# Tutorial 1

## Data Structures '11

Jonathan Cederberg <jonathan.cederberg@it.uu.se>

Monday, September $12^{th}$, 2011

# Outline

# Outline

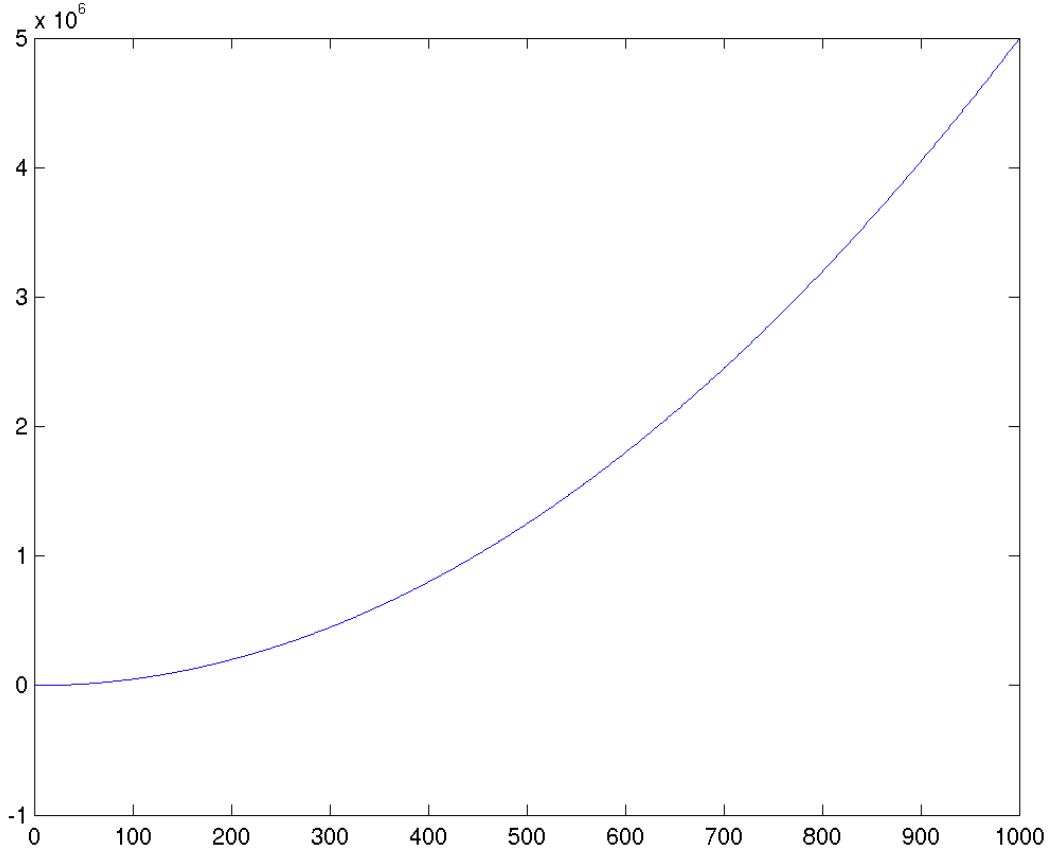
# An example of Asymptotic notation

Prove $5n^2 - 3n - 6 = \Theta(n^2)$.
That is, find $c_1, c_2, n_0$ such that for all $n > n_0$ we have

