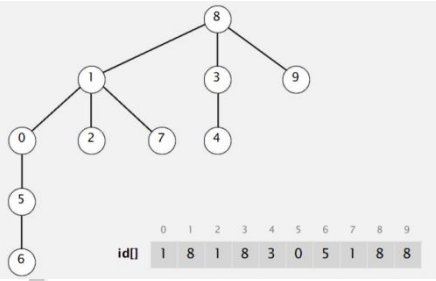
```
public class UF

        UF(int N)                      initialize union-find data structure with N singleton objects (0 to N – 1)

     void union(int p, int q)          add connection between p and q

      int find(int p)                  component identifier for p (0 to N – 1)

   boolean connected(int p, int q)     are p and q in the same component?
```

```
public class QuickFindUF {

   private int[] id;

   public QuickFindUF (int N) {
      id = new int[N];
      for (int i = 0; i < N; i++)
         id[i] = i;
   }

   public int find(int p)
   {  return id[p];  }

   public void union (int p, int q) {
      int pid = id[p];
      int qid = id[q];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = qid;
   }
}
```

1

```
public class QuickUnionUF {

   private int[] id;

   public QuickUnionUF (int N) {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   public int find (int i) {
      while (i != id[i]) i = id[i];
      return i;
   }

   public void union (int p, int q) {
      int i = find(p);
      int j = find(q);
      id[i] = j;
   }
}
```

2




```
public int find (int i) {
   while (i != id[i]) {
      id[i] = id[id[i]];
      i = id[i];
   }
   return i;
}
```

4

将到 root 的所有结点直接连接 root

Union. Modify quick-union to:

- Link root of smaller tree to root of larger tree.
- Update the `sz[]` array.

3

Weighted-Union
尽量平均树的高

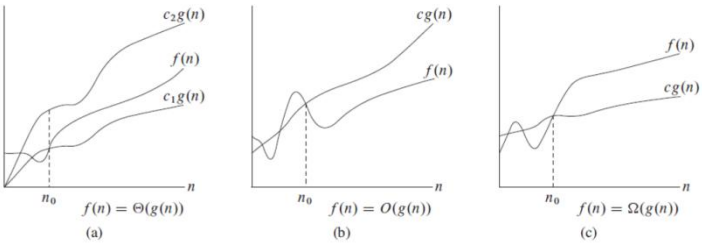
```
int i = find(p);
int j = find(q);
if (i == j) return;
if  (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
else                { id[j] = i; sz[i] += sz[j]; }
```

algorithm	worst-case time
quick-find	M N
quick-union	M N
weighted QU	N + M log N
QU + path compression	N + M log N
weighted QU + path compression	N + M lg* N

order of growth for M union-find operations on a set of N objects

algorithm	initialize	union	find	connected
quick-find	N	N	1	1
quick-union	N	N †	N	N
weighted QU	N	lg N †	lg N	lg N

† includes cost of finding roots



Ex 2. Compares for binary search.

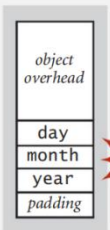
Best: ~ 1

Average: ~ lg N

Worst: ~ lg N

notation	provides	example	shorthand for	used to
Tilde	leading term	~ 10 N^2	10 N^2 10 N^2 + 22 N log N 10 N^2 + 2 N + 37	provide approximate model
Big Theta	asymptotic order of growth	$\Theta(N^2)$	½ N^2 10 N^2 5 N^2 + 22 N log N + 3N	classify algorithms
Big Oh	$\Theta(N^2)$ and smaller	$O(N^2)$	10 N^2 100 N 22 N log N + 3 N	develop upper bounds
Big Omega	$\Theta(N^2)$ and larger	$\Omega(N^2)$	½ N^2 N^5 N^3 + 22 N log N + 3 N	develop lower bounds

```
public class Date {
   private int day;
   private int month;
   private int year;
...
}
```



16 bytes (object overhead)
4 bytes (int)
4 bytes (int)
4 bytes (int)
4 bytes (padding)
32 bytes

```
public class WeightedQuickUnionUF {
   private int[] id;
   private int[] sz;
   private int count;

   public WeightedQuickUnionUF (int N) {
```

16 bytes (object overhead)
8 + (4N + 24) bytes each (reference + int[] array)
4 bytes (int)
4 bytes (padding)
8N + 88 bytes

```
public static int binarySearch (int[] a, int key) {
   int lo = 0, hi = a.length-1;

   while (lo <= hi) {
      int mid = lo + (hi - lo) / 2;
      if      (key < a[mid]) hi = mid - 1;
      else if (key > a[mid]) lo = mid + 1;
      else return mid;
   }
   return -1;
}
```

Total memory usage for a data type value:

- Primitive type: 4 bytes for `int`, 8 bytes for `double`, ...
- Object reference: 8 bytes.
- Array: 24 bytes + memory for each array entry.
- Object: 16 bytes + memory for each instance variable.
- Padding: round up to multiple of 8 bytes.

