Algorithms III

Agenda

Sorting

- Simple sorting algorithms
- Shellsort
- Mergesort
- Quicksort
- A lower bound for sorting
- Counting sort and Radix sort

Randomization

- Random number generation
- Generation of random permutations
- Randomized algorithms

Sorting

Informal definition:
Sorting is any process of arranging items in some sequence.

Why sort?

(1) Searching a sorted array is easier than searching an unsorted array. This is especially true for people. Example: Looking up a person in a telephone book.

(2) Many problems may be solved more efficiently when input is sorted. Example: If two files are sorted it is possible to find all duplicates in only one pass.

Detecting duplicates in an array

```
1     // Return true if array a has duplicates; false otherwise
2     public static boolean duplicates( Object [ ] a )
3     {
4         for( int i = 0; i < a.length; i++ )
5             for( int j = i + 1; j < a.length; j++ )
6                 if( a[ i ].equals( a[ j ] ) )
7                     return true;   // Duplicate found
8
9         return false;              // No duplicates found
10    }
```

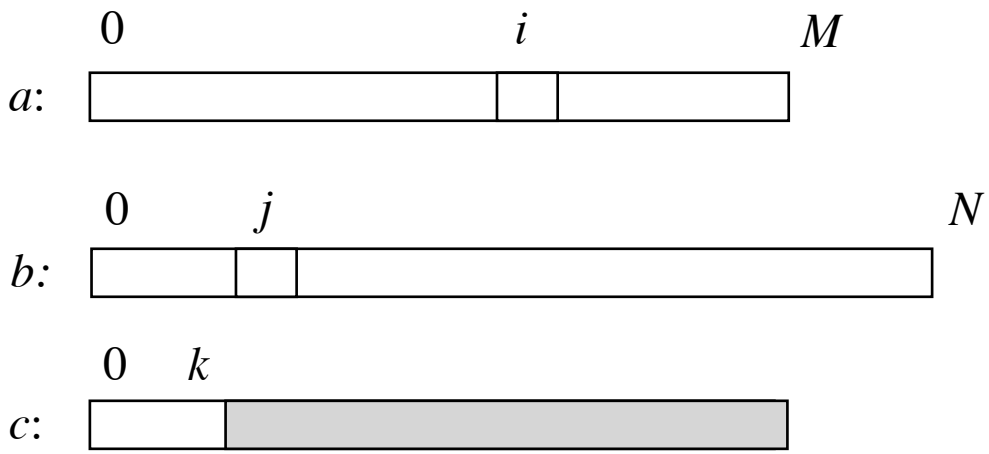
figure 8.1

A simple quadratic algorithm for detecting duplicates

If the array is sorted, duplicates can be detected by a linear-time scan of the array:

```
for (int i = 1; i < a.length; i++)
    if (a[i - 1].equals(a[i])
        return true;
```

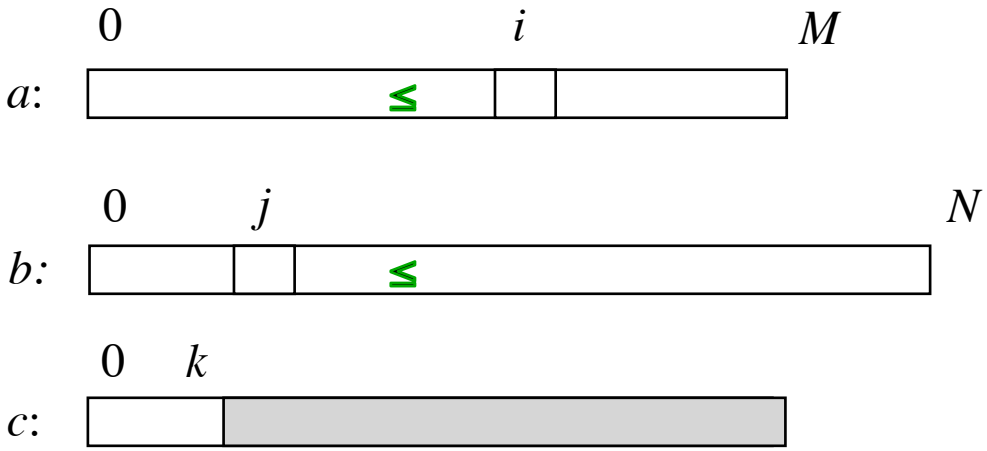
Finding the intersection of two unsorted arrays




```
k = 0;
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        if (a[i].equals(b[j]))
            c[k++] = a[i];
```

Complexity: $O(MN)$

Finding the intersection of two sorted arrays




```
i = j = k = 0;
while (i < M && j < N)
    if (a[i] < b[j]) i++;
    else if (a[i] > b[j]) j++;
    else { c[k++] = a[i]; i++; j++; }
```

Complexity: $O(M + N)$

Insertion sort

Problem: Given an array a of n elements
$a[0], a[1], ..., a[n\text{-}1]$.
Sort the array in increasing order.

Solution (by induction):

Base case: We know how to sort 1 element.
Induction hypothesis: We know how to sort n-1 elements.

We can sort an array of n elements by
(1) sorting its first n-1 elements,
(2) **insert** the n'th element into its proper place among the previously sorted elements

Insertion sort

(recursive version)

```
void insertionSort(int[] a, int n) {
    if (n > 1) {
        insertionSort(a, n - 1);
        int tmp = a[n];
        int j = n;
        for ( ; j > 0 && a[j - 1] > tmp; j--)
            a[j] = a[j - 1];
        a[j] = tmp;
    }
}
```

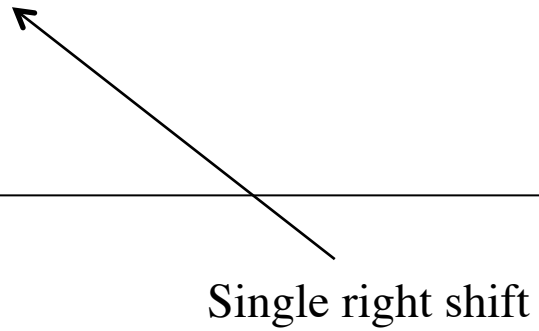


figure 8.3

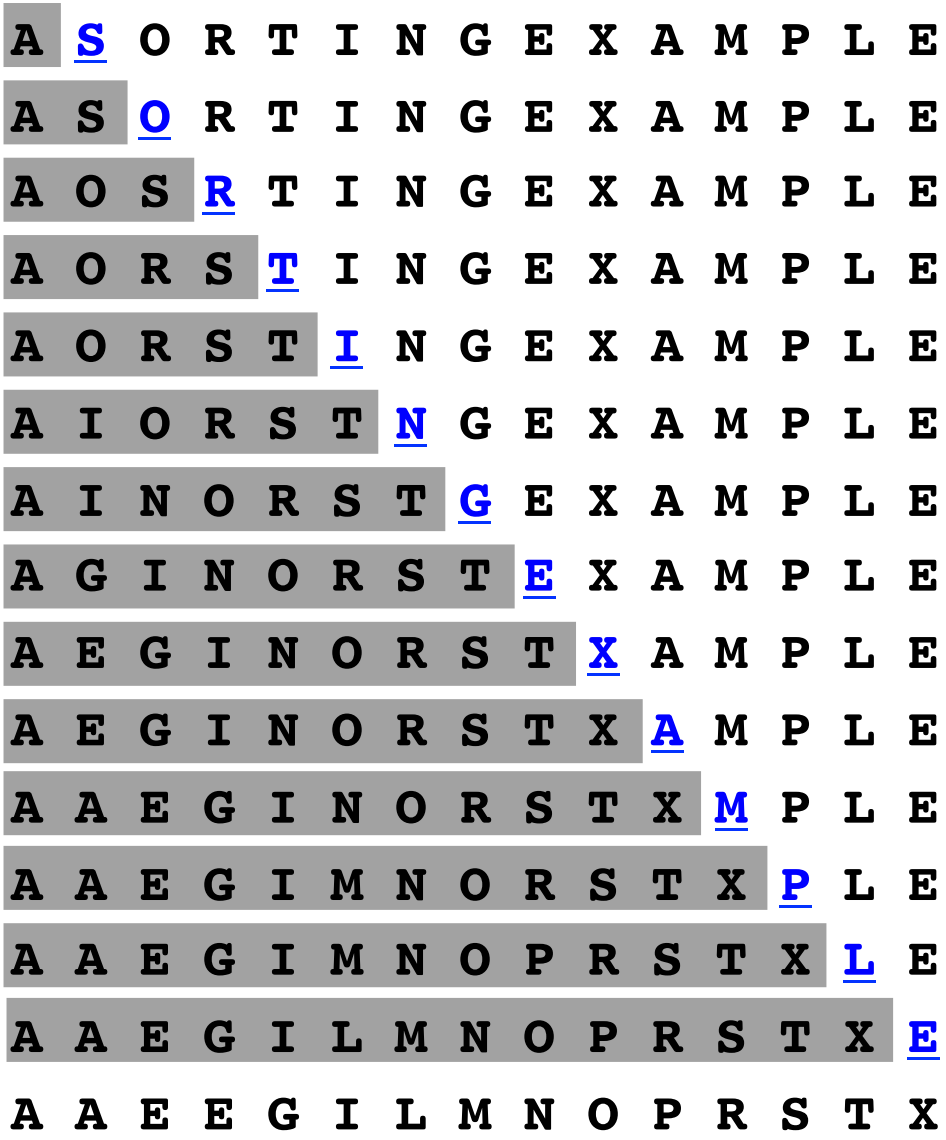
Basic action of insertion sort (the shaded part is sorted)

Array Position	0	1	2	3	4	5
Initial State	8	5	9	2	6	3
After `a[0..1]` is sorted	5	8	9	2	6	3
After `a[0..2]` is sorted	5	8	9	2	6	3
After `a[0..3]` is sorted	2	5	8	9	6	3
After `a[0..4]` is sorted	2	5	6	8	9	3
After `a[0..5]` is sorted	2	3	5	6	8	9

Array Position	0	1	2	3	4	5
Initial State	8	**5**				
After a[0..1] is sorted	5	8	**9**			
After a[0..2] is sorted	5	8	9	**2**		
After a[0..3] is sorted	2	5	8	9	**6**	
After a[0..4] is sorted	2	5	6	8	9	**3**
After a[0..5] is sorted	2	3	5	6	8	9

figure 8.4

A closer look at the action of insertion sort (the dark shading indicates the sorted area; the light shading is where the new element was placed)


A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A O S R T I N G E X A M P L E
A O R S T I N G E X A M P L E
A O R S T I N G E X A M P L E
A I O R S T N G E X A M P L E
A I N O R S T G E X A M P L E
A G I N O R S T E X A M P L E
A E G I N O R S T X A M P L E
A E G I N O R S T X A M P L E
A A E G I N O R S T X M P L E
A A E G I M N O R S T X P L E
A A E G I M N O P R S T X L E
A A E G I L M N O P R S T X E
A A E E G I L M N O P R S T X

Insertion sort

(non-recursive version)