$$0 \leq c_1 \cdot n^2 \leq 5n^2 - 3n - 6 \leq c_2 \cdot n^2$$

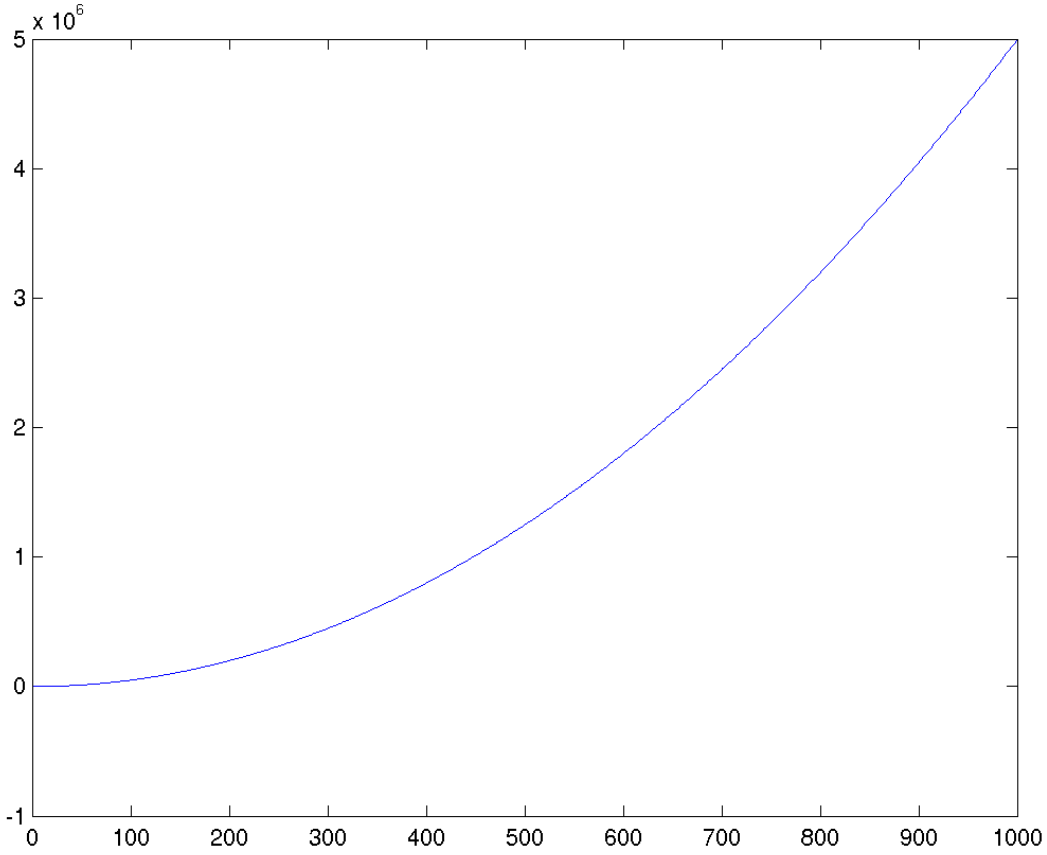
What does $5n^2 - 3n - 6$ look like?

