Recursion

notes Chapter 8

Decrease and Conquer

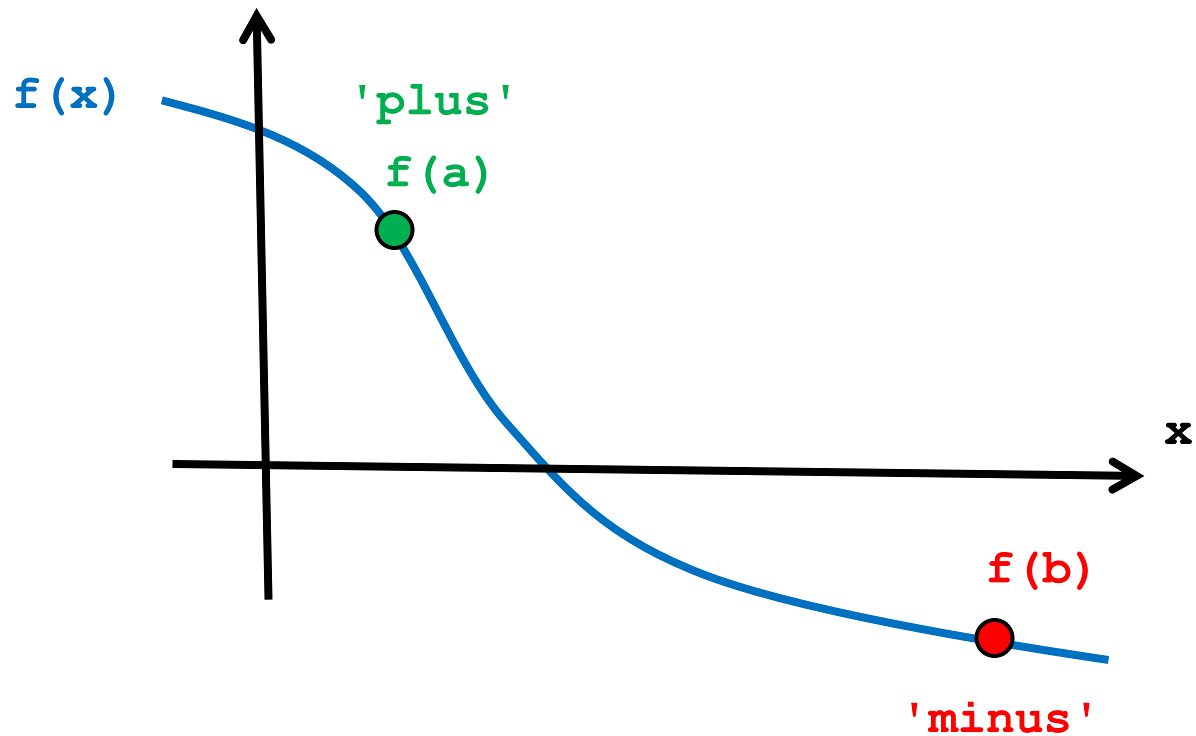
- a common strategy for solving computational problems
  - solves a problem by taking the original problem and converting it to *one* smaller version of the same problem
    - note the similarity to recursion
- decrease and conquer, and the closely related divide and conquer method, are widely used in computer science
  - allow you to solve certain complex problems easily
  - help to discover efficient algorithms

Root Finding

- suppose you have a mathematical function **f(x)** and you want to find x_0 such that $f(x_0) = 0$
  - why would you want to do this?
  - many problems in computer science, science, and engineering reduce to optimization problems
    - find the shape of an automobile that minimizes aerodynamic drag
    - find an image that is similar to another image (minimize the difference between the images)
    - find the sales price of an item that maximizes profit
  - if you can write the optimization criteria as a function **g(x)** then its derivative **f(x) = dg/dx = 0** at the minimum or maximum of **g** (as long as **g** has certain properties)

