `L[1..`n_1 `+ 1]` and `R[1..`n_2 `+ 1]`, in sorted order. Moreover, `L[`i`]` and `R[`j`]` are the <u>smallest elements</u> of their arrays that have not been copied back into `A`.

MERGE Loop Invariant

Invariant: At the start of each iteration of the `for` loop of lines 12-17, the subarray `A[`p`..`k`–1]` contains the $k - p$ smallest elements of `L[1..`n_1 `+ 1]` and `R[1..`n_2 `+ 1]`, in sorted order. Moreover, `L[`i`]` and `R[`j`]` are the smallest elements of their arrays that have not been copied back into `A`.

Initialization: Before the first loop, $k = p$, so `A[`p`..`k `– 1]` is empty. It contains the $k - p = 0$ smallest elements of `L` and `R`, and since $i = j = 1$, both `L[`i`]` and `R[`j`]` are the smallest elements of their arrays that have not been copied back into `A` (`L` and `R` are sorted).

MERGE Loop Invariant

Invariant: At the start of each iteration of the `for` loop of lines 12-17, the subarray `A[`p`..`k`–1]` contains the $k - p$ smallest elements of `L[1..`n_1 `+ 1]` and `R[1..`n_2 `+ 1]`, in sorted order. Moreover, `L[`i`]` and `R[`j`]` are the smallest elements of their arrays that have not been copied back into `A`.

Maintenance: Suppose `L[`i`] <= R[`j`]`. Then `L[`i`]` is the smallest element not yet copied back into `A`. Because `A[`p`..`k`–1]` contains the smallest $k - p$ elements, after line 14 copies `L[`i`]` into `A[`k`]`, the subarray `A[`p`..`k`]` will contain the $k - p + 1$ smallest elements. Then incrementing k (in the for loop update) and i (in line 15) reestablishes the loop invariant.

If `L[`i`] > R[`j`]` then lines 16-17 perform a similar action to maintain the invariant.

MERGE Loop Invariant

Invariant: At the start of each iteration of the `for` loop of lines 12-17, the subarray $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into `A`.

Termination: At termination, $k = r + 1$. By the loop invariant, the subarray $A[p..k-1]$, which is $A[p..r]$, contains the $k - p = r - p + 1$ smallest elements of $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$, in sorted order. The arrays `L` and `R` together contain $n_1 + n_2 + 2 = r - p + 3$ elements. This means all but the two largest elements (the inifinities) have been copied back into `A`.

Merge Runtime

```
MERGE(A,p,q,r)
1.  n1 = q - p + 1
2.  n2 = r - q
3.  create arrays L[1 .. n1+1] and R[1 .. n2+1]
4.  for i = 1 to n1
5.      do L[i] = A[p + i - 1]
6.  for j = 1 to n2
7.      do R[i] = A[q + j]
8.  L[n1 + 1] = infinity //or maximum possible value
9.  R[n2 + 1] = infinity //or maximum possible value
10. i = 1
11. j = 1
12. for k = p to r
13.     do if L[i] <= R[j]
14.           then A[k] = L[i]
15.                i = i + 1
16.           else A[k] = R[j]
17.                j = j + 1
```

Lines 1-3 and 8-11 take constant time. The for loops of lines 4-7 each take O(n) time, and $O(n_1 + n_2) = O(n)$.

The for loop of lines 12-17 performs another n iterations, so again $O(n + n_3) = O(2n) = O(n)$.

Therefore, `MERGE` takes O(n) time.

MERGESORT

```
MERGESORT(A,p,r)                    //if p == r then the length of the
1.  if p < r                        //subarray is 1 and it is already sorted.
2.      q = floor((p + r) / 2)
3.      MERGESORT(A,p,q)
4.      MERGESORT(A,q + 1,r)
5.      MERGE(A,p,q,r)
```