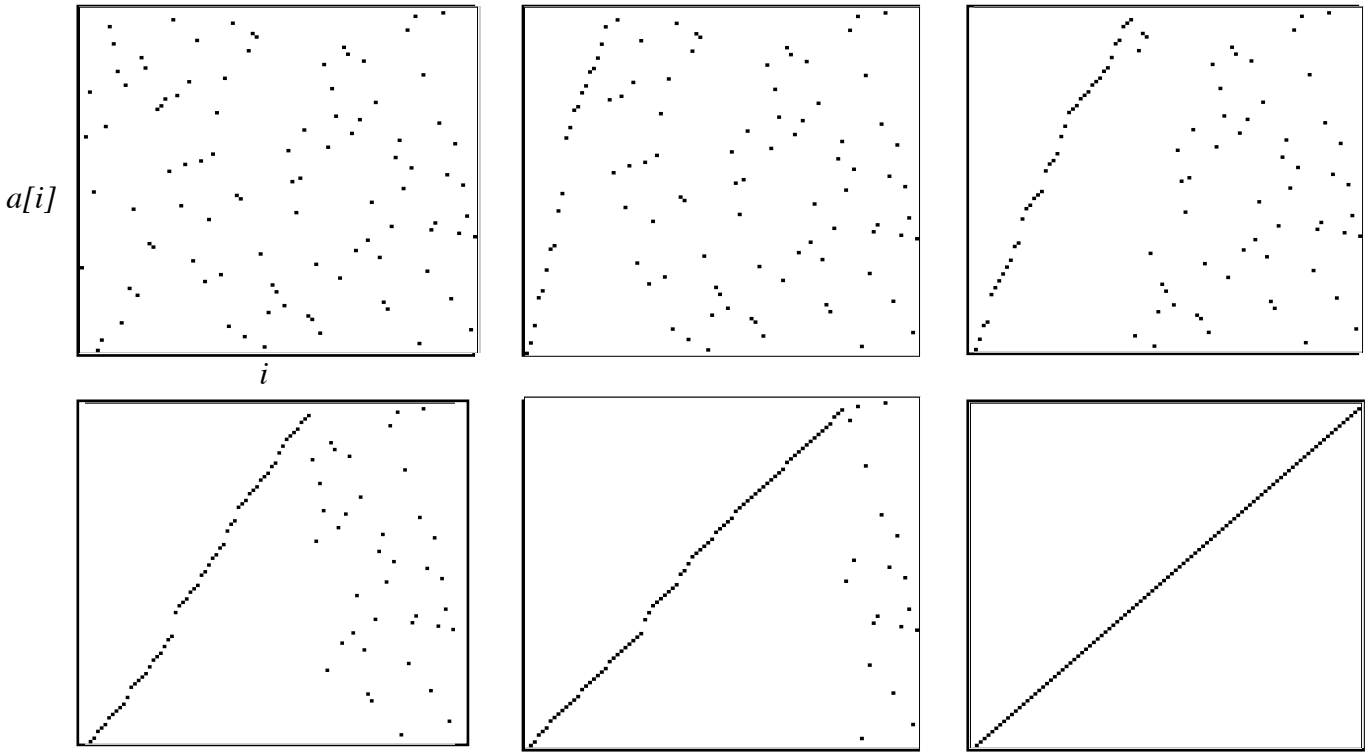
figure 8.2

Insertion sort implementation

```
1     /**
2      * Simple insertion sort
3      */
4     public static <AnyType extends Comparable<? super AnyType>>
5     void insertionSort( AnyType [ ] a )
6     {
7         for( int p = 1; p < a.length; p++ )
8         {
9             AnyType tmp = a[ p ];
10            int j = p;
11
12            for( ; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
13                a[ j ] = a[ j - 1 ];
14            a[ j ] = tmp;
15        }
16    }
```

Only the object references are right-shifted (not the objects themselves)

Animation of Insertion sort



Sorting the integers from 1 to 100.

Analysis of Insertion sort

Number of **comparisons**:

Best case: n-1

Worst case: $1 + 2 + ... + (n\text{-}1) = n(n\text{-}1)/2$, which is $O(n^2)$

Average case: $n(n\text{-}1)/4$, which is $O(n^2)$

Number of **moves**:

Best case: 0

Worst case: $1 + 2 + ... + (n\text{-}1) = n(n\text{-}1)/2$, which is $O(n^2)$

Average case: $n(n\text{-}1)/4$, which is $O(n^2)$

Running time is $O(n)$ for “almost sorted” arrays

Shellsort

Donald Shell, 1959

Idea:

Insertion sort is very efficient when the array is "almost sorted". However, it is slow for "very unsorted" arrays since it only allows exchange of neighboring elements.

Question:

Can we start by exchanging elements that are far apart from each other in the array before we do an ordinary Insertion sort?

Answer:

Yes. We can start by sorting all subfiles consisting of every h'th element of the array, where $h > 1$.

4-sorting

1. Divide the array into 4 subfiles:

   each 4'th element, starting in the first,
   each 4'th element, starting in the second,
   each 4'th element, starting in the third, and
   each 4'th element, starting in the fourth.

2. Sort each of these four subfiles.

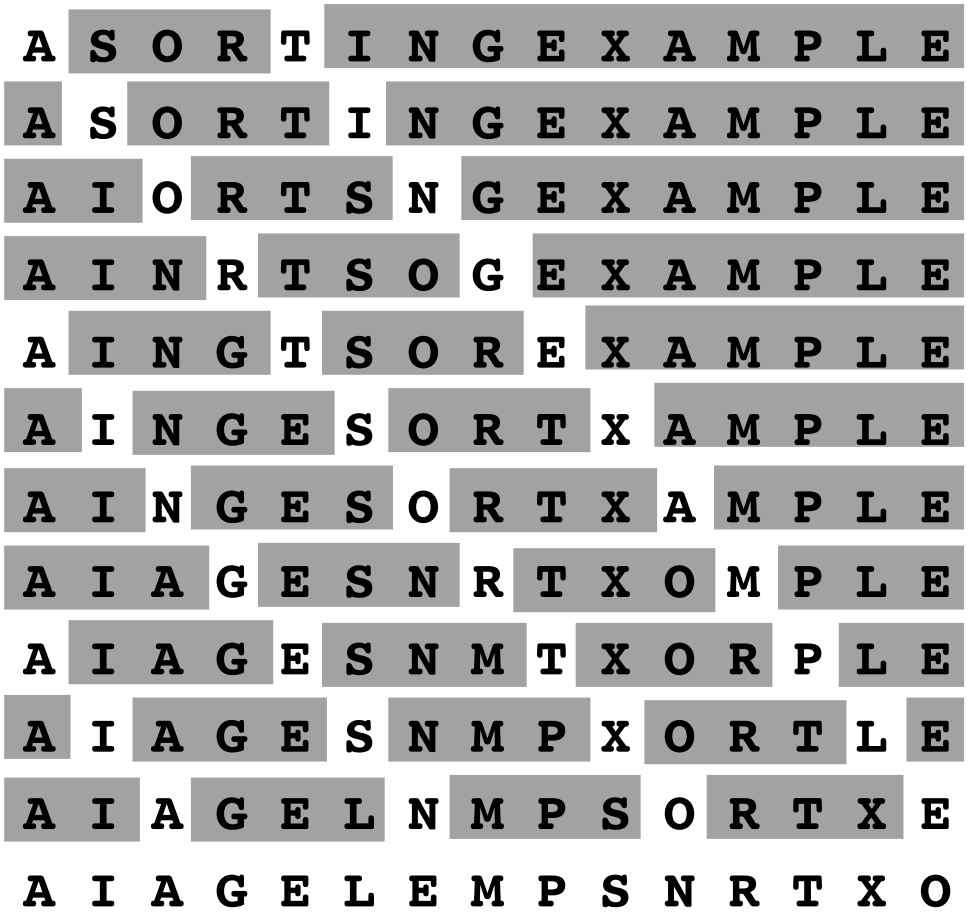
The array is then said to be **4-sorted**.

Similarly, we can define ***h*-sorted**.

Notice that an array that is **1-sorted** is sorted.

4-sorting

Use Insertion sort with an increment of 4



Implementation of *h*-sort

```
public static <AnyType extends Comparable<? super AnyType>>
void h_sort(AnyType[] a, int h) {
    for (int p = h; p < a.length; p++) {
        AnyType tmp = a[p];
        int j = p;
        for ( ; j >= h && tmp.compareTo(a[j - h]) < 0; j -= h)
            a[j] = a[j - h];
        a[j] = tmp;
    }
}
```

The code is obtained by replacing `1` by `h` in `insertionSort`.

Shellsort

Shellsort uses *h*-sorting for a decreasing sequence of *h*-values, ending with $h = 1$.

```
void shellsort(AnyType[] a) {
    for (int h = a.length / 2; h > 0;
         h = h == 2 ? 1 : (int) (h / 2.2))
        h_sort(a, h);
}
```

```
1       /**
2        * Shellsort, using a sequence suggested by Gonnet.
3        */
4       public static <AnyType extends Comparable<? super AnyType>>
5       void shellsort( AnyType [ ] a )
6       {
7           for( int gap = a.length / 2; gap > 0;
8                        gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) )
9               for( int i = gap; i < a.length; i++ )
10              {
11                  AnyType tmp = a[ i ];
12                  int j = i;
13
14                  for( ; j >= gap && tmp.compareTo( a[j-gap] ) < 0; j -= gap )
15                      a[ j ] = a[ j - gap ];
16                  a[ j ] = tmp;
17              }
18      }
```

figure 8.7

Shellsort implementation

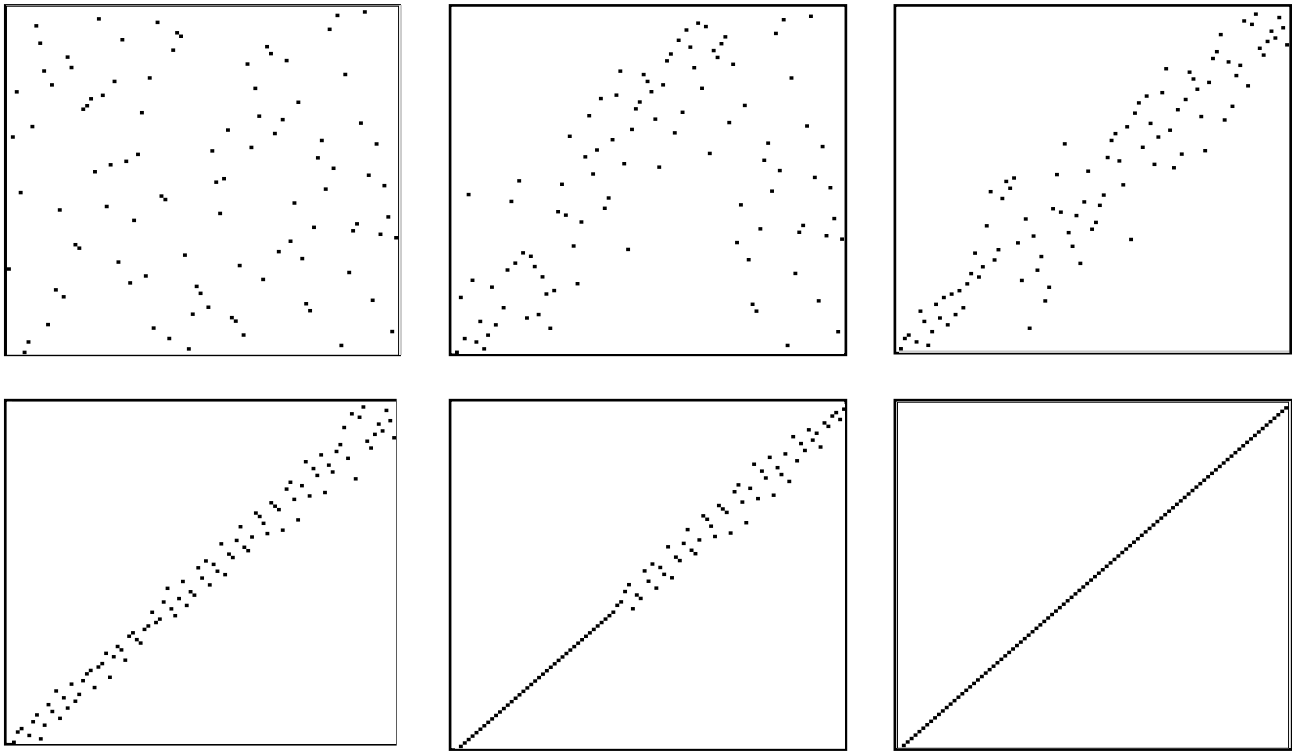
figure 8.5

Shellsort after each pass if the increment sequence is {1, 3, 5}

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

In the figure, elements spaced five and three apart are identically shaded.