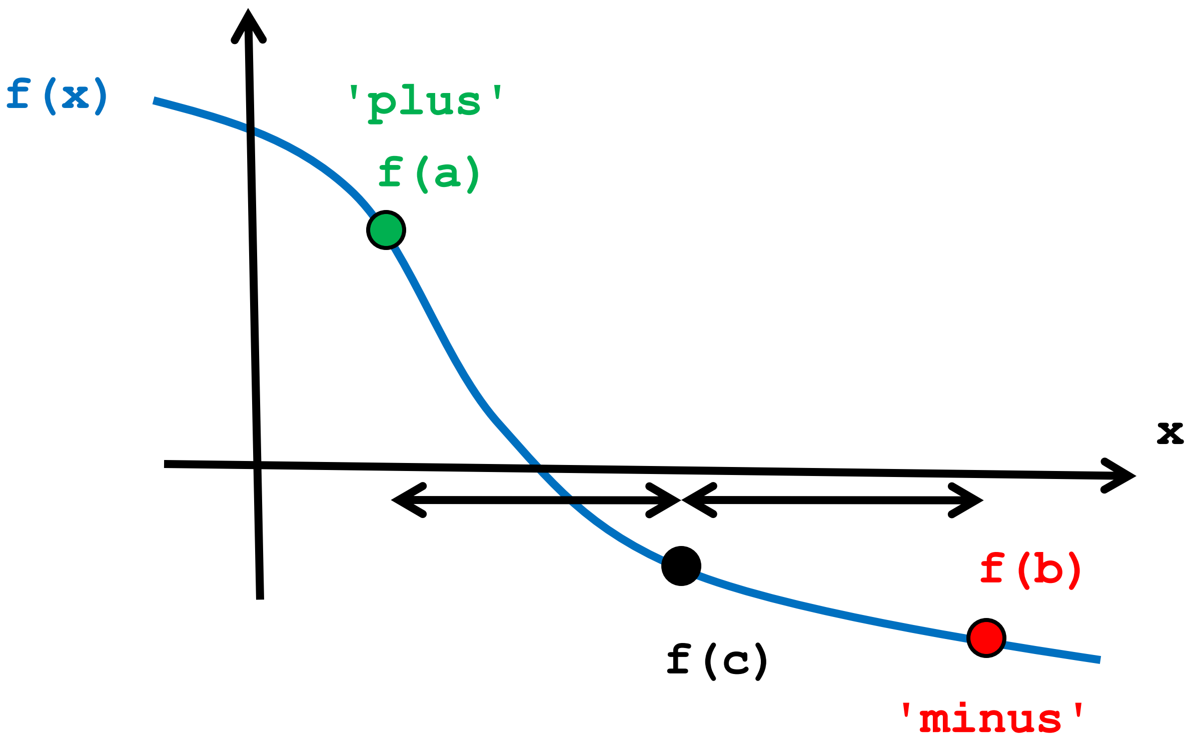
Bisection Method

- suppose you can evaluate **`f(x)`** at two points **`x = a`** and **`x = b`** such that
  - **`f(a) > 0`**
  - **`f(b) < 0`**