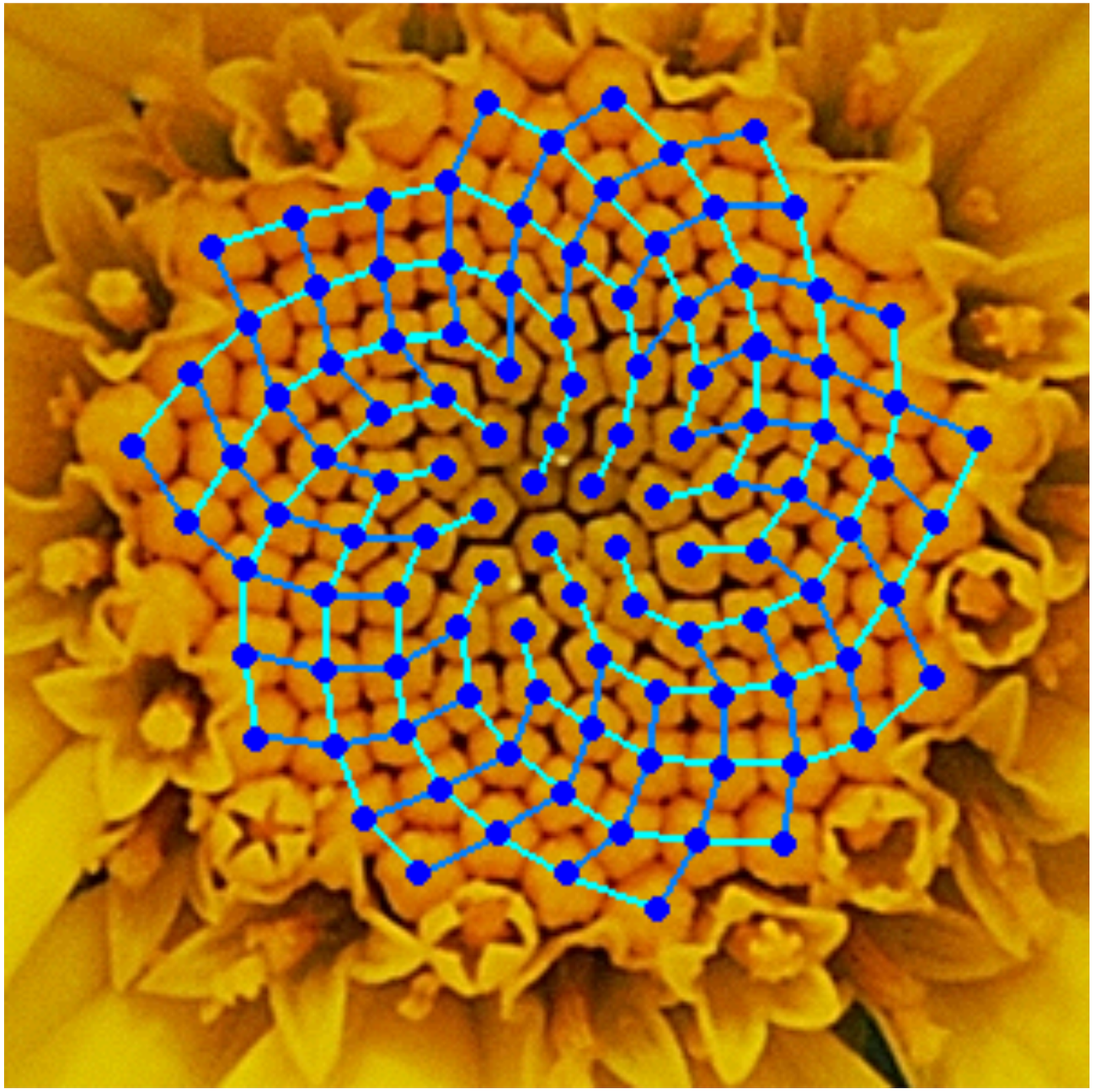
With `MERGE` defined we can finally get back to `MERGESORT`. We can see that `MERGE` does most of the work. All we need to do is get the middle of the array, call `MERGESORT` on either side and then `MERGE` to combine the two sorted subarrays.

To call `MERGESORT` on a whole array `A`, use *p* = 1 and *r* = `A.`*length*:

```
MERGESORT(A,1,A.length)
```

This is *recursion* -- having a function call itself, and it is a very useful and important programming concept.

Recursion



A common example of recursion is the Fibonacci sequence, which occurs often in nature.

The image on the left is the head of a chamomile flower, which is arranged with 21 (blue) and 13 (aqua) spirals -- Fibonacci numbers. This is a common occurrence in many plants, like pineapples fruitlets, artichoke flowering, and the arrangement of leaves on a stem.

The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Fibonacci Sequence

We can calculate the Fibonnaci sequence as follows:

```
FIBONACCI(i)
if (i == 0) return 0
if (i == 1) return 1
t1 = 0, t2 = 1, n = 0
for j = 2 to i
    n = t1 + t2
    t1 = t2
    t2 = n
return n;
```

Fibonacci Sequence

We can calculate the Fibonnaci sequence as follows:

```
FIBONACCI(i)
1. if (i == 0) return 0
2. if (i == 1) return 1
3. t1 = 0, t2 = 1, n = 0
4. for j = 2 to i
5.     n = t1 + t2
6.     t1 = t2
7.     t2 = n
8. return n;
```