Animation of Shellsort



		Shellsort		
N	Insertion Sort	Shell's Increments	Odd Gaps Only	Dividing by 2.2
10,000	575	10	11	9
20,000	2,489	23	23	20
40,000	10,635	51	49	41
80,000	42,818	114	105	86
160,000	174,333	270	233	194
320,000	NA	665	530	451
640,000	NA	1,593	1,161	939

figure 8.6 (milliseconds)

Running time of the insertion sort and Shellsort for various increment sequences

Shell's increments: Start gaps at $n/2$ and halve until 1 is reached.
Odd Gaps Only: As Shell's increments, but add 1 whenever the gap becomes even.

Analysis of Shellsort

Number of **comparisons** (for the increment sequence 1, 4, 13, 40,...):

Best case: $(n\text{-}1) + (n\text{-}4) + (n\text{-}13) + \ldots \leq n \log_3 n$

Worst case: less than $n^{1.5}$

Average case: not known

Two suggestions are $O(n^{1.25})$ and $O(n(\log n)^2)$

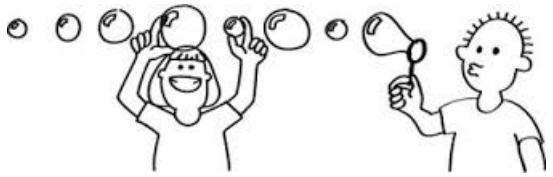
Number of **moves**:

Best case: 0

Worst case: as for comparisons

Average case: as for comparisons

Bubble Sort



Repeatedly **swap adjacent elements** if they are in wrong order.

```
boolean swapped;
do {
    swapped = false;
    for (int i = 1; i < a.length; i++) {
        if (a[i - 1] > a[i]) {
            swap(a, i - 1, i);
            swapped = true;
        }
    }
} while (swapped);
```

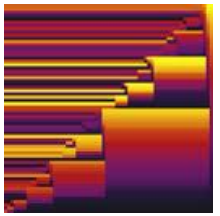
Worst-case and average complexity: $O(n^2)$
Best case complexity: $O(n)$

Improved Bubble Sort

All elements after the last swap are sorted, and do not need to be checked again.

```
int n = a.length;
do {
    int newn = 0;
    for (int i = 1; i < n; i++) {
        if (a[i - 1] > a[i]) {
            swap(a, i - 1, i);
            newn = i;
        }
    }
    n = newn;
} while (n > 0);
```

Worst-case and average complexity: $O(n^2)$
Best case complexity: $O(n)$



Mergesort

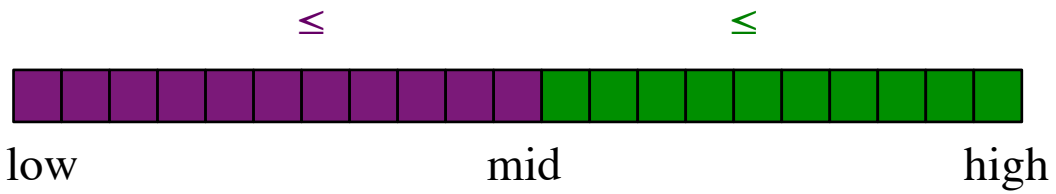
Mergesort is a divide-and-conquer based sorting algorithm.

The algorithm involves three steps:

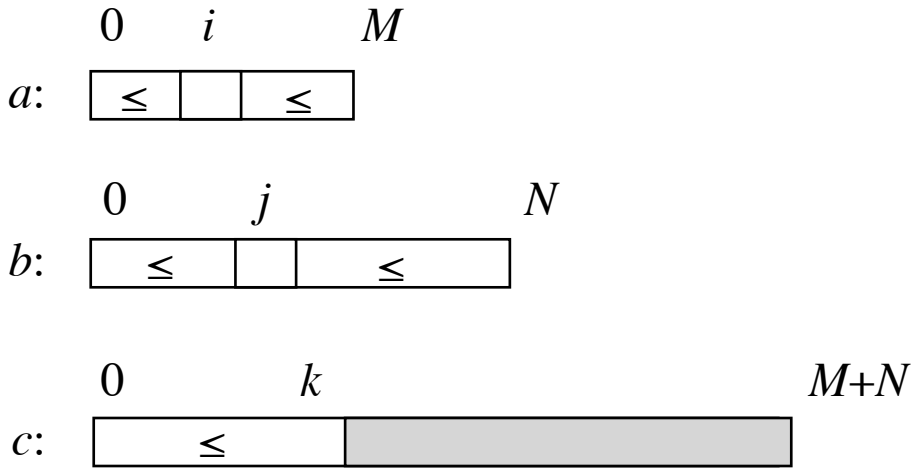
1. If the number of items to sort is 0 or 1, then return.
2. Recursively sort the first and second halves separately.
3. Merge the two sorted halves into one sorted group.

Mergesort

```
void mergeSort(Comparable[] a, int low, int high) {
    if (low < high) {
        int mid = (low + high) / 2;
        mergesort(a, low, mid);
        mergesort(a, mid + 1, high);
        merge(a, low, mid, high);
    }
}
```

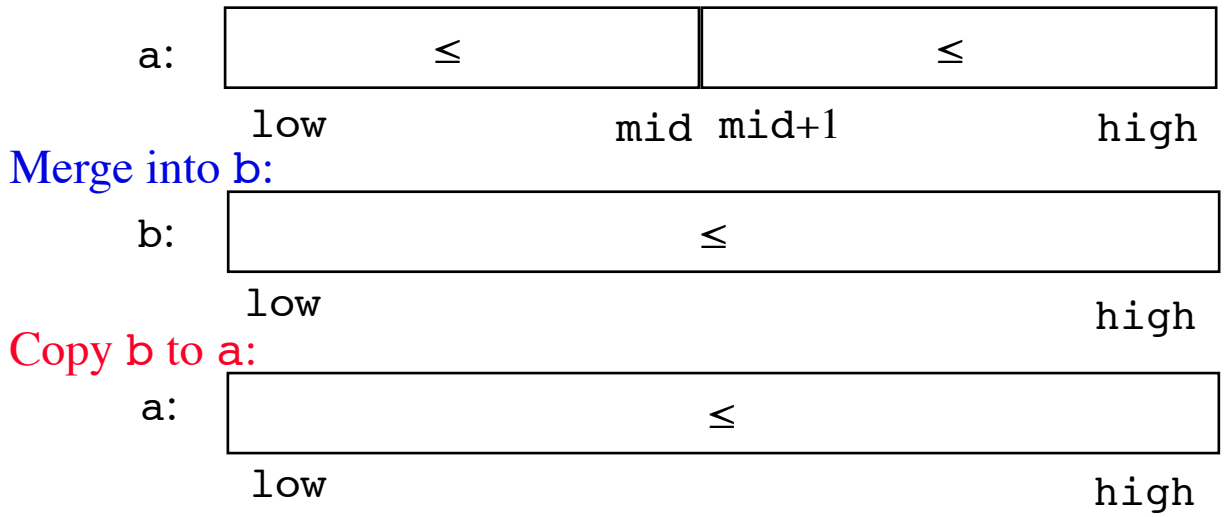


Linear-time merging of two sorted arrays




```
for (i = j = k = 0; k < M + N; k++)
    if (i == M) c[k] = b[j++]; else
    if (j == N) c[k] = a[i++]; else
    c[k] = a[i].compareTo(b[j]) < 0 ? a[i++] : b[j++];
```

Merging a[low:mid] with a[mid+1:high] to a[low:high]:




```
void merge(Comparable[] a, int low, int mid, int high) {
    int i = low, j = mid + 1;
    for (int k = low; k <= high; k++)
        if (i > mid)  b[k] = a[j++]; else
        if (j > high) b[k] = a[i++]; else
        b[k] = a[i].compareTo(a[j]) < 0 ? a[i++] : a[j++];
    for (int k = low; k <= high; k++) a[k] = b[k];
}
```

```
1       /**
2        * Mergesort algorithm.
3        * @param a an array of Comparable items.
4        */
5       public static <AnyType extends Comparable<? super AnyType>>
6       void mergeSort( AnyType [ ] a )
7       {
8           AnyType [ ] tmpArray = (AnyType []) new Comparable[ a.length ];
9           mergeSort( a, tmpArray, 0, a.length - 1 );
10      }
11
12      /**
13       * Internal method that makes recursive calls.
14       * @param a an array of Comparable items.
15       * @param tmpArray an array to place the merged result.
16       * @param left the left-most index of the subarray.
17       * @param right the right-most index of the subarray.
18       */
19      private static <AnyType extends Comparable<? super AnyType>>
20      void mergeSort( AnyType [ ] a, AnyType [ ] tmpArray,
21                      int left, int right )
22      {
23          if( left < right )
24          {
25              int center = ( left + right ) / 2;
26              mergeSort( a, tmpArray, left, center );
27              mergeSort( a, tmpArray, center + 1, right );
28              merge( a, tmpArray, left, center + 1, right );
29          }
30      }
```