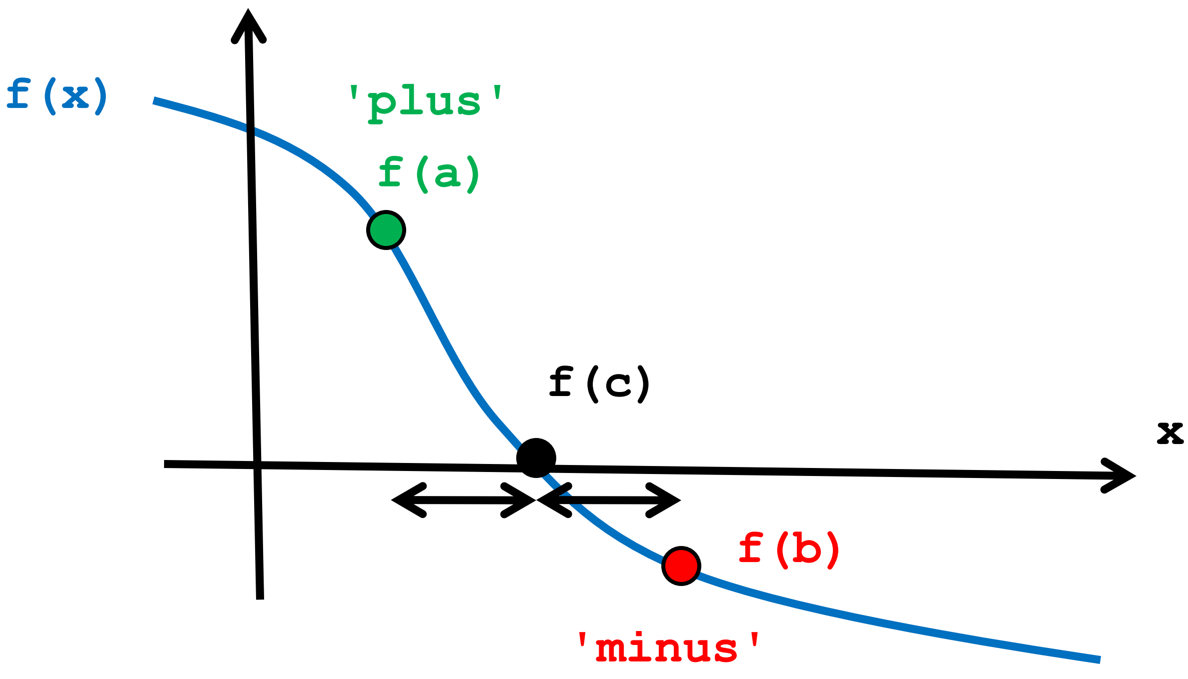
Bisection Method

- evaluate **`f(c)`** where **`c`** is halfway between **`a`** and **`b`**
  - if **`f(c)`** is close enough to zero done



Bisection Method

- otherwise **`c`** becomes the new end point (in this case, **`'minus'`**) and recursively search the range **`'plus'`** – **`'minus'`**



```
public class Bisect {

  // the function we want to find the root of
  public static double f(double x) {
    return Math.cos(x);
  }
```

```
public static double bisect(double xplus, double xminus,
                            double tolerance) {
   // base case
   double c = (xplus + xminus) / 2.0;
   double fc = f(c);
   if( Math.abs(fc) < tolerance ) {
     return c;
   }
   else if (fc < 0.0) {
     return bisect(xplus, c, tolerance);
   }
   else {
     return bisect(c, xminus, tolerance);
   }
 }
```

```
    public static void main(String[] args)
    {
            System.out.println("bisection returns: " +
                                    bisect(1.0, Math.PI, 0.001));
            System.out.println("true answer      : "
                                    + Math.PI / 2.0);
    }
}
```

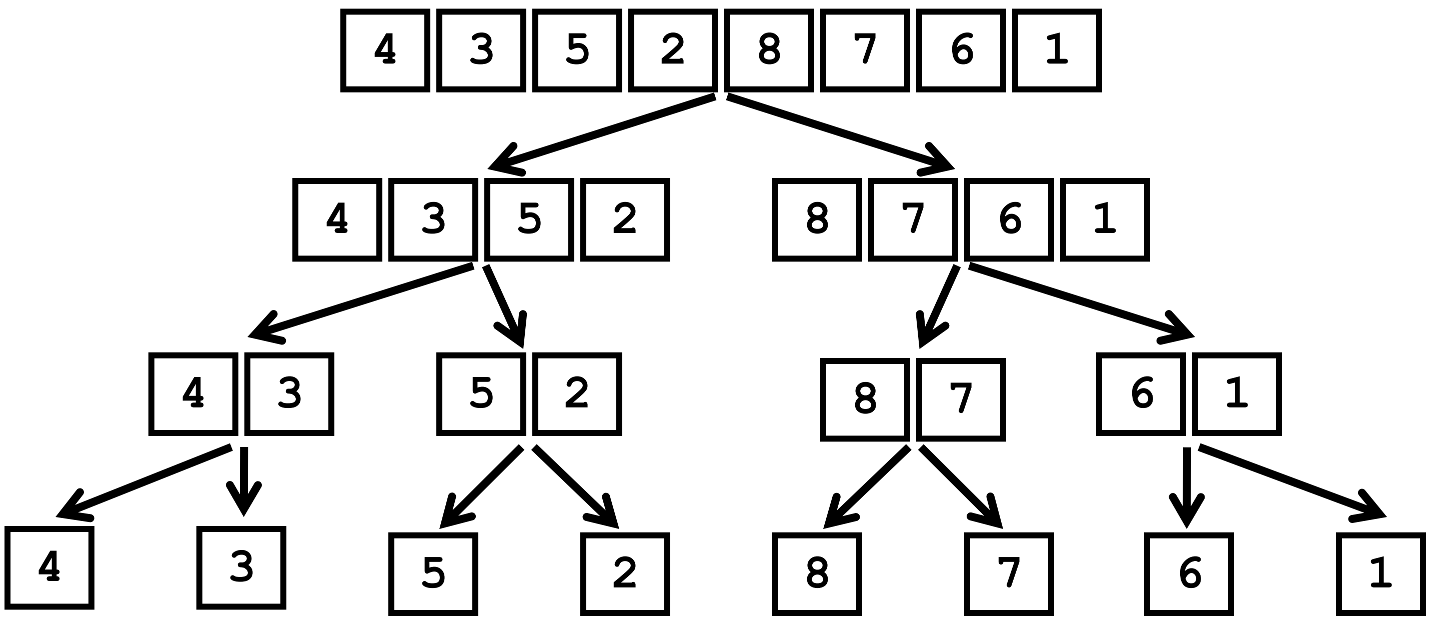
```
prints:

bisection returns: 1.5709519476855602
true answer      : 1.5707963267948966
```