+ 8 extra bytes per inner class object (for reference to enclosing class)

```
public class LinkedStackOfStrings {

   private Node first = null;

   private class Node {
      String item;
      Node next;
   }

   public boolean isEmpty() {
      return first == null;
   }

   public void push (String item) {
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop() {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

1

```
public class FixedCapacityStackOfStrings {

   private String[] s;
   private int N = 0;

   public FixedCapacityStackOfStrings (int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push (String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

a cheat (stay tuned)

数组使用率在 25%~100%

```
public ResizingArrayStackOfStrings()
{  s = new String[1];  }

public void push (String item) {
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}

private void resize (int capacity) {
   String[] copy = new String[capacity];
   for (int i = 0; i < N; i++)
      copy[i] = s[i];
   s = copy;
}

public String pop() {
   String item = s[--N];
   s[N] = null;
   if (N > 0 && N == s.length/4) resize(s.length/2);
   return item;
}
```

3

```
public class LinkedQueueOfStrings {

   private Node first, last;

   private class Node
   {  /* same as in LinkedStackOfStrings */  }

   public boolean isEmpty()
   {  return first == null;  }

   public void enqueue (String item) {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public String dequeue() {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

4

```
public interface Iterator<Item> {
   boolean hasNext();
   Item next();
   void remove();
}
```

optional; use at your own risk

```
public interface Iterable<Item> {
   Iterator<Item> iterator();
}
```

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item> {
   ...
   public Iterator<Item> iterator() { return new ListIterator(); }

   private class ListIterator implements Iterator<Item> {
      private Node current = first;

      public boolean hasNext() {  return current != null;  }
      public void remove()     {  /* not supported */      }
      public Item next() {
         Item item = current.item;
         current   = current.next;
         return item;
      }
   }
}
```

throw UnsupportedOperationException

throw NoSuchElementException if no more items in iteration

迭代接口

```
public class Selection {

   public static void sort (Comparable[] a) {
      int N = a.length;
      for (int i = 0; i < N; i++) {
         int min = i;
         for (int j = i+1; j < N; j++)
            if (less(a[j], a[min]))
               min = j;
         exch( a, i, min);
      }
   }
```

```
public class Insertion {

   public static void sort (Comparable[] a) {
      int N = a.length;
      for (int i = 0; i < N; i++)
         for (int j = i; j > 0; j--)
            if (less(a[j], a[j-1]))
               exch( a, j, j-1);
            else break;
   }
```

Def. An inversion is a pair of keys that are out of order.

Proposition. For partially-sorted arrays, insertion sort runs in linear time.

Pf. Number of exchanges equals the number of inversions.

Def. An array is partially sorted if the number of inversions is $\le c\,N$.

- Ex 1. A sorted array has 0 inversions.
- Ex 2. A subarray of size 10 appended to a sorted subarray of size N.

```
public class Shell {

   public static void sort (Comparable[] a) {

      int N = a.length;

      int h = 1;
      while (h < N/3) h = 3*h + 1; // 1, 4, 13, 40, 121, 364, ...

      while (h >= 1) {
         // h-sort the array.
         for (int i = h; i < N; i++) {
            for (int j = i; j >= h && less(a[j], a[j-h]); j -= h)
               exch( a, j, j-h);
         }

         h = h/3;
      }
   }
```

Sedgewick. 1, 5, 19, 41, 109, 209, 505, 929, 2161, 3905, ...