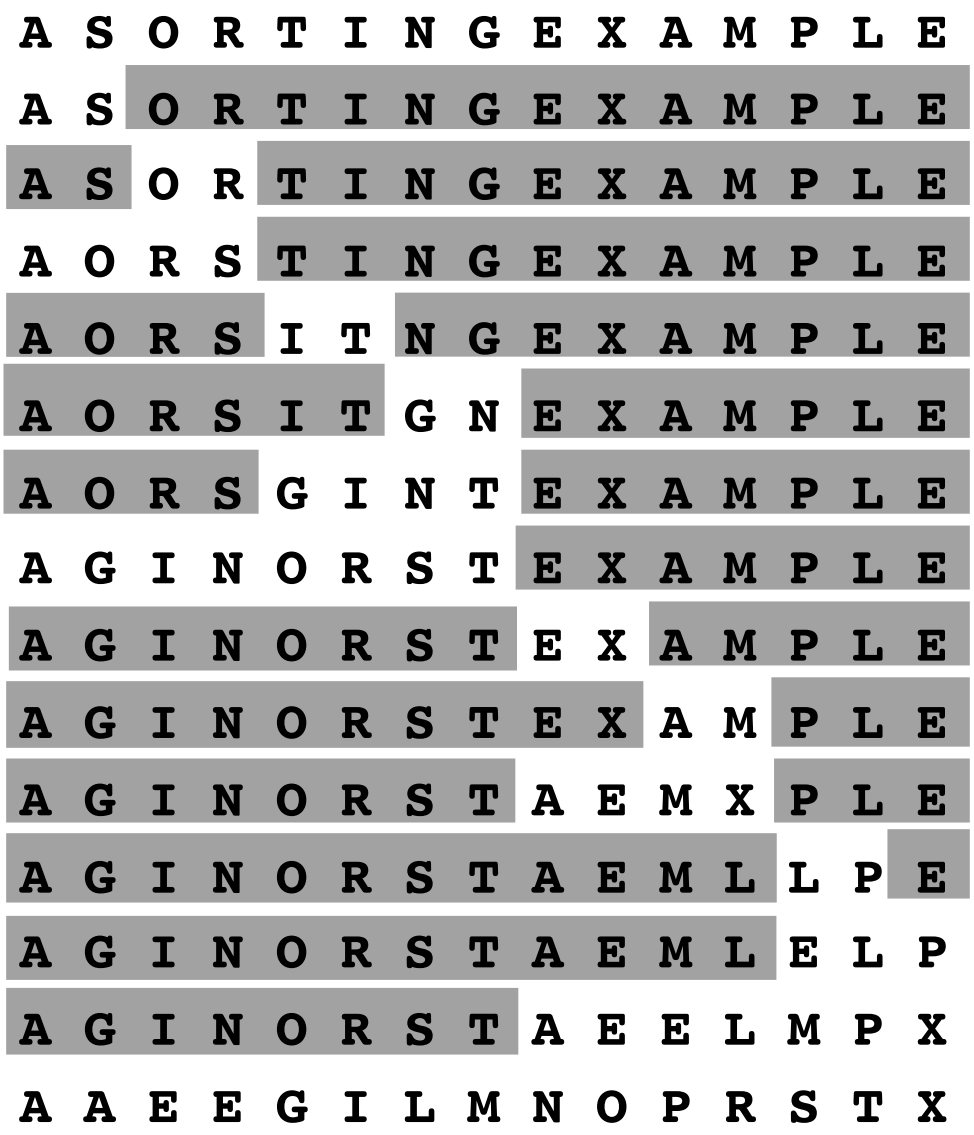
figure 8.8

Basic `mergeSort` routines

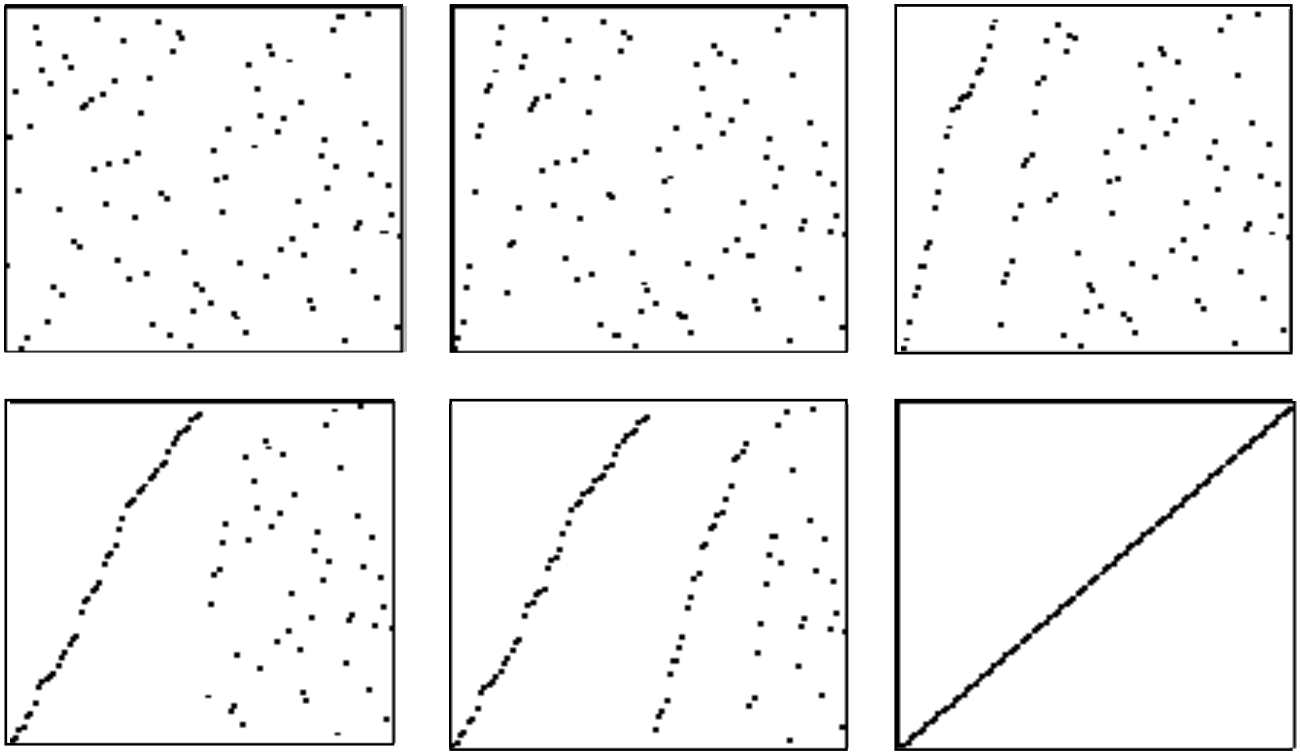
```
1     /**
2      * Internal method that merges two sorted halves of a subarray.
3      * @param a an array of Comparable items.
4      * @param tmpArray an array to place the merged result.
5      * @param leftPos the left-most index of the subarray.
6      * @param rightPos the index of the start of the second half.
7      * @param rightEnd the right-most index of the subarray.
8      */
9     private static <AnyType extends Comparable<? super AnyType>>
10    void merge( AnyType [ ] a, AnyType [ ] tmpArray,
11                int leftPos, int rightPos, int rightEnd )
12    {
13        int leftEnd = rightPos - 1;
14        int tmpPos = leftPos;
15        int numElements = rightEnd - leftPos + 1;
16
17        // Main loop
18        while( leftPos <= leftEnd && rightPos <= rightEnd )
19            if( a[ leftPos ].compareTo( a[ rightPos ] ) <= 0 )
20                tmpArray[ tmpPos++ ] = a[ leftPos++ ];
21            else
22                tmpArray[ tmpPos++ ] = a[ rightPos++ ];
23
24        while( leftPos <= leftEnd )    // Copy rest of first half
25            tmpArray[ tmpPos++ ] = a[ leftPos++ ];
26
27        while( rightPos <= rightEnd )  // Copy rest of right half
28            tmpArray[ tmpPos++ ] = a[ rightPos++ ];
29
30        // Copy tmpArray back
31        for( int i = 0; i < numElements; i++, rightEnd-- )
32            a[ rightEnd ] = tmpArray[ rightEnd ];
33    }
```

figure 8.9

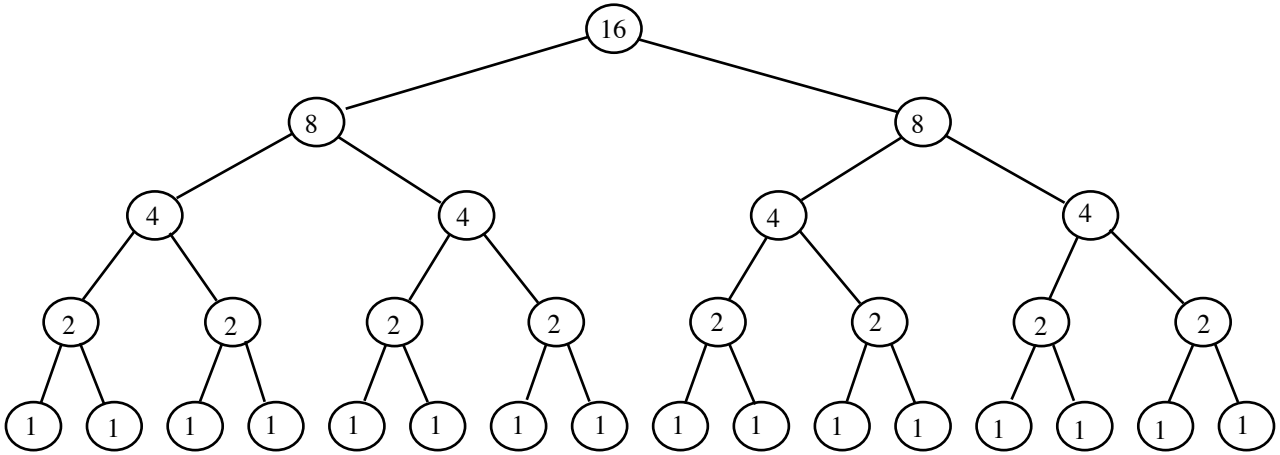
The merge routine

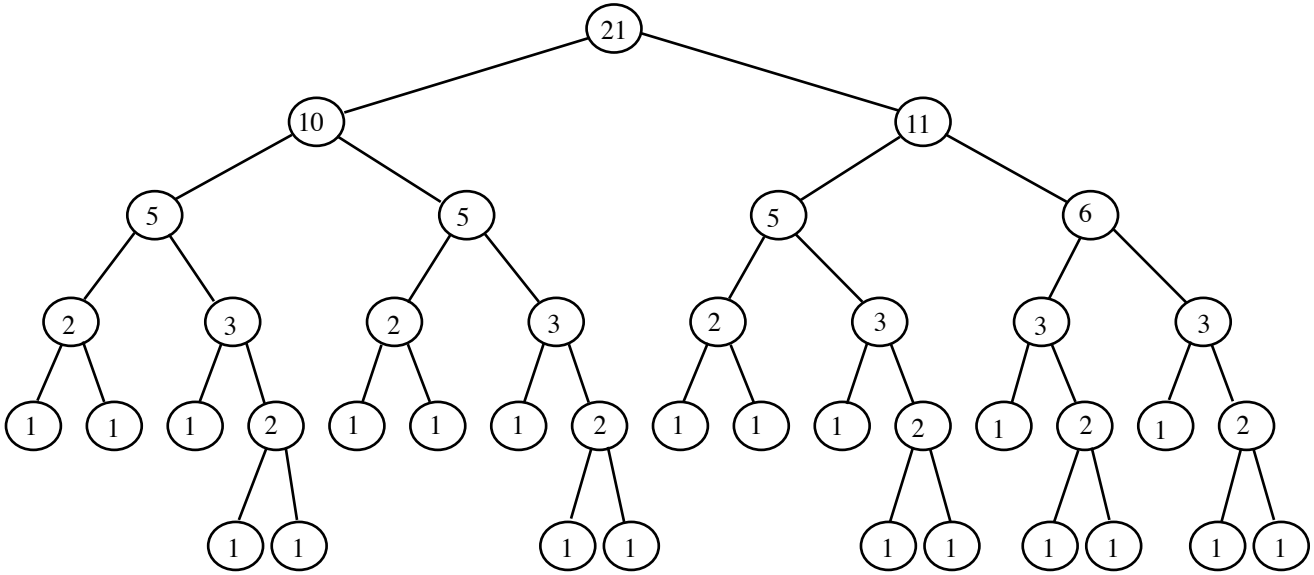

A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A S O R T I N G E X A M P L E
A O R S T I N G E X A M P L E
A O R S I T N G E X A M P L E
A O R S I T G N E X A M P L E
A O R S G I N T E X A M P L E
A G I N O R S T E X A M P L E
A G I N O R S T E X A M P L E
A G I N O R S T E X A M P L E
A G I N O R S T A E M X P L E
A G I N O R S T A E M L L P E
A G I N O R S T A E M L E L P
A G I N O R S T A E E L M P X
A A E E G I L M N O P R S T X

Animation of Mergesort

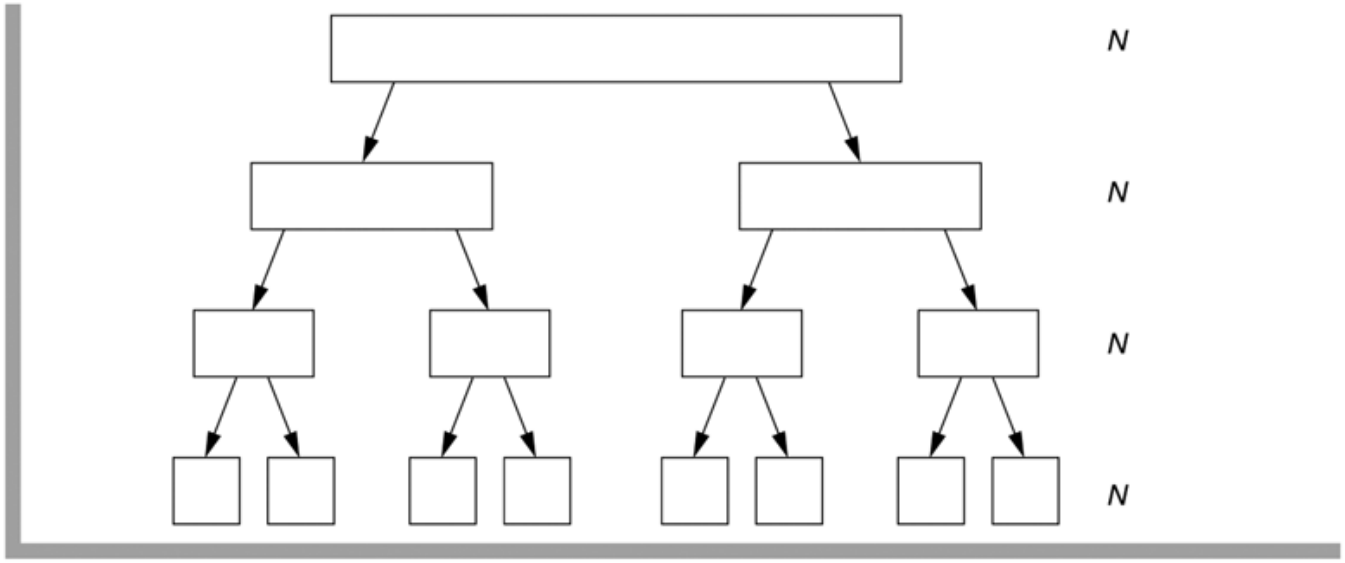


Call trees


16
8
8
4
4
4
4
2
2
2
2
2
2
2
2
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1
1


21
10
11
5
5
5
6
2
3
2
3
2
3
3
3
1
1
1
2
1
1
1
2
1
1
1
2
1
2
1
2
1
1
1
1
1
1
1
1
1
1

Call tree



There are about $\log_2 N$ levels
Each level uses $O(N)$ time

Evaluation of Mergesort

Advantages:

- insensitive to the input order
- requires about $N \log_2 N$ comparisons for *any* array of N elements

  $$C(N) = 2C(N/2) + N, C(1) = 0$$

- is *stable* (preserves the input order of equal elements)
- can be used for sorting linked lists
- is suitable for *external* sorting

Disadvantage:

- requires (in practice) extra memory proportional to N

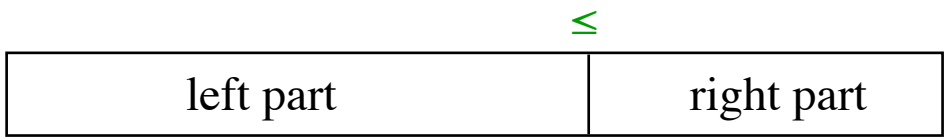
Quicksort

C. A. R. Hoare, 1962

Quicksort is in practice the fastest comparison-based algorithm for sorting arrays.