Divide and Conquer

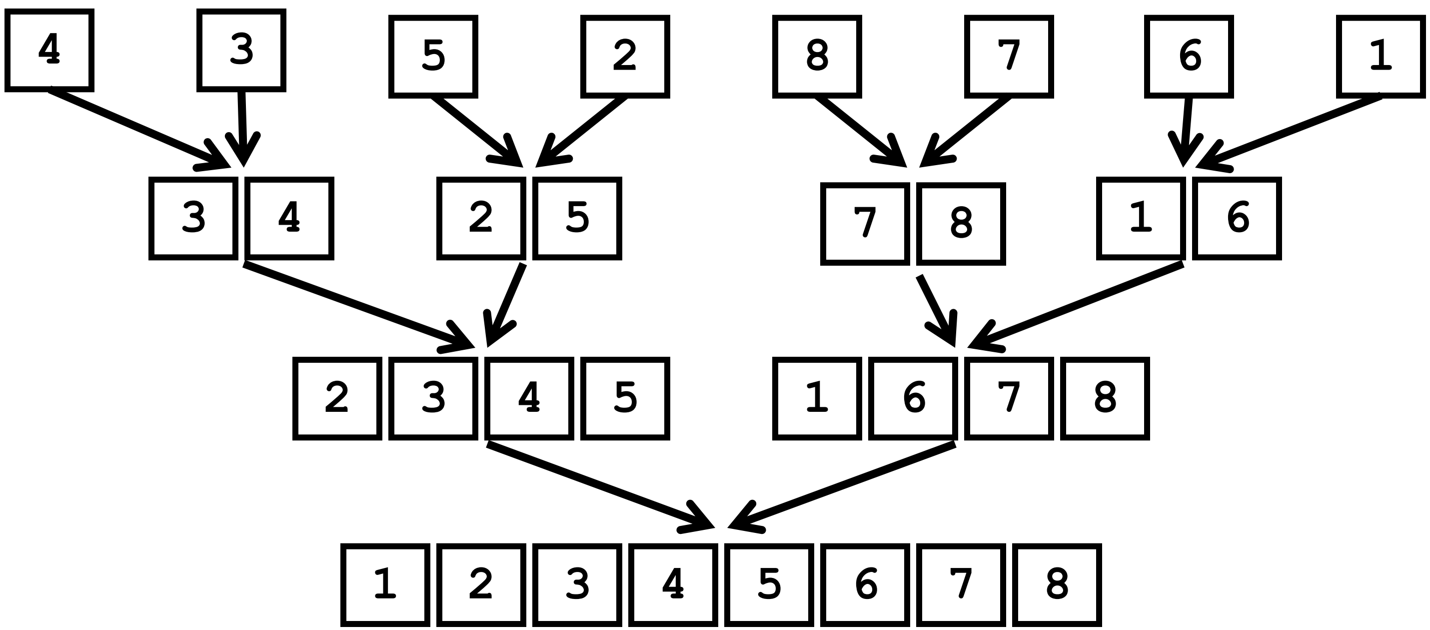
- bisection works by recursively finding which half of the range `'plus'` – `'minus'` the root lies in
  - each recursive call solves the same problem (tries to find the root of the function by guessing at the midpoint of the range)
  - each recursive call solves *one* smaller problem because half of the range is discarded
    - bisection method is decrease 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

