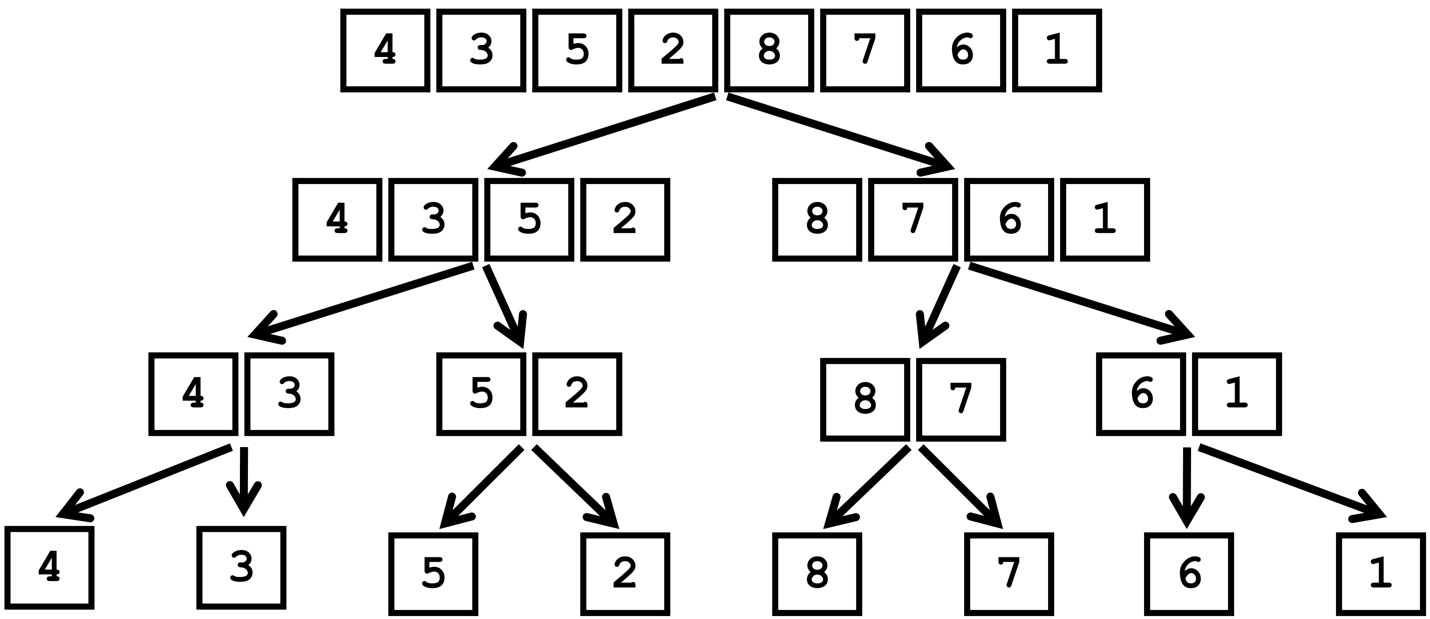
Divide and Conquer

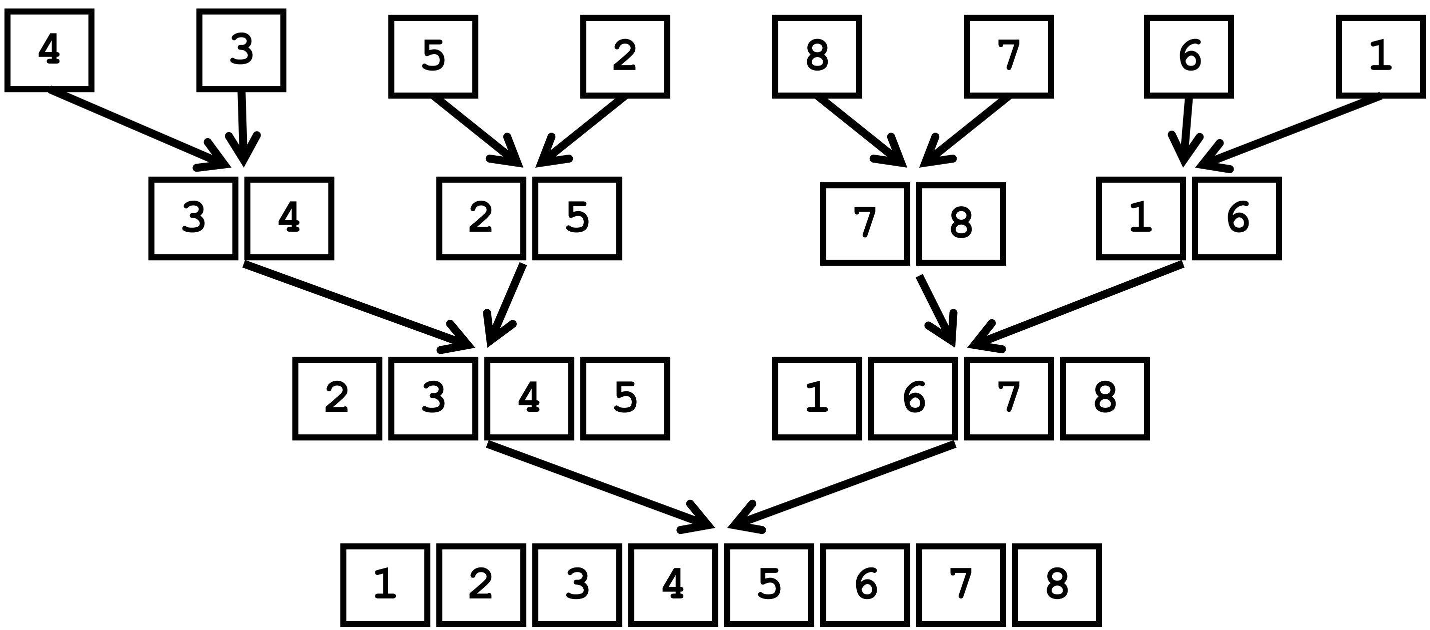
- divide and conquer algorithms typically recursively divide a problem into several smaller sub-problems until the sub-problems are small enough that they can be solved directly

Merge Sort

- merge sort is a divide and conquer algorithm that sorts a list of numbers by recursively splitting the list into two halves

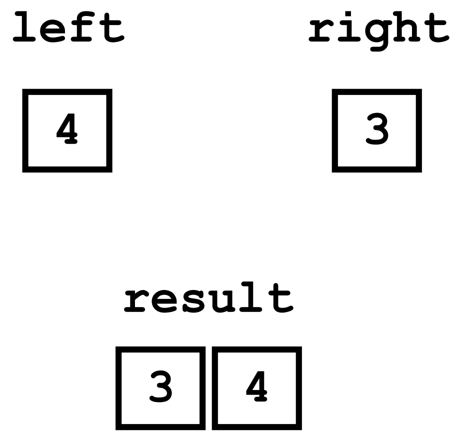


- the split lists are then merged into sorted sub-lists



Merging Sorted Sub-lists

- two sub-lists of length 1



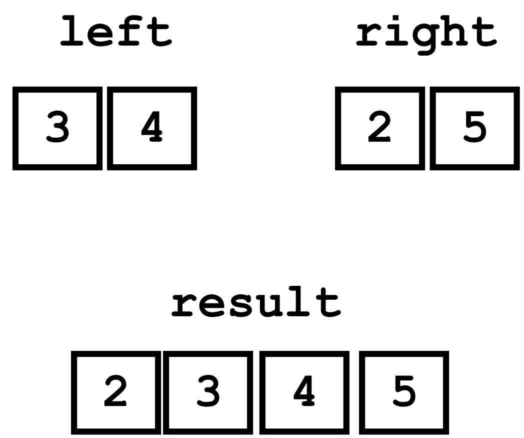
1 comparison
2 copies

```java
LinkedList<Integer> result = new LinkedList<Integer>();

int fL = left.getFirst();
int fR = right.getFirst();
if (fL < fR) {
  result.add(fL);
  left.removeFirst();
}
else {
  result.add(fR);
  right.removeFirst();
}
if (left.isEmpty()) {
  result.addAll(right);
}
else {
  result.addAll(left);
}
```