# What does $5n^2 - 3n - 6$ look like?

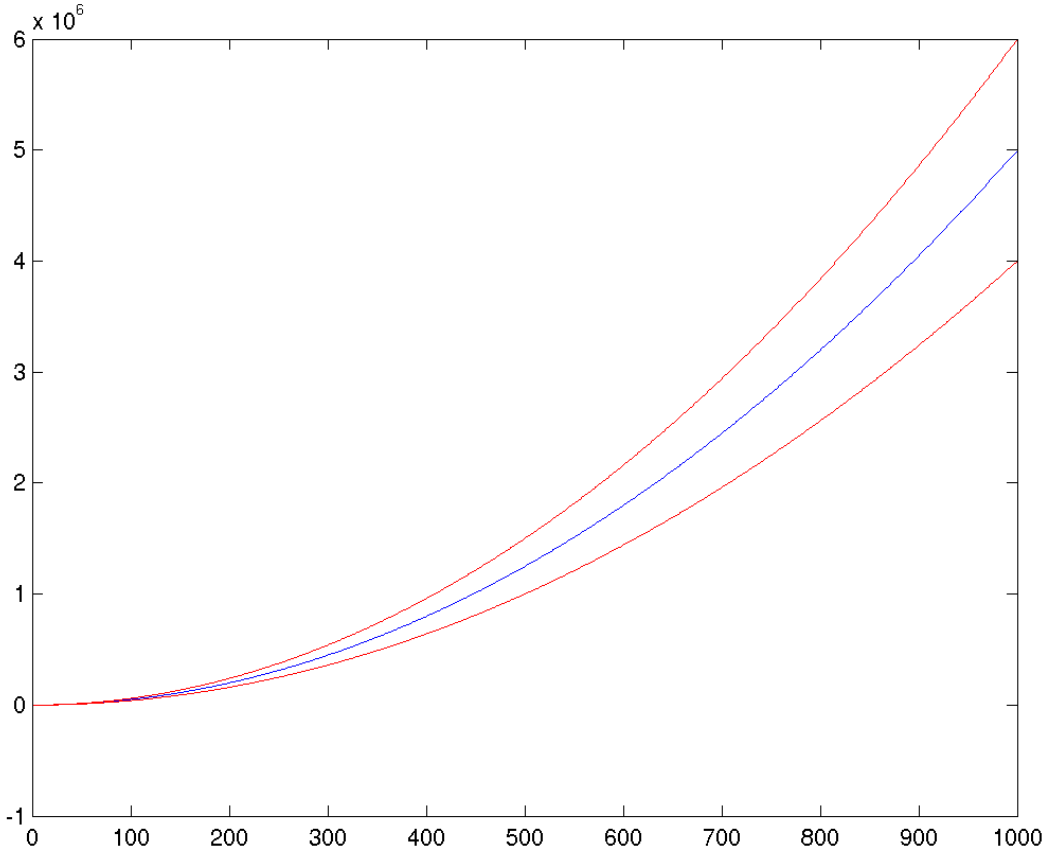


What does $5n^2$ look like?

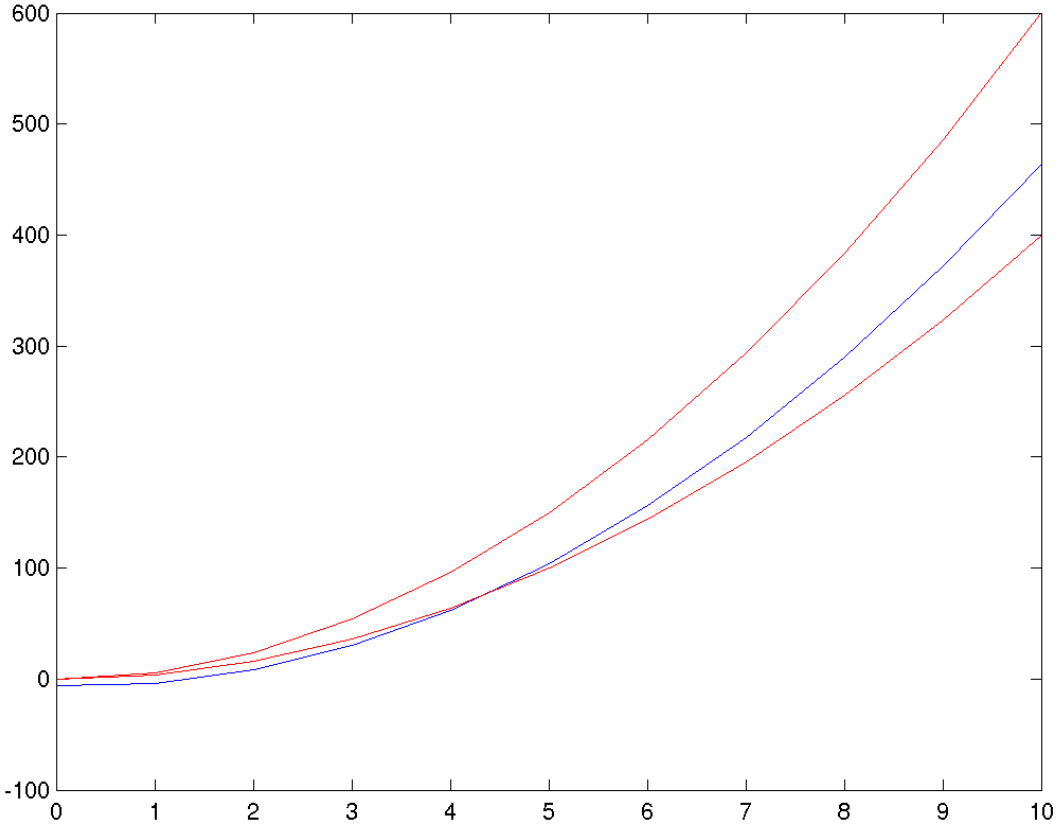
# What does $5n^2$ look like?



# What about $c_1 = 4$ and $c_2 = 6$?



# What $n_0$ do we want?



- Prove $5n^2 - 3n - 6 = \Theta(n^2)$.
- Pick $c_1 = 4$, $c_2 = 6$, and $n_0 = 6$.
- Show that
  - $0 \leq 4n^2$ for all $n \geq 6$ (trivial).
  - $4n^2 \leq 5n^2 - 3n - 6$ for all $n \geq 6$.
  - $5n^2 - 3n - 6 \leq 6n^2$ for all $n \geq 6$.