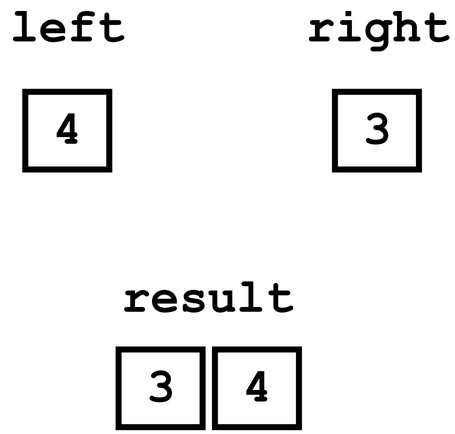


- the split lists are then merged into sorted sub-lists



Merging Sorted Sub-lists

- two sub-lists of length 1

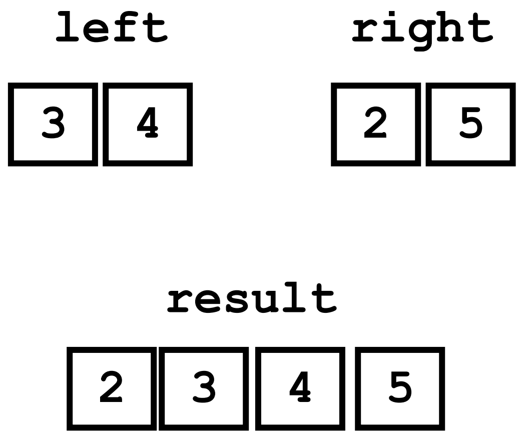


1 comparison
2 copies

```
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

```java
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

- we say that the worst-case complexity of merging is the order of $O(n)$
  - $O(\ldots)$ is called Big O notation
  - notice that we don't care about the constants 2 and 1

- formally, a function $f(n)$ is an element of $O(g(n))$ if and only if there is a positive real number M and a real number m such that

$$| f(n) | < M | g(n) | \quad \text{for all} \quad n > m$$

- is $\mathbf{2}n - \mathbf{1}$ an element of $O(n)$?
  - yes, let $M = \mathbf{2}$ and $m = \mathbf{0}$, then $\mathbf{2}n - \mathbf{1} < \mathbf{2}n$ for all $n > \mathbf{0}$

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

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

Solving the Recurrence Relation

$$
\begin{aligned}
T(n) &= 2^{\log(n)} T(n/2^{\log(n)}) + n \log(n) \\
&= n\, T(1) + n \log(n) \\
&= n + n \log(n) \\
&\in n \log(n) \qquad \text{(prove this)}
\end{aligned}
$$

Is Merge Sort Efficient?

- consider a simpler (non-recursive) sorting algorithm called insertion sort

```
// to sort an array a[0]..a[n-1]                    not Java!
for i = 0 to (n-1) {
  k = index of smallest element in sub-array a[i]..a[n-1]
  swap a[i] and a[k]
}
```

```
for i = 0 to (n-1) {                                not Java!
  for j = (i+1) to (n-1) {
    if (a[j] < a[i]) {                              1 comparison +
      k = j;                                        1 assignment
    }
  }
  tmp = a[i];   a[i] = a[k];   a[k] = tmp;          3 assignments
}
```