Idea:

To sort an array, partition it into a left and a right part, such that all elements in the left part are less than or equal to all elements in the right part.



Then, recursively sort the left part and the right part.

Basic Quicksort

The basic algorithm *Quicksort*(S) consists of the following four steps:

1. If the number of elements in S is 0 or 1, then return.

2. Pick *any* element v in S. It is called the *pivot*.

3. *Partition* $S - \{v\}$ (the remaining elements in S) into two disjoint groups $L = \{x \in S - \{v\} \mid x \leq v\}$ and $R = \{x \in S - \{v\} \mid x \geq v\}$.

4. Return the result of *Quicksort*(L) followed by v followed by *Quicksort*(R).

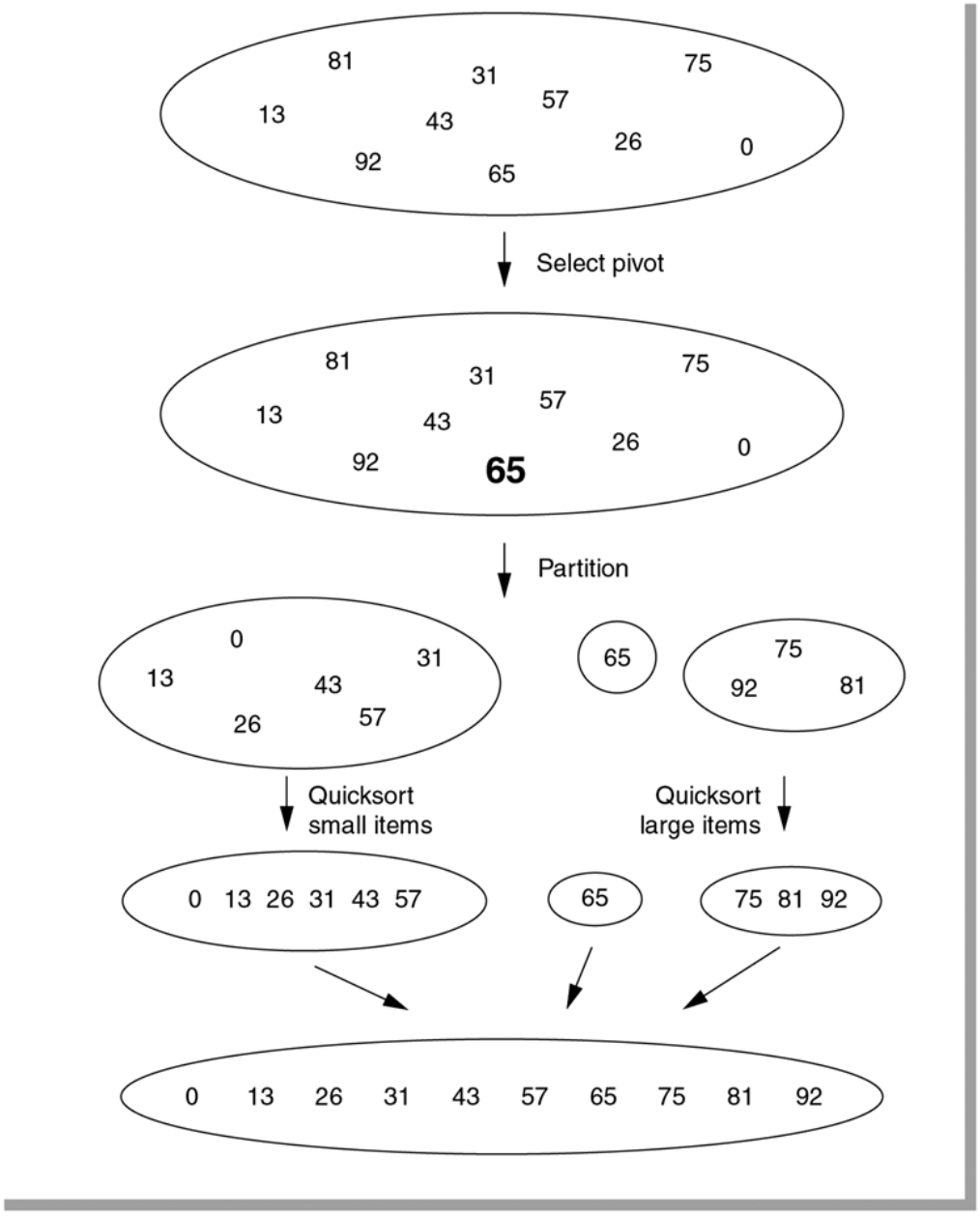


figure 8.10

The steps of quicksort

Partitioning

The partitioning of an array a may be performed as follows:

(1) Choose a pivot v among the elements in a.
(2) Traverse a from left to right until an element $a[i] \geq v$ is met.
(3) Traverse a from right to left until an element $a[j] \leq v$ is met.
(4) Swap $a[i]$ and $a[j]$.
(5) Continue traversing and swapping until the two traversals "cross".

Implementation

Partitioning `a[low:high]` about the pivot `v`:

```
i = low; j = high;
while (i <= j) {
    while (a[i].compareTo(v) < 0) i++;
    while (a[j].compareTo(v) > 0) j--;
    if (i <= j)
      { swap(a, i, j); i++; j--; }
}
```

Result: `a[low:i]` ≤ `a[j:high]` and `i` > `j`.

May be proven by showing that

`a[low:i-1]` ≤ `v` ≤ `a[j+1:high]`

is invariant for the outermost loop.

Swap

```
void swap(Object[] a, int i, int j) {
    Object temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Only the object references are swapped (not the objects themselves)

Quicksort

The pivot `v` may be *any* element in `a[low:high]`, for instance `a[(low + high)/2]`.

```
void quicksort(Comparable[] a, int low, int high) {
     if (low < high) {
          Comparable v = a[(low + high)/2];
          int i = low, j = high;
          while (true) {
               while (a[i].compareTo(v) < 0) i++;
               while (a[j].compareTo(v) > 0) j--;
               if (i >= j) break;
               swap(a, i++, j--);
          }
          quicksort(a, low, j);
          quicksort(a, i, high);
     }
 }
```

Alternative partitioning algorithm

Choose `a[high]` as pivot, partition `a[low:high-1]`, and swap `a[high]` with `a[i]`.