Or using recursion:

```
FIBONACCI-REC(i)
1. if (i == 0) return 0
2. else if (i == 1) return 1
3. else return FIBONACCI-REC(i-1) + FIBONACCI-REC(i-2)
```

Fibonacci Sequence

We can calculate the Fibonnaci sequence as follows:

```
FIBONACCI(i)
1. if (i == 0) return 0
2. if (i == 1) return 1
3. t1 = 0, t2 = 1, n = 0
4. for j = 2 to i
5.     n = t1 + t2
6.     t1 = t2
7.     t2 = n
8. return n;
```

Or using recursion:

```
FIBONACCI-REC(i)
1. if (i == 0) return 0
2. else if (i == 1) return 1
3. else return FIBONACCI-REC(i-1) + FIBONACCI-REC(i-2)
```

The recursive version is much simpler!

Analyzing Divide-and-Conquer Algorithms

Analyzing D&C Algorithms

When algorithms (or functions) contain a call to themselves, we call this *recursion*.

For recursive algorithms, we usually describe their runtime using a *recurrence* (or *recurrence equation*).

Recurrence Equations

Recurrences are made from the three steps of the divide-and-conquer paradigm:

$T(n)$ is the running time on a problem of size n.

If the problem size is small enough (in the case of merge sort, $n == 1$, or really $n == 2$), $n <= c$, then the run time for that n, $T(n = 1) = O(1)$.

The original problem will be divided into a subproblems, each b the size of the original. It takes $T(n/b)$ time to solve a sub problem, and we need to solve a of them, so it takes $aT(n/b)$ to solve all of them.

It takes time $D(n)$ to divide the problem, and time $C(n)$ to combine the subproblems.

Recurrence Equations

Putting this together, we get:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \end{cases}$$

(aside) The LaTeX for this is:

```
$T(n) =
  \begin{cases}
     O(1) & \text{if } n \leq c \\
     D(n) + aT(n/b) + C(n)  & \text{otherwise}
  \end{cases}$
```

Recurrence Equations

Putting this together, we get:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \end{cases}$$

Our trivial case (two arrays of length 1 that are already sorted).

Recurrence Equations

Putting this together, we get:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \end{cases}$$

The time to divide our current problem into subproblems.

Recurrence Equations

Putting this together, we get:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \end{cases}$$

The time to calculate the subproblems.

Recurrence Equations

Putting this together, we get:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \end{cases}$$

The time to combine the subproblems.

Analyzing Mergesort

```
MERGESORT(A,p,r)                    //if p == r then the length of the
1.  if p < r                        //subarray is 1 and it is already sorted.
2.      q = floor((p + r) / 2)
3.      MERGESORT(A,p,q)
4.      MERGESORT(A,q + 1,r)
5.      MERGE(A,p,q,r)
```

We have our three steps:

Divide: (lines 1 and 2) This just finds the center of the current part of the array. The time of this is $D(n) = O(1)$, because it takes constant time.

Conquer: (lines 3 and 4) We need to solve two subproblems, each of size $n/2$. So filling in $a = 2$ and $b = 2$, this part takes $aT(n/b) = 2T(n/2)$ time.

Combine: (line 5) We have already figured out the runtime of the MERGE algorithm, which takes $O(n)$ time, so $C(n) = O(n)$.

Analyzing Mergesort

```
MERGESORT(A,p,r)                    //if p == r then the length of the
1.  if p < r                        //subarray is 1 and it is already sorted.
2.      q = floor((p + r) / 2)
3.      MERGESORT(A,p,q)
4.      MERGESORT(A,q + 1,r)
5.      MERGE(A,p,q,r)
```

Combining everything from the previous steps, we get:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ D(n) + aT(n/b) + C(n) & \text{otherwise} \end{cases}$$

or, because the O(*n*) dominates the O(1):

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ 2T(n/2) + O(n) & \text{otherwise} \end{cases}$$

The "Master Theorem"

```
MERGESORT(A,p,r)                    //if p == r then the length of the
1.  if p < r                        //subarray is 1 and it is already sorted.
2.      q = floor((p + r) / 2)
3.      MERGESORT(A,p,q)
4.      MERGESORT(A,q + 1,r)
5.      MERGE(A,p,q,r)
```

The "master theorem" shows that:

$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ 2T(n/b) + O(n) & \text{otherwise} \end{cases}$$

is equivalent to T(*n*) = O(*n* lg *n*).

This is proven in detail in Introduction to Algorithms - Chapter 4 (so we won't get into that).