Good. Tough to beat in empirical studies.

merging of $(9 \square 4^i) - (9 \square 2^i) + 1$ and $4^i - (3 \square 2^i) + 1$

```
public class StdRandom {

   ...

   public static void shuffle (Object[] a) {
      int N = a.length;

      for (int i = 0; i < N; i++) {

         int r = StdRandom.uniform( i + 1);
         exch( a, i, r);
      }
   }
}
```

Between 0 - i

```
ResultType DandC(Problem p) {
   if (p is trivial) {
      solve p directly                          t_s
      return the result
   } else {
      divide p into p1,p2,...,pn                t_d
      for (i = 1 to n)
         ri = DandC(pi)                         t_r
      combine r1,r2,...,rn into r               t_c
      return r
   }
}
```

Trivial Case / Divide / Recursive / Combine

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {

   for (int k = lo; k <= hi; k++)                                   copy
      aux[k] = a[k];

   int i = lo, j = mid+1;
   for (int k = lo; k <= hi; k++) {
      if      (i > mid)                    a[k] = aux[j++];
      else if (j > hi)                     a[k] = aux[i++];          merge
      else if (less(aux[j], aux[i]))       a[k] = aux[j++];
      else                                 a[k] = aux[i++];
   }
}
```

```
public class Merge {

   private static void merge (...) {
      /* as before */
   }

   private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
      if (hi <= lo) return;
      int mid = lo + (hi - lo) / 2;
      sort(a, aux, lo, mid);
      sort(a, aux, mid+1, hi);
      merge(a, aux, lo, mid, hi);
   }

   public static void sort (Comparable[] a) {
      Comparable[] aux = new Comparable[a.length];
      sort( a, aux, 0, a.length - 1);
   }
}
```

```
private static void sort (Comparable[] a, Comparable[]
   if (hi <= lo + CUTOFF - 1) {
      Insertion.sort( a, lo, hi);
      return;
   }
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   merge(a, aux, lo, mid, hi);
}
```

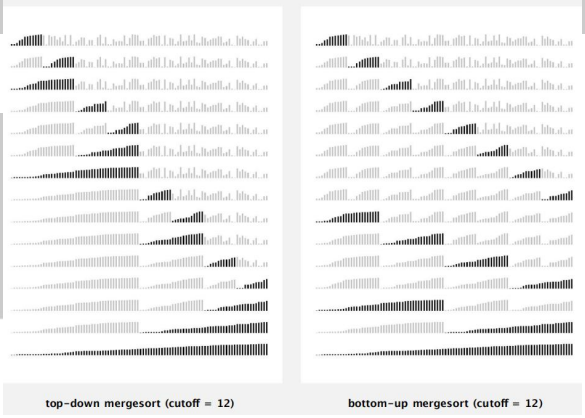
优化 1

```
private static void sort (Comparable[] a,
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   sort (a, aux, lo, mid);
   sort (a, aux, mid+1, hi);
   if (!less(a[mid+1], a[mid])) return;
   merge(a, aux, lo, mid, hi);
}
```

优化 2

优化 3，交换红框的 a 和 aux

```
public class MergeBU {

   private static void merge (...) {
      /* as before */
   }

   public static void sort (Comparable[] a) {
      int N = a.length;
      Comparable[] aux = new Comparable[N];
      for (int sz = 1; sz < N; sz = sz+sz)
         for (int lo = 0; lo < N-sz; lo += sz+sz)
            merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
   }
}
```



Timsort

- Natural mergesort.
- Use binary insertion sort to make initial runs (if needed).
- A few more clever optimizations.

Consequence. Linear time on many arrays with pre-existing order.

```
private static int partition(Comparable[] a, int lo, int hi) {

   int i = lo, j = hi+1;

   while (true) {

      while (less(a[++i], a[lo]))          find item on left to swap
         if (i == hi) break;

      while (less(a[lo], a[--j]))          find item on right to swap
         if (j == lo) break;

      if (i >= j) break;                   check if pointers cross
      exch( a, i, j);                      swap
   }

   exch( a, lo, j);                        swap with partitioning item
   return j;                               return index of item now known to be in place
}
```

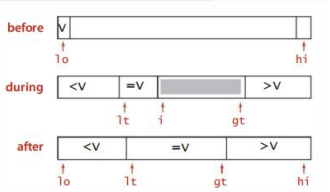
```
private static void sort (Comparable[] a, int lo, int hi) {
   if (hi <= lo + CUTOFF - 1) {
      Insertion.sort( a, lo, hi);
      return;
   }
   int j = partition( a, lo, hi);
   sort( a, lo, j-1);
   sort( a, j+1, hi);
}
```

优化 1
原 sort 去掉白色部分

```
private static void sort (Comparable[] a, int lo, int hi) {
   if (hi <= lo) return;

   int median = medianOf3( a, lo, lo + (hi - lo)/2, hi);
   swap( a, lo, median);

   int j = partition( a, lo, hi);
   sort( a, lo, j-1);
   sort( a, j+1, hi);
}
```

优化 2

