

Outline	
1 Partitioning	
2 Sorting	
3 An Algorithmic Improvement	
an Algorithmic Improvement	



Partitioning	
Quick sort works by partitioning the input array, about a pivot, into two subarrays, and then sorting the subarr independently	ays

Quick sort works by partitioning the input array, about a pivot, into two subarrays, and then sorting the subarrays independently $\frac{1}{2}$



Quick sort works by partitioning the input array, about a pivot, into two subarrays, and then sorting the subarrays independently $\frac{1}{2}$



Quick sort works by partitioning the input array, about a pivot, into two subarrays, and then sorting the subarrays independently $\frac{1}{2}$



Quick sort works by partitioning the input array, about a pivot, into two subarrays, and then sorting the subarrays independently $\frac{1}{2}$



Quick sort works by partitioning the input array, about a pivot, into two subarrays, and then sorting the subarrays independently $\frac{1}{2}$



Quick sort works by partitioning the input array, about a pivot, into two subarrays, and then sorting the subarrays independently $\frac{1}{2}$



Quick sort works by partitioning the input array, about a pivot, into two subarrays, and then sorting the subarrays independently $\frac{1}{2}$







The partitioning procedure rearranges the array such that:

 \bullet The entry ${\tt a[j]}$ is in its final place in the array, for some ${\tt j}$

The partitioning procedure rearranges the array such that:

- \bullet The entry ${\tt a[j]}$ is in its final place in the array, for some ${\tt j}$
- No entry in a[10] through a[j 1] is greater than a[j]

The partitioning procedure rearranges the array such that:

- ullet The entry a[j] is in its final place in the array, for some j
- No entry in a[10] through a[j-1] is greater than a[j]
- No entry in a[j + 1] through a[hi] is less than a[j]



		a[]															
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		K	R	A	Т	Е	L	Е	P	U	I	М	Q	С	Х	0	S



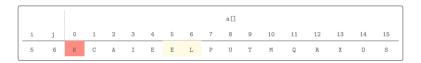


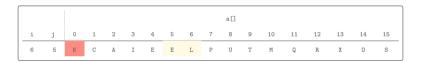














		a[]															
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	5	Е	C	Α	I	Е	K	L	P	U	Т	М	Q	R	Х	0	S

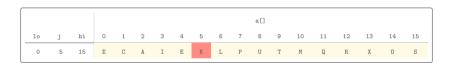


```
② Quick.java
public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        int i = lo;
        int i = hi + 1:
        Comparable v = a[lo];
        while (true) {
            while (less(a[++i], v)) {
                if (i == hi) {
                    break:
            while (less(v, a[--i])) {
                if (i == lo) {
                    break;
            if (i >= i) {
                break:
            exchange(a, i, j);
        exchange(a, lo, i):
        return i:
    private static int partition(Object[] a. int lo. int hi. Comparator c) {
        int i = lo;
        int i = hi + 1:
        Object v = a[lo]:
        while (true) {
            while (less(a[++i], v, c)) {
                if (i == hi) {
                    break:
            }
```



			аП															
lo	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			Q	U	I	С	K	S	0	R	Т	Е	Х	A	М	P	L	E











			аП															
10	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1		1	A	С	E	Е	I	K	L	Р	U	Т	М	Q	R	Х	0	S























			a[]															
10	j	hi	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
			A	С	Е	Е	I	K	L	M	0	P	Q	R	S	Т	U	х