```
int partition(Comparable[] a, int low, int high) {
    Comparable v = a[high];
    int i = low - 1, j = high;
    while (true) {
        while (a[++i].compareTo(v) < 0) ;
        while (v.compareTo(a[--j]) < 0)
            if (j == low) break;
        if (i >= j) break;
        swap(a, i, j);
    }
    swap(a, i, high);
    return i;
}
```

Picking the pivot

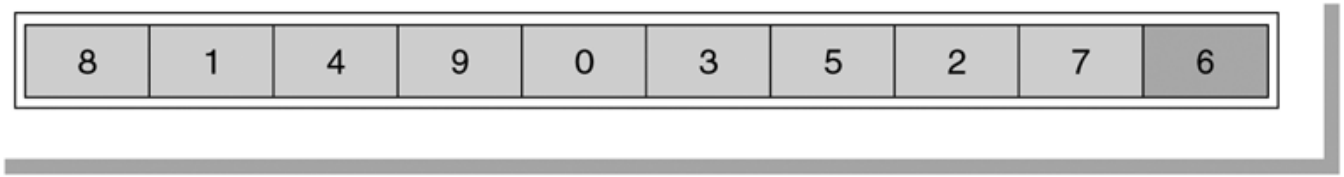


figure 8.11

Partitioning algorithm: Pivot element 6 is placed at the end.

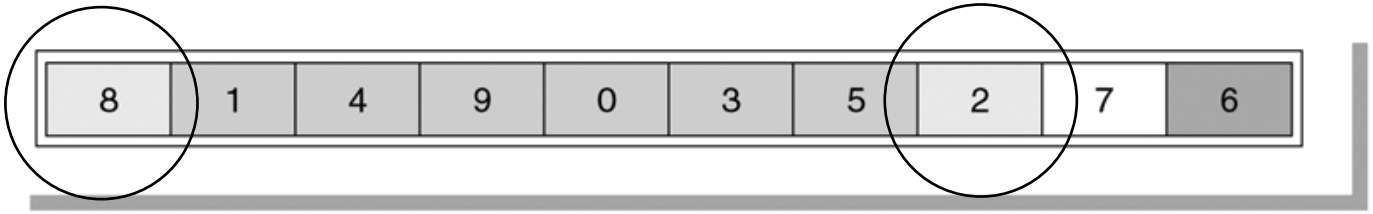


figure 8.12

Partitioning algorithm: i stops at large element 8; j stops at small element 2.

figure 8.13

Partitioning algorithm: The out-of-order elements 8 and 2 are swapped.

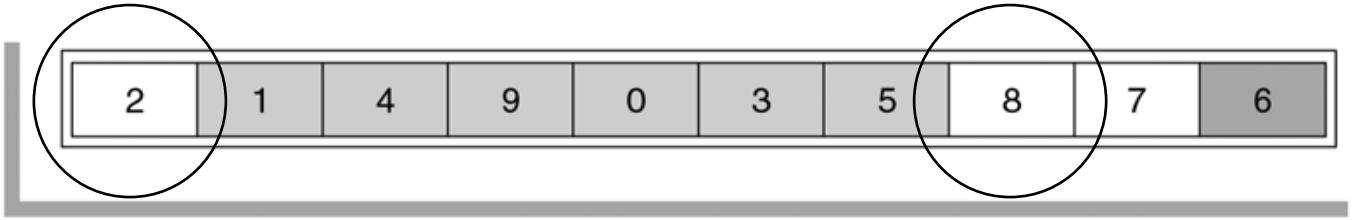


figure 8.14

Partitioning algorithm: i stops at large element 9; j stops at small element 5.

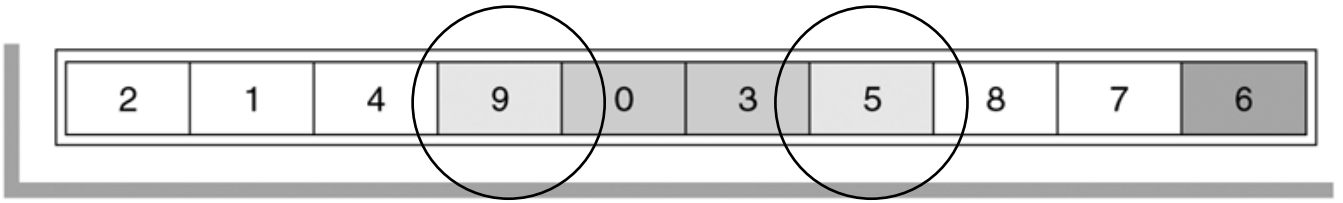


figure 8.15

Partitioning algorithm: The out-of-order elements 9 and 5 are swapped.

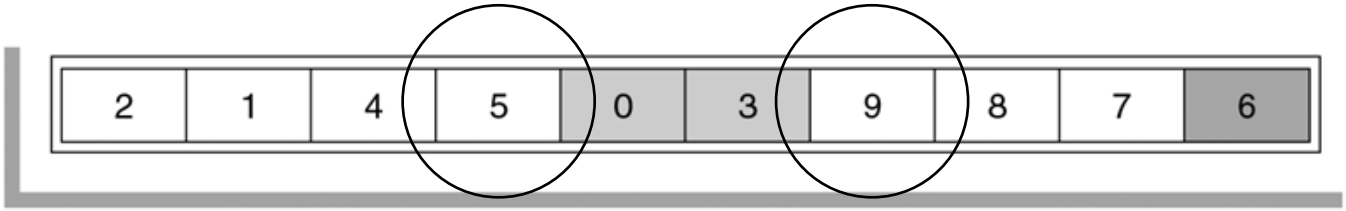


figure 8.16

Partitioning algorithm: `i` stops at large element 9; `j` stops at small element 3.

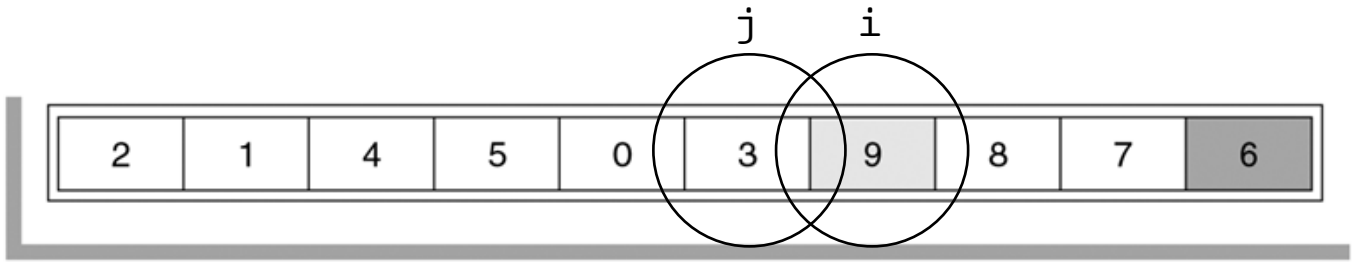
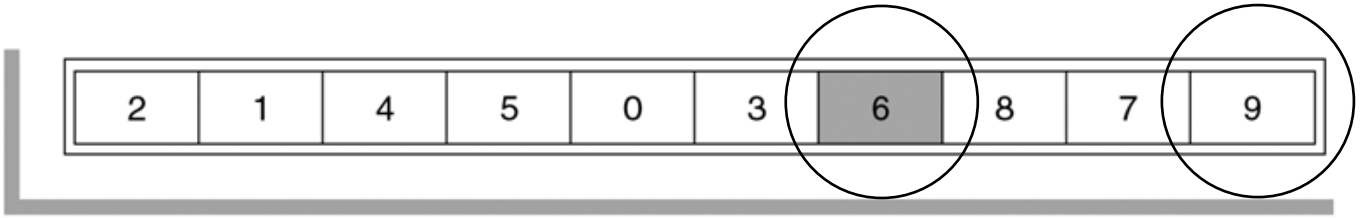


figure 8.17

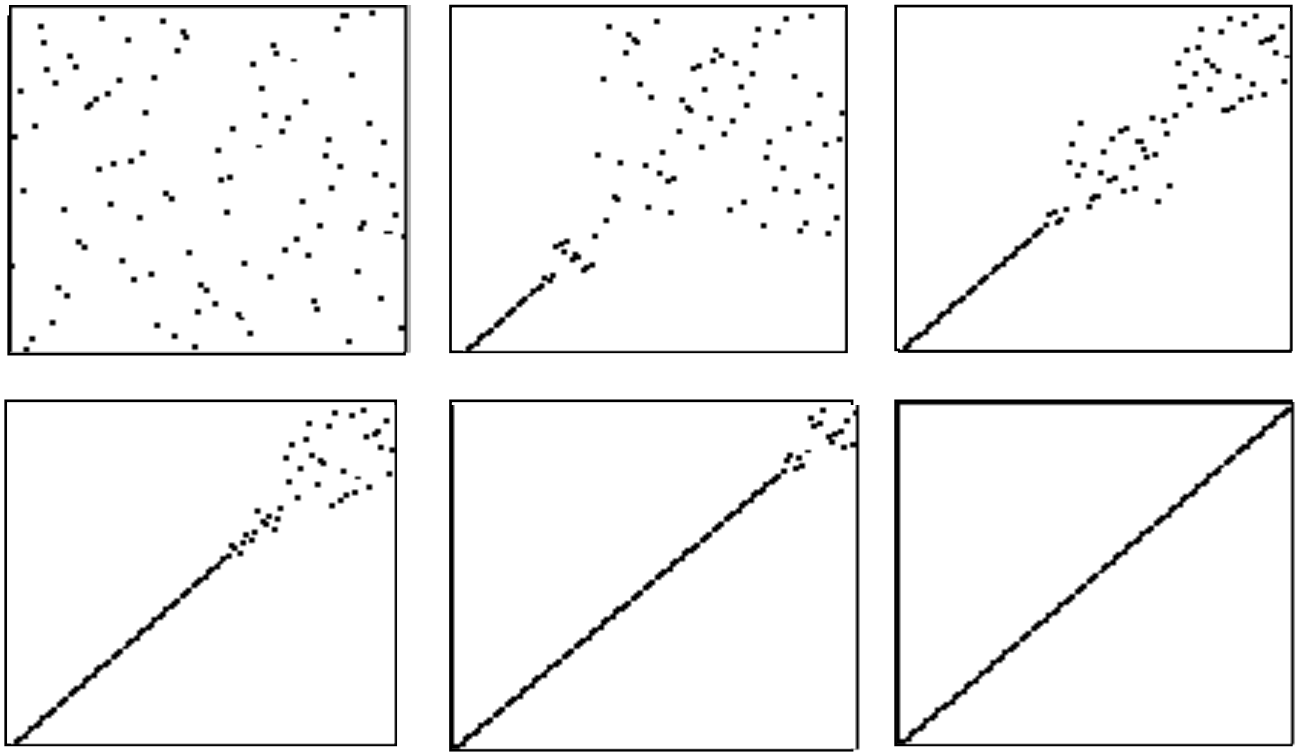
Partitioning algorithm: Swap pivot and element in position `i`.



The method `quicksort`

```
int quicksort(Comparable[] a, int low, int high) {
    if (low < high) {
        int i = partition(a, low, high);
        quicksort(a, low, i - 1);
        quicksort(a, i + 1, high);
    }
}
```

Animation of Quicksort



Number of comparisons

Let $C(N)$ denote the number of comparisons required for executing Quicksort on an array of N elements.

Partitioning requires N comparisons. Next, the left part and the right part are sorted separately.

On average each part consists of about $N/2$ elements. Then we obtain the recurrence

$$C(N) = N + 2C(N/2) \text{ for } N \geq 2,$$
$$C(0) = C(1) = 0,$$

which has the solution $C(N) = N \log_2 N$.

Average number of comparisons

More precise computations give

Quicksort uses about $2N \ln N$ comparisons on the average

where ln denotes the natural logarithm.

$$2N \ln N \approx 1.39\, N \log_2 N$$

So the average number of comparisons is only about 39% higher than in the best case.

Number of comparisons in the worst case

The worst case occurs when the partitioning for each N results in a part of one element and a part of N-1 elements.

In this case we have the recurrence

$$C(N) = N + C(N\text{-}1) \text{ for } N \geq 2,$$
$$C(0) = C(1) = 0,$$

which has the solution $C(N) = N(N+1)/2$.

Picking the pivot at random or by the "median-of-three" method makes the worst case unlikely to occur.

The median of N numbers is the $\lceil N/2 \rceil$th smallest number.

Median-of-three partitioning

figure 8.18

Original array

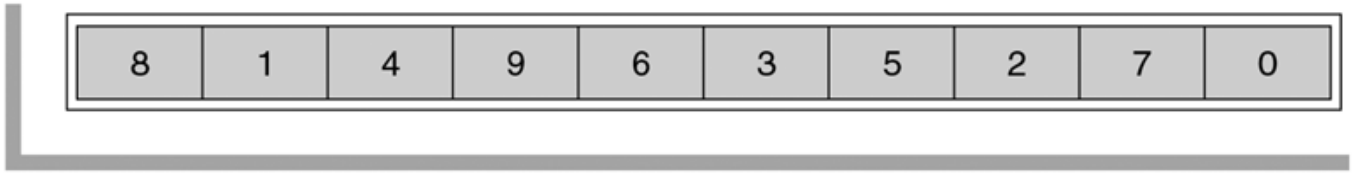


figure 8.19

Result of sorting three elements (first, middle, and last)

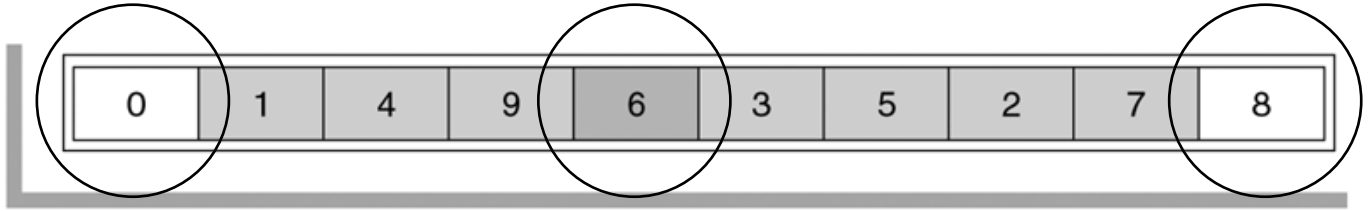
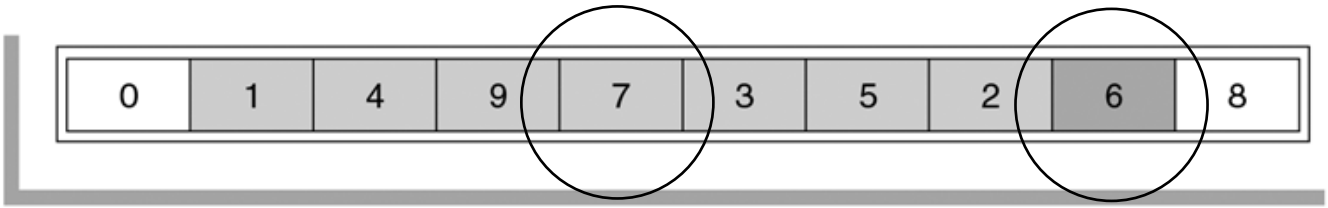


figure 8.20

Result of swapping the pivot with the next-to-last element



Quicksort with median-of-three partitioning

```
void quicksort(Comparable[] a, int low, int high) {
    if (low < high) {
        if (a[high].compareTo(a[low]) < 0)
            swap(a, low, high);
        int mid = (low + high) / 2;
        if (mid == low) return;
        if (a[mid].compareTo(a[low]) < 0)
            swap(a, low, mid);
        if (a[high].compareTo(a[mid]) < 0)
            swap(a, mid, high);
        swap(a, mid, high - 1);
        int i = partition(a, low + 1, high - 1);
        quicksort(a, low, i - 1);
        quicksort(a, i + 1, high);
    }
}
```

Small array segments

Use a simple method for sorting small array segments.

The test in the beginning of `quicksort`:

```
if (low < high)
```

is replaced by

```
if (high - low < CUTOFF)
    insertionSort(a, low, high);
else
```

where `CUTOFF` is in the range from 5 to 25.

figure 8.21

Quicksort with median-of-three partitioning and cutoff for small arrays

```
1     /**
2      * Quicksort algorithm (driver)
3      */
4     public static <AnyType extends Comparable<? super AnyType>>
5     void quicksort( AnyType [ ] a )
6     {
7         quicksort( a, 0, a.length - 1 );
8     }
9
10    /**
11     * Internal quicksort method that makes recursive calls.
12     * Uses median-of-three partitioning and a cutoff.
13     */
14    private static <AnyType extends Comparable<? super AnyType>>
15    void quicksort( AnyType [ ] a, int low, int high )
16    {
17        if( low + CUTOFF > high )
18            insertionSort( a, low, high );
19        else
20        {      // Sort low, middle, high
21            int middle = ( low + high ) / 2;
22            if( a[ middle ].compareTo( a[ low ] ) < 0 )
23                swapReferences( a, low, middle );
24            if( a[ high ].compareTo( a[ low ] ) < 0 )
25                swapReferences( a, low, high );
26            if( a[ high ].compareTo( a[ middle ] ) < 0 )
27                swapReferences( a, middle, high );
28
29                // Place pivot at position high - 1
30            swapReferences( a, middle, high - 1 );
31            AnyType pivot = a[ high - 1 ];
32
33                // Begin partitioning
34            int i, j;
35            for( i = low, j = high - 1; ; )
36            {
37                while( a[ ++i ].compareTo( pivot ) < 0 )
38                    ;
39                while( pivot.compareTo( a[ --j ] ) < 0 )
40                    ;
41                if( i >= j )
42                    break;
43                swapReferences( a, i, j );
44            }
45                // Restore pivot
46            swapReferences( a, i, high - 1 );
47
48            quicksort( a, low, i - 1 );    // Sort small elements
49            quicksort( a, i + 1, high );   // Sort large elements
50        }
51    }
```

Memory usage

The *average* extra space complexity of quicksort is $O(\log_2 N)$.
This extra space comes from the call stack. Each recursive call will create a stack frame.

The *worst case* extra space complexity for a naive implementation is $O(N)$. The worst case occurs when the partitioning for each N results in a part of one element and a part of N-1 elements, and we quicksort the largest part first.

Extra space complexity of $O(\log_2 N)$ is guaranteed if we always sort the array segment with the fewest elements first, and sort the other array segment using iteration (that is, eliminates the *tail recursion*).

Elimination of tail recursion

A subroutine is said to be *tail-recursive*, if it calls itself as its final action.

Tail recursion can always be replaced by *iteration*.

Recursion (call: `p(a)`)

```
void p(type x) {
    if (b(x))
        S1;
    else {
        S2;
        p(f(x));
    }
}
```

Iteration

```
type x = a;
while (!b(x)) {
    S2;
    x = f(x);
}
S1;
```

```
int quicksort(Comparable[] a, int low, int high) {
    if (low < high) {
        int i = partition(a, low, high);
        quicksort(a, low, i - 1);
        quicksort(a, i + 1, high);
    }
}
```

```
int quicksort(Comparable[] a, int low, int high) {
    while (low < high) {
        int i = partition(a, low, high);
        quicksort(a, low, i - 1);
        low = i + 1;
   }
}
```

Smart compilers (but not Java) can detect tail recursion and convert it to iteration in order to optimize code

Selection

Problem: Find the k'th smallest element in an array of N elements

Example: The 3'rd smallest element in $\{3, 6, 5, 2, 8, 4\}$ is 4.

Solution 1: Sort the array in increasing order.
The k'th element in the sorted array is the solution to the problem.

Complexity: Depends on the sorting algorithm -
using Mergesort: $O(N \log N)$.

Can we do it faster?

Selection by means of `partition`

```
void quickSelect(Comparable[] a, int low, int high, int k) {
    if (low < high) {
        int i = partition(a, low, high);
        if (k <= i) quickSelect(a, low, i - 1, k);
        else if (k > i + 1) quickSelect(a, i + 1, high, k);
    }
}
```

Since only *tail recursion* is used, the recursion may be eliminated:

```
void quickSelect(Comparable a[], int low, int high, int k) {
    while (low < high) {
        int i = partition(a, low, high);
        if (k <= i) high = i - 1;
        else if (k >= i + 1) low = i + 1;
    }
}
```

```
1     /**
2      * Internal selection method that makes recursive calls.
3      * Uses median-of-three partitioning and a cutoff.
4      * Places the kth smallest item in a[k-1].
5      * @param a an array of Comparable items.
6      * @param low the left-most index of the subarray.
7      * @param high the right-most index of the subarray.
8      * @param k the desired rank (1 is minimum) in the entire array.
9      */
10    private static <AnyType extends Comparable<? super AnyType>>
11    void quickSelect( AnyType [ ] a, int low, int high, int k )
12    {
13        if( low + CUTOFF > high )
14            insertionSort( a, low, high );
15        else
16        {
17                // Sort low, middle, high
18            int middle = ( low + high ) / 2;
19            if( a[ middle ].compareTo( a[ low ] ) < 0 )
20                swapReferences( a, low, middle );
21            if( a[ high ].compareTo( a[ low ] ) < 0 )
22                swapReferences( a, low, high );
23            if( a[ high ].compareTo( a[ middle ] ) < 0 )
24                swapReferences( a, middle, high );
25
26                // Place pivot at position high - 1
27            swapReferences( a, middle, high - 1 );
28            AnyType pivot = a[ high - 1 ];
29
30                // Begin partitioning
31            int i, j;
32            for( i = low, j = high - 1; ; )
33            {
34                while( a[ ++i ].compareTo( pivot ) < 0 )
35                    ;
36                while( pivot.compareTo( a[ --j ] ) < 0 )
37                    ;
38                if( i >= j )
39                    break;
40                swapReferences( a, i, j );
41            }
42                // Restore pivot
43            swapReferences( a, i, high - 1 );
44
45                // Recurse; only this part changes
46            if( k <= i )
47                quickSelect( a, low, i - 1, k );
48            else if( k > i + 1 )
49                quickSelect( a, i + 1, high, k );
50        }
51    }
```

figure 8.22

Quickselect with median-of-three partitioning and cutoff for small arrays

Complexity of `quickSelect`

$O(N)$ on the average

since $N + N/2 + N/4 + N/8 + \ldots \leq 2N$.

It is possible (but not quite easy) to achieve *guaranteed* linear running time.

A lower bound for sorting

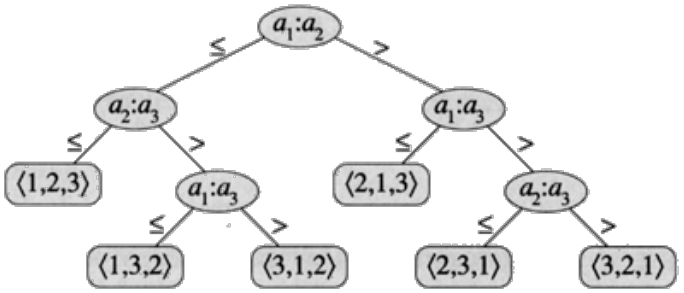
Any sorting algorithm that sorts *using comparisons* must use at least $\lceil \log(N!) \rceil$ comparisons for some input sequence.

Informal proof:
Sorting is equivalent to finding a permutation of the input sequence. Thus, sorting may be modeled by a *decision tree*, where each internal node corresponds to a comparison, and each external node corresponds to one of the $N!$ possible permutations. The height of the tree must at least be $\log_2(N!)$.

How large is $\lceil \log_2(N!) \rceil$?

From Stirling's formula $N! \approx \sqrt{2\pi N}(\frac{N}{e})^N$ we get

$$\log_2(N!) \approx N\log_2 N - 1.44N.$$



Counting sort

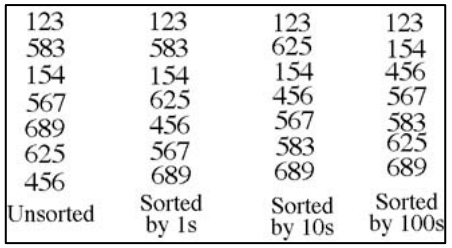
A linear sorting algorithm

Sort an array `a` of non-negative integers less than `m`.

```
void sort(int[] a, int m) {
    int[] count = new int[m];
    for (int i = 0; i < a.length; i++)
        count[a[i]]++;
    for (int i = 0, j = 0; j < m; j++)
        for (int k = count[j]; k > 0; k--)
            a[i++] = j;
}
```

Radix sort

A linear sorting algorithm



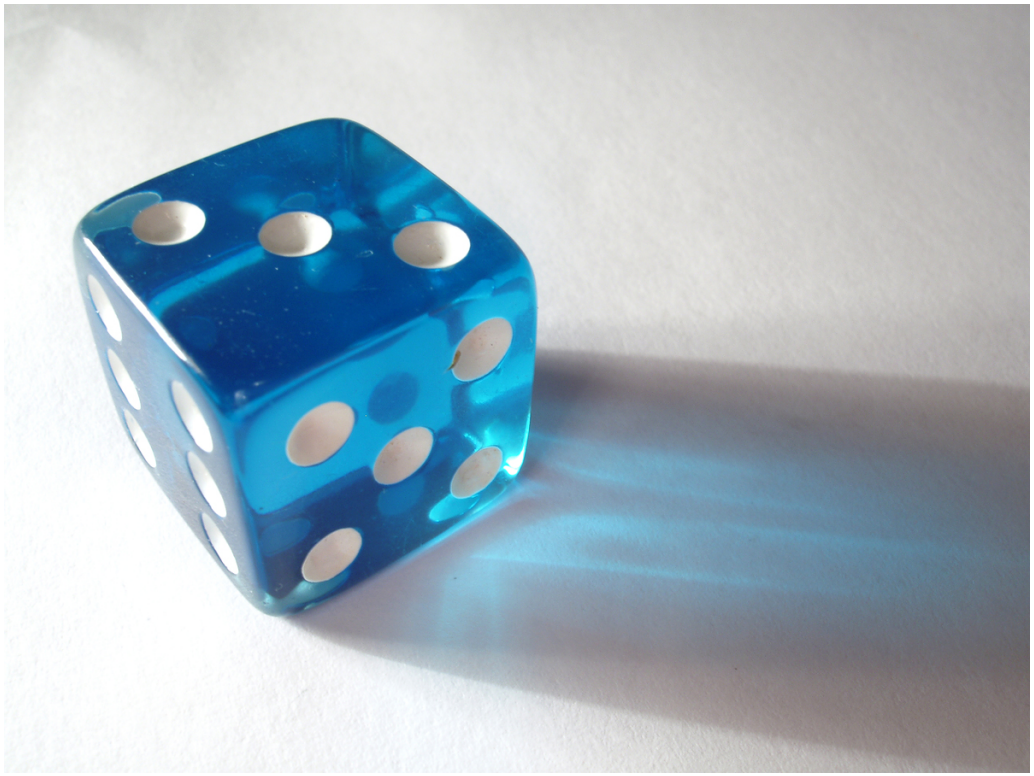
Sort an array `a` of non-negative `d`-digit integers in radix (base) `r`.