Merging Sorted Sub-lists

- two sub-lists of length 2



3 comparisons
4 copies

```
LinkedList<Integer> result = new LinkedList<Integer>();

while (left.size() > 0 && right.size() > 0 ) {
  int fL = left.getFirst();
  int fR = right.getFirst();
  if (fL < fR) {
    result.add(fL);
    left.removeFirst();
  }
  else {
    result.add(fR);
    right.removeFirst();
  }
}
if (left.isEmpty()) {
  result.addAll(right);
}
else {
  result.addAll(left);
}
```

Merging Sorted Sub-lists

- two sub-lists of length 4

`left`

2	3	4	5

`right`

1	6	7	8

`result`

1	2	3	4	5	6	7	8

5 comparisons
8 copies

Simplified Complexity Analysis

- in the worst case merging a total of **`n`** elements requires

  `n - 1` comparisons **`+`**

  `n` copies

  `= 2n - 1` total operations

- the worst-case complexity of merging is the order of *O(n)*

Informal Analysis of Merge Sort

- suppose the running time (the number of operations) of merge sort is a function of the number of elements to sort
  - let the function be $T(n)$
- merge sort works by splitting the list into two sub-lists (each about half the size of the original list) and sorting the sub-lists
  - this takes $2T(n/2)$ running time
- then the sub-lists are merged
  - this takes $O(n)$ running time
- total running time $T(n) = 2T(n/2) + O(n)$

Solving the Recurrence Relation

$T(n)$ approaches…

$$
\begin{aligned}
T(n) &\rightarrow 2T(n/2) + O(n) \\
&\approx 2T(n/2) + n \\
&= 2[\ 2T(n/4) + n/2\ ] + n \\
&= 4T(n/4) + 2n \\
&= 4[\ 2T(n/8) + n/4\ ] + 2n \\
&= 8T(n/8) + 3n \\
&= 8[\ 2T(n/16) + n/8\ ] + 3n \\
&= 16T(n/16) + 4n \\
&= 2^k T(n/2^k) + kn
\end{aligned}
$$

Solving the Recurrence Relation

$$T(n) \quad = \qquad 2^k T(\underline{n/2^k}) + kn$$

- for a list of length **1** we know $T(1) = 1$
  - if we can substitute $T(1)$ into the right-hand side of $T(n)$ we might be able to solve the recurrence
  - we have $T(n/2^k)$ on the right-hand side, so we need to find some value of k such that

$$\underline{n/2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2(n)$$

Solving the Recurrence Relation

$$
\begin{aligned}
T(n) &= 2^{\log_2 n}\, T\left(n/2^{\log_2 n}\right) + n\log_2 n \\
&= n\,T(1) + n\log_2 n \\
&= n + n\log_2 n \\
&\in O(n\log_2 n)
\end{aligned}
$$

Quicksort

- quicksort, like mergesort, is a divide and conquer algorithm for sorting a list or array
- it can be described recursively as follows:
  1. choose an element, called the *pivot*, from the list
  2. reorder the list so that:
     - values less than the pivot are located before the pivot
     - values greater than the pivot are located after the pivot
  3. quicksort the sublist of elements before the pivot
  4. quicksort the sublist of elements after the pivot

Quicksort

- step 2 is called the *partition* step
- consider the following list of unique elements

0	8	7	6	4	3	5	1	2	9

- assume that the pivot is 6

Quicksort

- the partition step reorders the list so that:
  - values less than the pivot are located before the pivot
    - we need to move the cyan elements before the pivot