Analyzing Mergesort

We can rewrite:

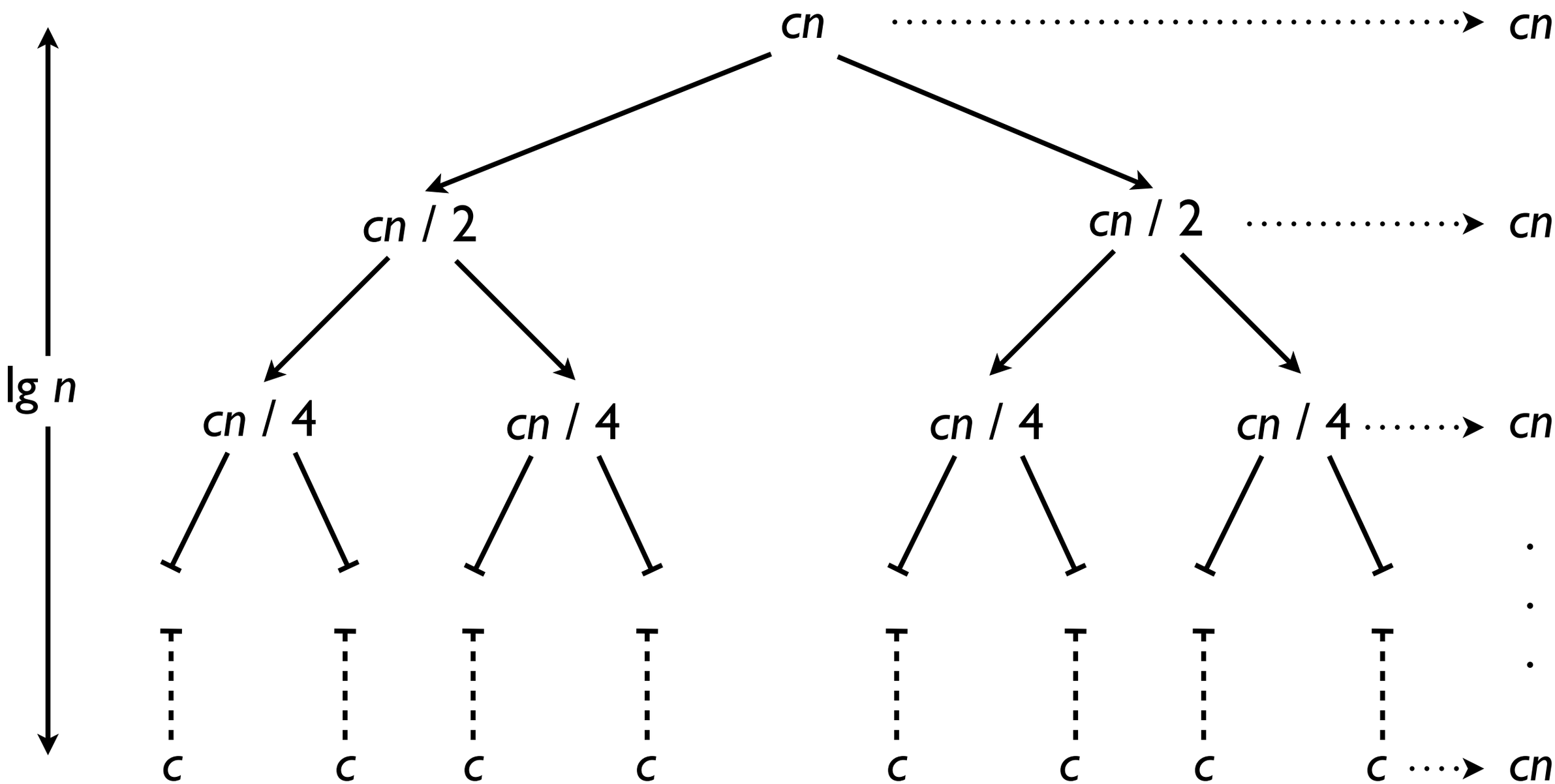
$$T(n) = \begin{cases} O(1) & \text{if } n \leq c \\ 2T(n/b) + O(n) & \text{otherwise} \end{cases}$$

As:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Because O(1) is constant time, and O(*n*) ignores constants multiplied by *n*.

Analyzing Mergesort



The different levels of mergesort all add up to a runtime of *cn* and there are lg *n* levels, so the runtime is O(*n* lg *n*)

Analyzing Mergesort

We can show that there are *n* lg *n* levels, because:

Level 1 has 1 leaf.	lg 1 = 1
Level 2 has 2 leaves.	lg 2 = 2
Level 3 has 4 leaves.	lg 4 = 3
Level 4 has 8 leaves.	lg 8 = 4
Level 5 has 16 leaves.	lg 16 = 5
...	...
Level n has n^2 leaves.	lg n^2 = n