- $4n^2 \leq 5n^2 - 3n - 6$ for all $n \geq 6$:

$$4n^2 \leq 5n^2 - 3n - 6$$

$$0 \leq n^2 - 3n - 6 = n(n-3) - 6$$

$$6 \leq \underbrace{\underbrace{n}_{\geq 6}\underbrace{(n-3)}_{\geq 3}}_{\geq 18}$$

- $5n^2 - 3n - 6 \leq 6n^2$ for all $n \geq 6$:

$$5n^2 - 3n - 6 \leq 6n^2$$

$$-6 \leq n^2 + 3n = \underbrace{\underbrace{n}_{\geq 6}\underbrace{(n+3)}_{\geq 9}}_{\geq 54}$$

# Outline

# 1. Asymptotic growth rates

Rank these functions in order of growth:

$$\frac{n}{2} \quad 4^{\lg n} \quad n! \quad \lg n \quad 2^{2^n} \quad (n+1)!$$

$$\left(\frac{100}{99}\right)^n \quad n \lg n \quad n^3 \quad 2^n \quad n \cdot 2^n$$

# Example

$$n \lg n \quad n \quad (\lg n)^{\lg n}$$

# 2. Tradeoffs

The Merge-Sort procedure described in the lectures calls Merge once for each recursive call. Each time memory space of the same size as input is allocated (the $L$ and $R$ arrays). Insertion-Sort, on the other hand, allocates no extra memory, but is an algorithm with quadratic running time. For each case below, justify in *at most* three sentences which algorithm is the asymptotically better one.

1. Memory allocation is done in constant time no matter how much memory is allocated.
2. Memory allocation is done in linear time: allocating something of size $n$ takes $\mathcal{O}(n)$.
3. Memory allocation is done in quadratic time: allocating something of size $n$ takes $\mathcal{O}(n^2)$.

# 3. Foo, Bar **and** Baz

Below the procedures Foo, Bar and Baz are given in pseudocode. For Foo and Bar, answer the following four questions:

1. What does the procedure do?
2. What is the best case running time?
3. What is the average case running time?
4. What is the worst case running time?

Each answer should be justified in one sentence. Answers without justifications give 0 points.

Foo($A$)

```
n ← size(A)
for i ← 1 to ⌊n/2⌋
        do if not A[i] = A[n − i + 1]
                then return False
return True
```

```
Bar(A, z)
1   return Baz(A, 1, size(A), z)

Baz(A, x, y, z)
1   if x > y or (x = y and A[x] ≠ z)
2       then return −1
3   i ← x + ⌊(y−x)/2⌋
4   if A[i] = z
5       then return i
6   if A[i] > z
7       then Baz(A, x, i − 1, z)
8       else Baz(A, i + 1, y, z)
```

# Outline

# invariant

**Pronunciation:** /ɪnˈvɛːrɪənt/

*adjective*

never changing:

*the pattern of cell divisions was found to be invariant*

*noun*

*Mathematics*

a function, quantity, or property which remains unchanged when a specified transformation is applied.

- Oxford English Dictionary

MERGE($A, p, q, r$)

```
n1 ← q − p + 1
n2 ← r − q
create arrays L[1 .. n1 + 1]
and R[1 .. n2 + 1]
for i ← 1 to n1
        do L[i] ← A[p + i − 1]
for j ← 1 to n2
        do R[j] ← A[q + j]
L[n1 + 1] ← ∞
R[n2 + 1] ← ∞
i ← 1
j ← 1
for k ← p to r
        do if L[i] ≤ R[j]
                then A[k] ← L[i]
                     i ← i + 1
                else A[k] ← R[j]
                     j ← j + 1
```

Question: What is the invariant of merge?

# Outline

# Towers of Hanoi

You are given a tower of $n$ disks initially stacked in decreasing order of diameter on one out of three pegs. The goal is to transfer the entire tower to one of the other pegs, moving one disk at a time and never placing a larger disk onto a smaller one. Devise a divide-and-conquer algorithm, analyze its running time and write and solve the recurrence.

- First thing: we should use divide-and-conquer. This means, put in words, that the way to move a big tower is by moving a smaller tower.