0	9	7	6	4	3	5	1	2	8

  - values greater than the pivot are located after the pivot
    - we need to move the red elements after the pivot

0	9	7	6	4	3	5	1	2	8

Quicksort

- can you describe an algorithm to perform the partitioning step?
  - talk amongst yourselves here

Quicksort

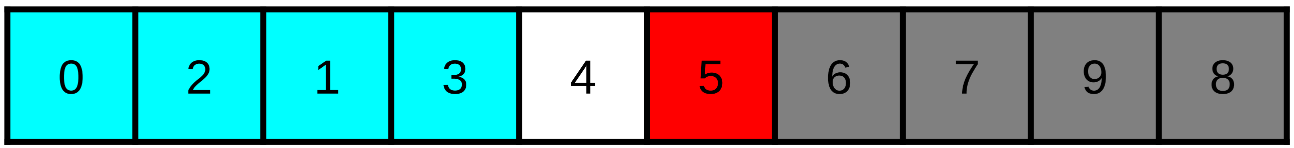
- after partitioning the list looks like:

0	2	1	5	4	3	6	7	9	8

- partioning has 3 results:
  - the pivot is in its correct final sorted location
  - the left sublist contains only elements less than the pivot
  - the right sublist contains only elements greater than the pivot

Quicksort

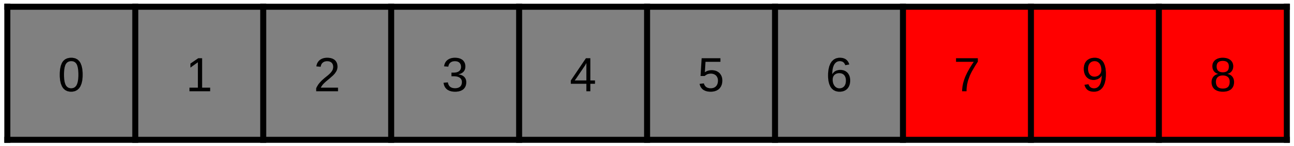
- after partitioning we recursively quicksort the left sublist
- for the left sublist, let's assume that we choose 4 as the pivot
  - after partitioning the left sublist we get:



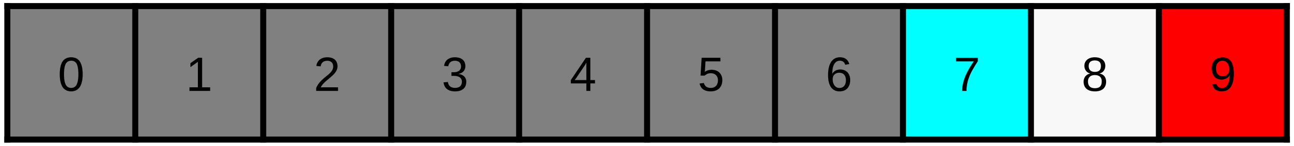
    - we then recursively quicksort the left and right sublists
      - and so on...

Quicksort

- eventually, the left sublist from the first pivoting operation will be sorted; we then recursively quicksort the right sublist:



- if we choose 8 as the pivot and partition we get:



- the left and right sublists have size 1 so there is nothing left to do

Quicksort

- the computational complexity of quicksort depends on:
  - the computational complexity of the partition operation
    - without proof I claim that this is $O(n)$ for a list of size n
  - how the pivot is chosen

Quicksort

- let's assume that when we choose a pivot we always choose the smallest (or largest) value in the sublist
  - yields a sublist of size $(n-1)$ which we recursively quicksort
- let $T(n)$ be the number of operations needed to quicksort a list of size n when choosing a pivot as described above
  - then the recurrence relation is:

$$T(n) = T(n-1) + O(n)$$

same as selection sort

  - solving the recurrence results in

$$T(n) = O(n^2)$$

Quicksort

- let's assume that when we choose a pivot we always choose the median value in the sublist
  - yields 2 sublists of size $\left(\frac{n}{2}\right)$ which we recursively quicksort
- let $T(n)$ be the number of operations needed to quicksort a list of size n when choosing a pivot as described above
  - then the recurrence relation is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

same as merge sort

  - solving the recurrence results in

$$T(n) = O(n \log_2 n)$$

- what is the fastest way to sort a deck of playing cards?
- what is the big-O complexity?
- talk amongst ourselves here….