```
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;

   while (i <= gt) {
      int cmp = a[i].compareTo(v);
      if      (cmp < 0) exch( a, lt++, i++);
      else if (cmp > 0) exch( a, i, gt--);
      else              i++;
   }

   sort( a, lo, lt - 1);
   sort( a, gt + 1, hi);
}
```




```
public static Comparable select(Comparable[] a, int k)
{
    StdRandom.shuffle( a);

    int lo = 0, hi = a.length - 1;
    while (hi > lo) {
       int j = partition( a, lo, hi);
       if      (j < k)  lo = j + 1;
       else if (j > k)  hi = j - 1;
       else             return a[k];
    }
    return a[k];
}
```

```
private void swim( int k) {

   while (k > 1 && less( k/2, k)) {

      exch(k, k/2);
      k = k/2;
   }
}
```

parent of node at k is at k/2

```
public Key delMax () {
   Key max = pq[1];
   exch(1, n--);
   sink(1);
   pq[n+1] = null;
   return max;
}
```

```
private void sink (int k) {

   while (2*k <= n) {
      int j = 2*k;
      if (j < n && less( j, j+1)) j++;
      if (!less(k, j)) break;
      exch( k, j);
      k = j;
   }
}
```

children of node at k are 2*k and 2*k+1

```
public void insert (Key x) {
   pq[++n] = x;

   swim(n);
}
```

```
public class MaxPQ<Key extends Comparable<Key>> {

   private Key[] pq;
   private int n;

   public MaxPQ(int capacity)
   {  pq = (Key[]) new Comparable[capacity+1];  }

   public boolean isEmpty()
   {  return n == 0;  }
   public void insert(Key key)  // see previous code
   public Key delMax()          // see previous code

   private void swim(int k)     // see previous code
   private void sink(int k)     // see previous code

   private boolean less(int i, int j)
   {  return pq[i].compareTo(pq[j]) < 0;  }
   private void exch(int i, int j)
   {  Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;  }
}
```

Floyd's "bounce" heuristic.

- Sink key at root all the way to bottom. ← only 1 compare per node
- Swim key back up. ← some extra compares and exchanges
- Overall, fewer compares; more exchanges.

```
public class Heap {
   public static void sort (Comparable[] a) {
      int n = a.length;
      for (int k = n/2; k >= 1; k--)
         sink( a, k, n);
      while (n > 1) {
         exch( a, 1, n);
         sink( a, 1, --n);
      }
   }

   private static void sink(Comparable[] a, int k, int n)
   {  /* as before */  }

   private static boolean less(Comparable[] a, int i, int j)
   {  /* as before */  }

   private static void exch(Object[] a, int i, int j)
   {  /* as before */  }
}
```

but make static (and pass arguments)

but convert from 1-based indexing to 0-base indexing

a[i]

N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values		S	O	R	T	E	X	A	M	P	L	E	
11	5	S	O	R	T	L	X	A	M	P	E	E	
11	4	S	O	R	T	L	X	A	M	P	E	E	
11	3	S	O	X	T	L	R	A	M	P	E	E	
11	2	S	T	X	P	L	R	A	M	O	E	E	
11	1	X	T	S	P	L	R	A	M	O	E	E	
heap-ordered		X	T	S	P	L	R	A	M	O	E	E	
10	1	T	P	S	O	L	R	A	M	E	E	X	
9	1	S	P	R	O	L	E	A	M	E	T	X	
8	1	R	P	E	O	L	E	A	M	S	T	X	
7	1	P	O	E	M	L	E	A	R	S	T	X	
6	1	O	M	E	A	L	E	P	R	S	T	X	
5	1	M	L	E	A	E	O	P	R	S	T	X	
4	1	L	E	E	A	M	O	P	R	S	T	X	
3	1	E	A	E	L	M	O	P	R	S	T	X	
2	1	E	A	E	L	M	O	P	R	S	T	X	
1	1	A	E	E	L	M	O	P	R	S	T	X	
sorted result		A	E	E	L	M	O	P	R	S	T	X	

Heapsort trace (array contents just after each sink)

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✔	✔	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✔		$n \log_3 n$	?	$c\, n^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
timsort		✔	n	$n \lg n$	$n \lg n$	improves mergesort when preexisting order
quick	✔		$n \lg n$	$2\, n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2\, n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
heap	✔		$3\, n$	$2\, n \lg n$	$2\, n \lg n$	$n \log n$ guarantee; in-place
?	✔	✔	n	$n \lg n$	$n \lg n$	holy sorting grail