```
void sort(int[] a, int d, int r) {
    int[] count = new int[r];
    int[] b = new int[a.length];
    for (int pass = 0, pow = 1; pass < d; pass++, pow *= r) {
        for (int i = 0; i < r; i++) count[i] = 0;
        for (int i = 0; i < a.length; i++)
            count[a[i] / pow % r]++;
        for (int i = 1; i < r; i++)
            count[i] += count[i - 1];
        for (int i = a.length - 1; i >= 0; i--)
            b[--count[a[i] / pow % r]] = a[i];
        for (int i = 0; i < a.length; i++) a[i] = b[i];
    }
}
```

Radix-10 sort

31	41	59	26	53	58	97	23	93	84	`a`	unsorted
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*		
0	2	0	3	1	0	1	1	1	1	`count`	
0	2	2	5	6	6	7	8	9	10	`count`	
31	41	23	53	93	84	26	97	58	59	a	sorted by 1s
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*		
0	0	2	1	1	3	0	0	1	2	`count`	
0	0	2	3	4	7	7	7	8	10	`count`	
23	26	31	41	53	58	59	84	93	97	`a`	sorted by 10s (and 1s)
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*		

Randomization



Why do we need random numbers?

Many applications:

- Program testing (generation of random input data)
- Sorting (e.g., determination of the pivot in Quicksort)
- Simulation (e.g., generation of arrival times for bank customers)
- Games (e.g., choice of opening moves in chess)
- Randomized algorithms (e.g., primality testing)

Generation of random numbers

True randomness in a computer is impossible to achieve.

Generally, it is sufficient to produce **pseudorandom numbers**, or numbers that *appear* to be random because they satisfy many of the properties of random numbers.

A pseudorandom number generator must pass a number of statistical tests. One such generator is the *linear congruential generator* (Lehmer, 1951) in which numbers X_1, X_2, ... are generated that satisfy

$$X_{i+1} = AX_i + C \pmod{M}$$

The initial value X_0 is called the **seed**.

A, C, and M must be chosen in such a way that the length of the sequence until a number is repeated (the **period**) becomes as large as possible. This happens when M is a large prime number.

Generation of random numbers in Java

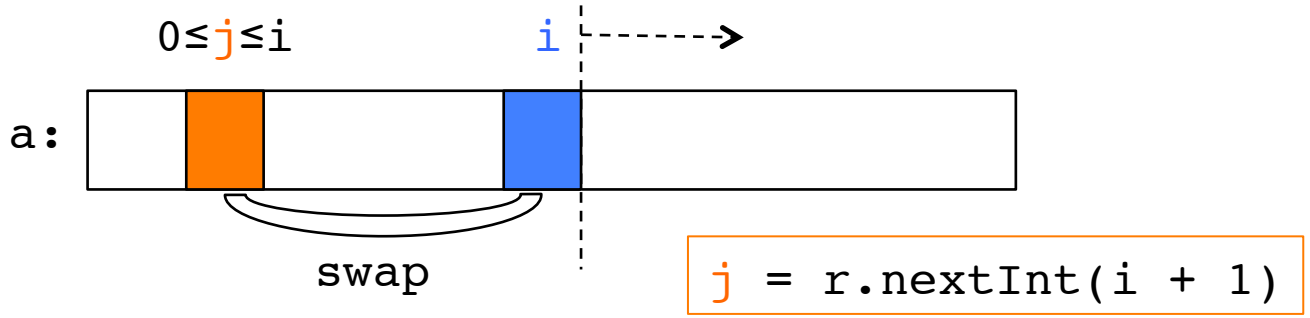
Java provides the class java.util.Random.

```
public class Random {
    public Random();
    public Random(long seed);
    public int nextInt();
    public int nextInt(int n);
    public long nextLong();
    public float nextFloat();
    public double nextDouble();
    public double nextBytes(byte[] bytes);
    public double nextGaussian();
    public void setSeed(long seed);
    protected int next(int bits);
}
```

Implementation of Lehmer's method

```
public class Random {
    private long seed;
    private final static long multiplier = 0x5DEECE66DL;   // = 25214903917
    private final static long addend = 0xBL;               // = 11
    private final static long mask = (1L << 48) - 1;

    public Random() { this(System.currentTimeMillis()); }

    public Random(long seed) {
        this.seed = (seed ^ multiplier) & mask;
    }

    public int nextInt() { return next(32); }

    protected int next(int bits) {
        long nextseed = (seed * multiplier + addend) & mask;
        seed = nextseed;
        return (int) (nextseed >>> (48 - bits));
    }
}
```

Generation of random permutations




```
void permute(Object[] a) {
    Random r = new Random();
    for (int i = 1; i < a.length; i++)
        swap(a, i, r.nextInt(i + 1));
}
```

The algorithm runs in linear time.
The number of different possible outcomes $2 \cdot 3 \cdot \ldots \cdot N\text{-}1 \cdot N$ is equal to the number of possible permutations, $N!$.

Primality testing by trial division

figure 9.7

Primality testing by trial division

```
1     /**
2      * Returns true if odd integer n is prime.
3      */
4     public static boolean isPrime( long n )
5     {
6         for( int i = 3; i * i <= n; i += 2 )
7             if( n % i == 0 )
8                 return false; // not prime
9
10        return true;          // prime
11    }
```

Randomized primality testing

P. de Fermat, 1601-65

Fermat's little theorem:
If P is prime and $0 < A < P$, then $A^{P-1} = 1 (\text{mod } P)$

If for a number N we can find a value $0 < A < N$
such that $A^{N-1} (\text{mod } N)$ is not 1, then N is not a prime.
A is said to be a *witness* to N's compositeness.

Finding a witness

Every composite number has a witness. But for some numbers it can be hard to find. The following theorem can be used to improve our chances of finding a witness.

Theorem:
If P is prime and $X^2 = 1 (\text{mod } P)$, then $X = \pm 1 (\text{mod } P)$

Corollary:
If $X^2 = 1 (\text{mod } N)$ and $X \neq \pm 1 (\text{mod } N)$, then N is not prime

Exponentiation

Efficient algorithm

If n is even, then

$$x^n = (x \cdot x)^{\frac{n}{2}}$$

If n is odd, then

$$x^n = x \cdot x^{n-1} = x \cdot (x \cdot x)^{\left\lfloor \frac{n}{2} \right\rfloor}$$

```
public static long power(long x, int n) {
    if (n == 0)
        return 1;
    long tmp = power(x * x, n / 2);
    if (n % 2 != 0)
        tmp *= x;
    return tmp;
}
```

Number of multiplications $< 2 \log_2 n$.

Modular exponentiation

```
1     /**
2      * Return x^n (mod p)
3      * Assumes x, n >= 0, p > 0, x < p, 0^0 = 1
4      * Overflow may occur if p > 31 bits.
5      */
6     public static long power( long x, long n, long p )
7     {
8         if( n == 0 )
9             return 1;
10
11        long tmp = power( ( x * x ) % p, n / 2, p );
12
13        if( n % 2 != 0 )
14            tmp = ( tmp * x ) % p;
15
16        return tmp;
17    }
```

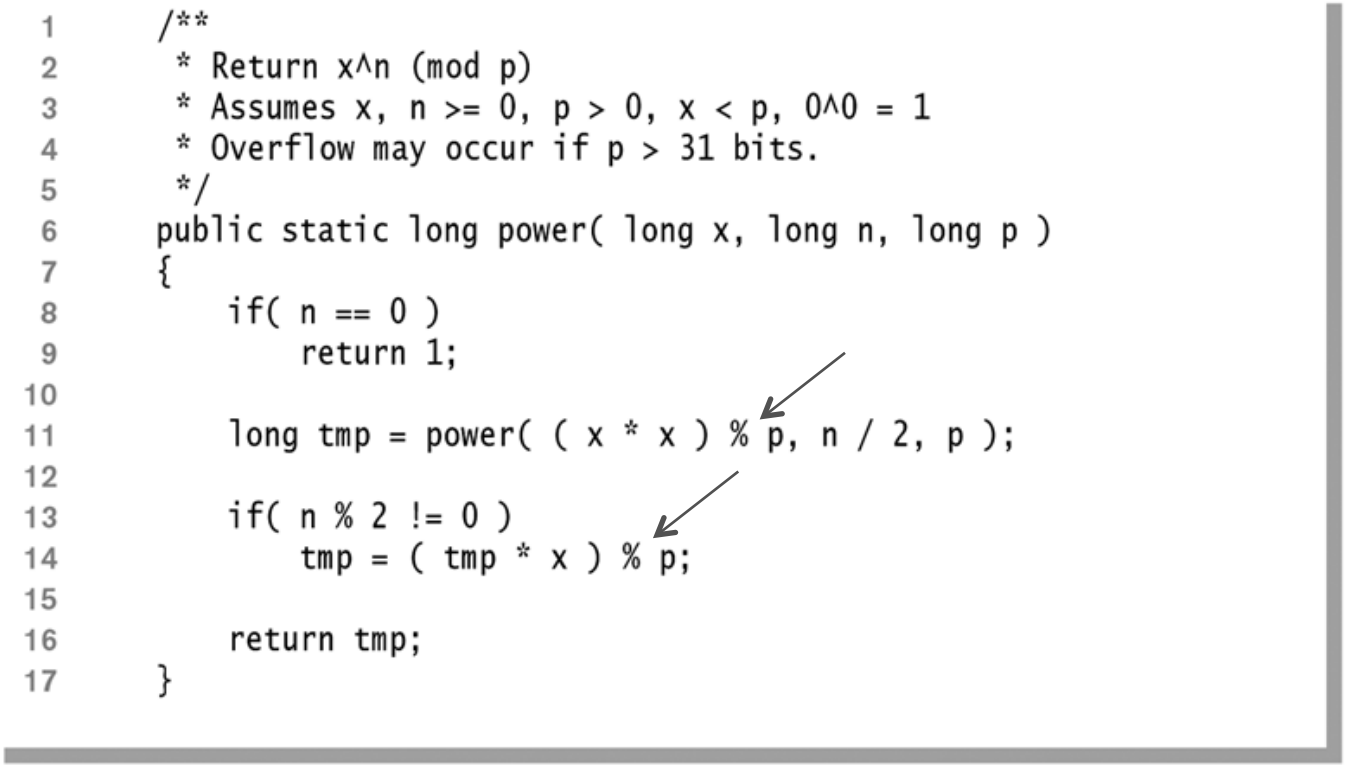


figure 7.16

Modular exponentiation routine

Modular exponentiation

```
/**
 * Return a^i (mod n)
 * Assumes a, i >= 0, n > 0, a < n, 0^0 = 1
 * Overflow may occur if n > 31 bits.
 */
 public static long power( long a, long i, long n )
 {
    if( i == 0 )
       return 1;

    long tmp = power( ( a * a ) % n, i / 2, n );

    if( i % 2 != 0 )
       tmp = ( tmp * a ) % n;

    return tmp;
}
```

```
1       /**
2        * Private method that implements the basic primality test.
3        * If witness does not return 1, n is definitely composite.
4        * Do this by computing a^i (mod n) and looking for
5        * nontrivial square roots of 1 along the way.
6        */
7       private static long witness( long a, long i, long n )
8       {
9           if( i == 0 )
10              return 1;
11
12          long x = witness( a, i / 2, n );
13          if( x == 0 )    // If n is recursively composite, stop
14              return 0;
15
16          // n is not prime if we find a nontrivial square root of 1
17          long y = ( x * x ) % n;
18          if( y == 1 && x != 1 && x != n - 1 )
19              return 0;
20
21          if( i % 2 != 0 )
22              y = ( a * y ) % n;
23
24          return y;
25      }
```

```
26
27      /**
28       * The number of witnesses queried in randomized primality test.
29       */
30      public static final int TRIALS = 5;
31
32      /**
33       * Randomized primality test.
34       * Adjust TRIALS to increase confidence level.
35       * @param n the number to test.
36       * @return if false, n is definitely not prime.
37       *     If true, n is probably prime.
38       */
39      public static boolean isPrime( long n )
40      {
41          Random r = new Random( );
42
43          for( int counter = 0; counter < TRIALS; counter++ )
44              if( witness( r.nextInt( (int) n - 3 ) + 2, n - 1, n ) != 1 )
45                  return false;
46
47          return true;
48      }
```

figure 9.9

A randomized test for primality

Confidence of randomized primality testing

Some values of A will trick the algorithm into declaring that N is prime.

In fact, if we choose A randomly, we have at most ¼ chance of failing to detect a composite number.

However, if we independently use 20 values of A (`TRIALS` = 20), the chances that none of them will witness a composite number is $¼^{20}$, which is about 1 in a million million.