Proving correctness and terminaton

Proving Correctness and Termination

- to show that a recursive method accomplishes its goal you must prove:
  1. that the base case(s) and the recursive calls are correct
  2. that the method terminates

Proving Correctness

- to prove correctness:
  1. prove that each base case is correct
  2. assume that the recursive invocation is correct and then prove that each recursive case is correct

printItToo

```
public static void printItToo(String s, int n) {
  if (n == 0) {
    return;
  }
  else {
    System.out.print(s);
    printItToo(s, n - 1);
  }
}
```

Correctness of printItToo

1. (prove the base case) If `n == 0` nothing is printed; thus the base case is correct.
2. Assume that `printItToo(s, n-1)` prints the string `s` exactly `(n - 1)` times. Then the recursive case prints the string `s` exactly `(n - 1)+1 = n` times; thus the recursive case is correct.

Proving Termination

- to prove that a recursive method terminates:
  1. define the size of a method invocation; the size must be a non-negative integer number
  2. prove that each recursive invocation has a smaller size than the original invocation

Termination of printItToo

1. **`printItToo(s, n)`** prints **`n`** copies of the string **`s`**; define the size of **`printItToo(s, n)`** to be **`n`**
2. The size of the recursive invocation **`printItToo(s, n-1)`** is **`n-1`** (by definition) which is smaller than the original size **`n`**.

countZeros

```
public static int countZeros(long n) {

  if(n == 0L) {  // base case 1
    return 1;
  }
  else if(n < 10L) {  // base case 2
    return 0;
  }

  boolean lastDigitIsZero = (n % 10L == 0);
  final long m = n / 10L;
  if(lastDigitIsZero) {
    return 1 + countZeros(m);
  }
  else {
    return countZeros(m);
  }
}
```

Correctness of countZeros

1. (base cases) If the number has only one digit then the method returns **`1`** if the digit is zero and **`0`** if the digit is not zero; therefore, the base case is correct.
2. (recursive cases) Assume that **`countZeros(n/10L)`** is correct (it returns the number of zeros in the first **`(d - 1)`** digits of **`n`**).

   There are two recursive cases:

Correctness of countZeros

a. If the last digit in the number is zero, then the recursive case returns **`1 +`** the number of zeros in the first **`(d - 1)`** digits of **`n`**, which is correct.

b. If the last digit in the number is one, then the recursive case returns the number of zeros in the first **`(d - 1)`** digits of **`n`**, which is correct.

Termination of countZeros

1. Let the size of **`countZeros(n)`** be **`d`** the number of digits in the number **`n`**.
2. The size of the recursive invocation **`countZeros(n/10L)`** is **`d-1`**, which is smaller than the size of the original invocation.

Selection Sort

```
public class Recursion {

  // minToFront not shown

  public static void selectionSort(List<Integer> t) {
    if (t.size() > 1) {
      Recursion.minToFront(t);
      Recursion.selectionSort(t.subList(1, t.size()));
    }
  }

                              Prove that selection sort is correct and terminates.

}
```

Proving Termination

- prove that the algorithm on the next slide terminates

```java
public class Print {

  public static void done(int n) {
    if (n == 1) {
      System.out.println("done");
    }
    else if (n % 2 == 0) {
      System.out.println("not done");
      Print.done(n / 2);
    }
    else {
      System.out.println("not done");
      Print.done(3 * n + 1);
    }
  }

}
```