$$T(n) = \sum_{i=0}^{n-1}\left(\left(\sum_{j=(i+1)}^{n-1} 2\right) + 3\right)$$

$$= \sum_{i=0}^{n-1}\left(2(n-i-1)\right) \quad + \quad 3n$$

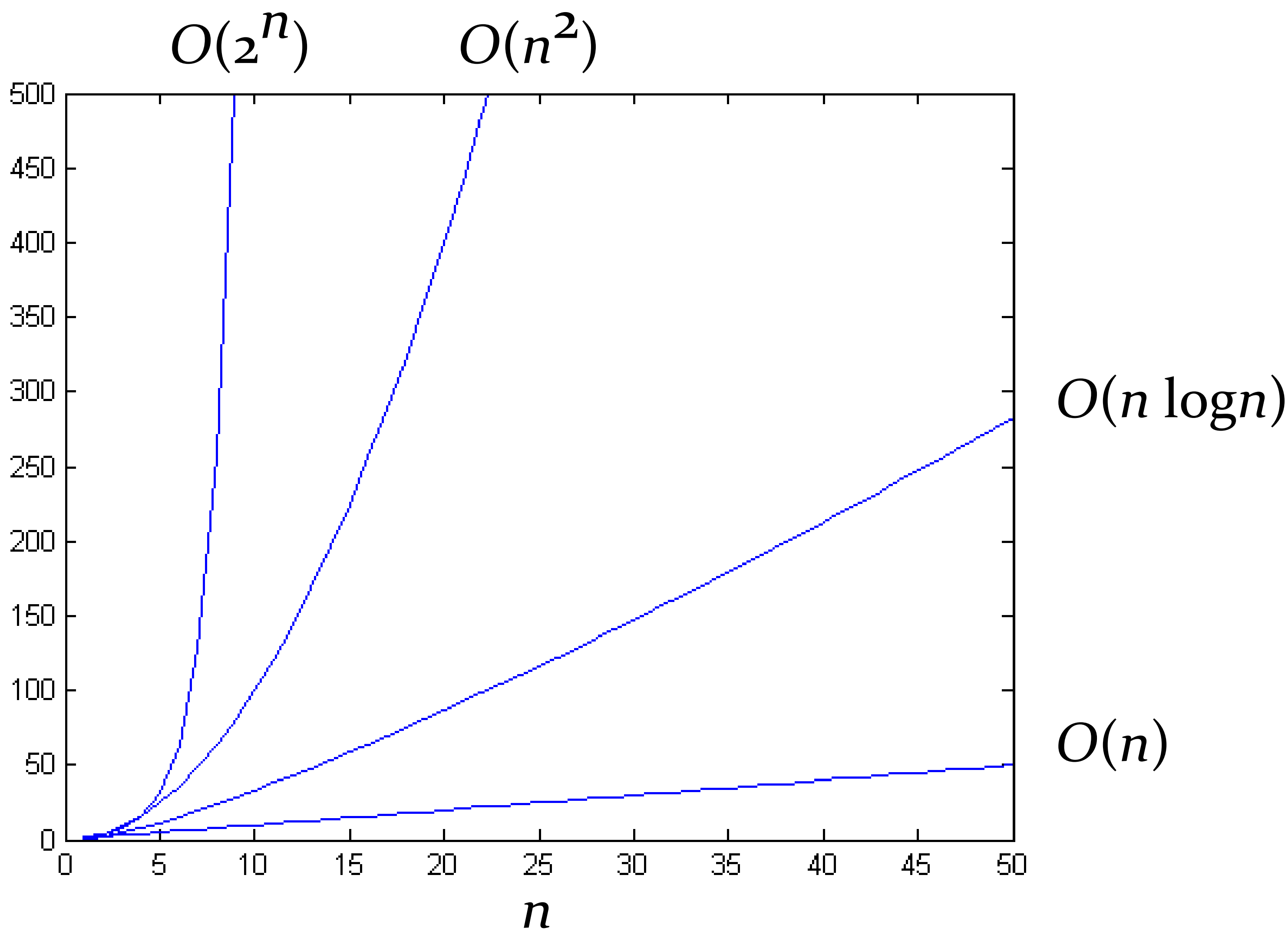
$$= 2\sum_{i=0}^{n-1} n \quad - \quad 2\sum_{i=0}^{n-1} i \quad - \quad 2\sum_{i=0}^{n-1} 1 \quad + \quad 3n$$

$$= 2n^2 \quad - \quad 2\frac{n(n-1)}{2} \quad - \quad 2n \quad + \quad 3n$$

$$= 2n^2 \quad - \quad n^2 \quad + \quad n \quad - \quad 2n \quad + \quad 3n$$

$$= n^2 \quad + \quad 2n \quad \in O\left(n^2\right)$$

Comparing Rates of Growth



Comments

- big O complexity tells you something about the running time of an algorithm as the size of the input, n, approaches infinity
  - we say that it describes the limiting, or asymptotic, running time of an algorithm
- for small values of n it is often the case that a less efficient algorithm (in terms of big O) will run faster than a more efficient one
  - insertion sort is typically faster than merge sort for short lists of numbers

Revisiting the Fibonacci Numbers

- the recursive implementation based on the definition of the Fibonacci numbers is inefficient

```
public static int fibonacci(int n) {
  if (n == 0) {
   return 0;
  }
  else if (n == 1) {
   return 1;
  }
  int f = fibonacci(n - 1) + fibonacci(n - 2);
  return f;
}
```

- how inefficient is it?
- let $T(n)$ be the running time to compute the nth Fibonacci number
  - $T(0) = T(1) = 1$
  - $T(n)$ is a recurrence relation

$$
\begin{aligned}
T(n) \quad & \rightarrow T(n-1)+T(n-2) \\
& =\big(T(n-2)+T(n-3)\big)+T(n-2) \\
& =2T(n-2)+T(n-3) \\
& >2T(n-2) \\
& >2\big(2T(n-4)\big)=4T(n-4) \\
& >4\big(2T(n-6)\big)=8T(n-6) \\
& >8\big(2T(n-8)\big)=16T(n-8) \\
& >2^{k}T(n-2k)
\end{aligned}
$$

Solving the Recurrence Relation

$T(n) > 2^k T(n - 2k)$

- 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

$$n - 2k = 1 \Rightarrow 1 + 2k = n \Rightarrow k = (n - 1)/2$$

$$T(n) > 2^k T(n-2k) = 2^{(n-1)/2} T(1) = 2^{(n-1)/2} \in O(2^n)$$