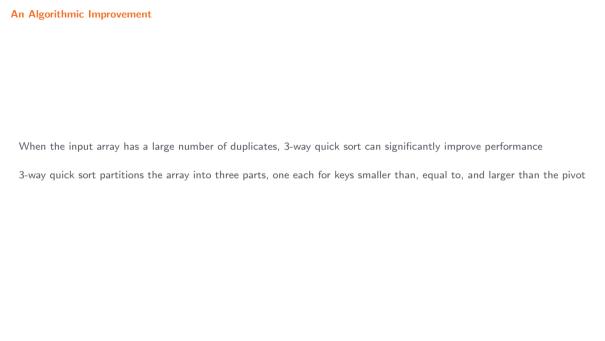


```
@ Quick.java
public class Quick {
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a):
        sort(a, 0, a.length - 1);
    public static void sort(Object[] a. Comparator c) {
        StdRandom.shuffle(a):
        sort(a, 0, a.length - 1, c);
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) {
            return;
        int | = partition(a, lo, hi);
        sort(a, lo, j - 1);
        sort(a, j + 1, hi);
    private static void sort(Object[] a, int lo, int hi, Comparator c) {
        if (hi <= lo) {
            return:
        int i = partition(a, lo, hi, c);
        sort(a, lo, i - 1, c):
        sort(a, i + 1, hi, c);
```

```
🗷 Quick.java
public class Quick {
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a):
        sort(a, 0, a.length - 1);
    public static void sort(Object[] a, Comparator c) {
        StdRandom.shuffle(a):
        sort(a, 0, a.length - 1, c);
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) {
            return;
        int | = partition(a, lo, hi);
        sort(a, lo, j - 1);
        sort(a, j + 1, hi);
    private static void sort(Object[] a, int lo, int hi, Comparator c) {
        if (hi <= lo) {
            return:
        int i = partition(a, lo, hi, c);
        sort(a, lo, i - 1, c);
        sort(a, i + 1, hi, c);
```



An Algorithmic Improvement
When the input array has a large number of duplicates, 3-way quick sort can significantly improve performance





			a []											
1t	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
			В	R	В	R	W	W	W	R	R	W	R	В



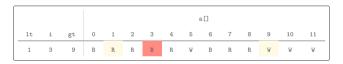


























			a[]											
lt	i	gt	0	1	2	3	4	5	6	7	8	9	10	11
3	8	7	В	В	В	R	R	R	R	R	W	W	W	W



```
☑ Quick3way.java
public class Quick3wav {
    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length - 1);
    public static void sort(Object[] a, Comparator c) {
        StdRandom.shuffle(a):
        sort(a, 0, a, length - 1, c):
    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) {
            return;
        int lt = lo, gt = hi;
        Comparable v = a[lo]:
        int i = lo + 1:
        while (i <= gt) {
            int cmp = a[i].compareTo(v);
            if (cmp < 0) {
                exchange(a, lt++, i++);
            } else if (cmp > 0) {
                exchange(a, i, gt--):
            } else {
                i++:
        sort(a, lo, lt - 1):
        sort(a, gt + 1, hi):
    private static void sort(Object[] a. int lo. int hi. Comparator c) {
        if (hi <= 10) {
            return:
```

```
🗷 Quick3way.java
        int lt = lo, gt = hi;
        Object v = a[lo];
        int i = lo + 1:
        while (i <= gt) {
            int cmp = c.compare(a[i], v);
            if (cmp < 0) {
                exchange(a, lt++, i++);
            } else if (cmp > 0) {
                exchange(a, i, gt--);
            } else {
                1++;
        sort(a, lo, lt - 1, c);
        sort(a, gt + 1, hi, c);
```



Given an array of n keys with k of them distinct, let f_i be the frequency of the ith key and $p_i = f_i/n$ the probability that the ith key is found when the array is sampled

Given an array of n keys with k of them distinct, let f_i be the frequency of the ith key and $p_i = f_i/n$ the probability that the ith key is found when the array is sampled

The Shannon entropy of the keys is defined as

$$H = -(p_1 \lg p_1 + p_2 \lg p_2 + \dots + p_k \lg p_k)$$

= $-\sum_{i=1}^k p_i \lg p_i$

Given an array of n keys with k of them distinct, let f_i be the frequency of the ith key and $p_i = f_i/n$ the probability that the ith key is found when the array is sampled

The Shannon entropy of the keys is defined as

$$H = -(p_1 \lg p_1 + p_2 \lg p_2 + \dots + p_k \lg p_k)$$

=
$$-\sum_{i=1}^k p_i \lg p_i$$

Note that $0 \le H \le \ln n$

Given an array of n keys with k of them distinct, let f_i be the frequency of the ith key and $p_i = f_i/n$ the probability that the ith key is found when the array is sampled

The Shannon entropy of the keys is defined as

$$H = -(p_1 \lg p_1 + p_2 \lg p_2 + \dots + p_k \lg p_k)$$

=
$$-\sum_{i=1}^{k} p_i \lg p_i$$

Note that $0 \le H \le \ln n$

Running time for 3-way quick sort in terms of Shannon entropy is T(n) = nH