- First thing: we should use divide-and-conquer. This means, put in words, that the way to move a big tower is by moving a smaller tower.
- Algorithm: to move a tower of $n$ disks from peg $A$ to peg $C$, move the smaller tower consisting of $n-1$ disks to peg $B$, move the $n$th disk to peg $C$, and move the tower consisting of $n-1$ disks on peg $B$ to peg $C$. If $n=1$, skip the recursive step.

- First thing: we should use divide-and-conquer. This means, put in words, that the way to move a big tower is by moving a smaller tower.
- Algorithm: to move a tower of $n$ disks from peg $A$ to peg $C$, move the smaller tower consisting of $n-1$ disks to peg $B$, move the $n$th disk to peg $C$, and move the tower consisting of $n-1$ disks on peg $B$ to peg $C$. If $n=1$, skip the recursive step.
- **In pseudocode**:

```
Towers-Of-Hanoi(n)
if n = 1
    then move disk to target peg
    else Towers-Of-Hanoi(n − 1)
         move nth disk to target peg
         Towers-Of-Hanoi(n − 1)
```

## Complexity

- What type of complexity?

## Complexity

- What type of complexity?
- So, how do we get the recurrence $T(n)$?

## Complexity

- What type of complexity?
- So, how do we get the recurrence $T(n)$?
- The call with $n = 1$ clearly requires 1 operation on line 2.

## Complexity

- What type of complexity?
- So, how do we get the recurrence $T(n)$?
- The call with $n = 1$ clearly requires 1 operation on line 2.
- A call with $n > 1$ requires $T(n-1)$ operations on line 3, 1 op on line 4 and $T(n-1)$ ops on line 5

## Complexity

- What type of complexity?
- So, how do we get the recurrence $T(n)$?
- The call with $n = 1$ clearly requires 1 operation on line 2.
- A call with $n > 1$ requires $T(n-1)$ operations on line 3, 1 op on line 4 and $T(n-1)$ ops on line 5

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(n-1) + 1 & \text{if } n > 1 \end{cases}$$

## How to solve it

- Start by some unfolding, and compute for small values.

## How to solve it

- Start by some unfolding, and compute for small values.
- Guess closed form solution: $T_c(n) = 2^n - 1$

## How to solve it

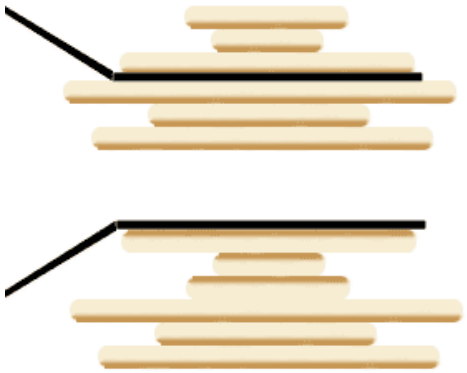
- Start by some unfolding, and compute for small values.
- Guess closed form solution: $T_c(n) = 2^n - 1$
- Prove it by induction:
  - Basecase (n=1): $T(1) = 1 = 2^1 - 1 = T_c(1)$

## How to solve it

- Start by some unfolding, and compute for small values.
- Guess closed form solution: $T_c(n) = 2^n - 1$
- Prove it by induction:
  - Basecase (n=1): $T(1) = 1 = 2^1 - 1 = T_c(1)$
  - Induction step: assume $T(n) = T_c(n)$.
    Then $T(n+1) = 2T(n) + 1 = 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1 = T_c(n)$

# Pancake sorting

Pancake sorting is a sorting problem where *prefix reversal* is the only means with which you can alter the list to sort.



Devise and analyse a simple algorithm for pancake sorting. Do an exact worst case analysis in terms of the number of prefix reversals, and give an asymptotic upper bound, given that the reverse-prefix-operation takes $O(n)$ time.

# Pancake sorting

Divide-and-conquer approach:

Pancake-Sort($s$)

```
if size(s) < 1
    then return
    else  i ← indexOfMax(s)
          Flip(s[1, ..., i])
          Flip(s)
          Pancake-Sort(s[1, ..., size(s)])
```

# Pancake sorting

Iterative approach:

Pancake-Sort($s$)

```
for i ← size(s) down to 2
        do j ← indexOfMax(s)
            Flip(s[1, ..., j])
            Flip(s[1, ..., i])
```