北京航空航天大學
BEIHANG UNIVERSITY

Introduction to Computer Science and Programming

Lecture 6

Sebastian Wandelt (小塞)

Beihang University

Outline

- Recap
- Algorithm analysis

Recap

What do you remember about last lecture?

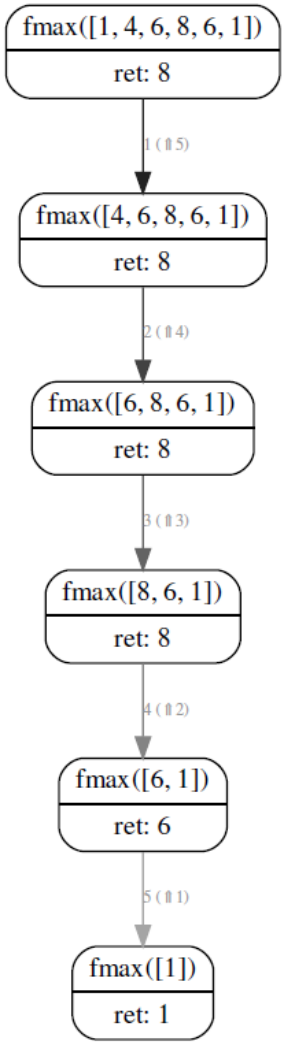
Task

- Please write a recursive function for obtaining the maximum of a list of numbers

Maximum: Source code and call graph

```
def fmax(L):
    if len(L)==1:
        return L[0]
    else:
        submax=fmax(L[1:])
        if L[0]>submax:
            return L[0]
        else:
            return submax

print(fmax([1,4,6,8,6,1]))
```



Task

- Please write a class for a circle. A circle (in a 2D plane) is defined by a point and a radius.
- Implement the constructor and two member functions:
  - One function computes the area of the circle
  - The second function checks whether a point x,y is located inside the circle

Midterm exam

Midterm exam

- Reason:
  - Divergence in performance; partial inability to write simplest programs
- Time: November 9th, during the lecture slot
- Duration: 45-90 minutes (tbd)
- Grading: 40% of your final grade
- Exam style:
  - Closed book
  - Paper&pencil
- Topics
  - Trace Python programs
  - Write Python programs for a specific problem
  - Finding bugs in a given Python program

Algorithm Analysis

Note

- In the following we will often use pseudocode!
- What is pseudocode?
  - An abstraction from software engineering details
  - High similarities with C/C++/Python
  - Does not deal with modularity, error handling, etc.
  - Sometimes uses shortcuts, which can cannot be executed directly, e.g., English sentences
  - Covers the **essence** of an algorithm
    - Essence is the core/critical parts/cannot be removed

Real (Python) code vs. pseudocode

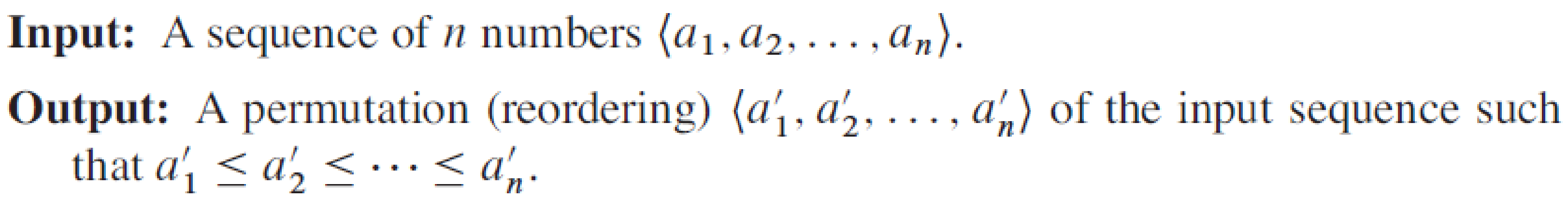
```python
def hanoi(n,source,destination,temp):
    if n!=0:
        hanoi(n-1,source,temp,destination)
        print("Move from ",source," to ",destination)
        hanoi(n-1,temp,destination,source)

hanoi(3,"a","b","c")
```

```
Hanoi (n,source, destination and some kind of temp variable)
If n is not yet 0, the do the following:
1)  Call recursively Hanoi (n-1,source,temp,destination)
2)  Print that we make  a move from source to destination
3)  Call recursively Hanoi (n-1,temp,destination,source)
```

The formal problem of sorting

- Sorting is a fundamental operation in CS



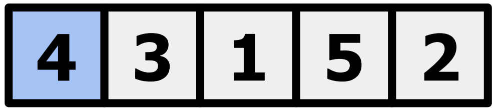
Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

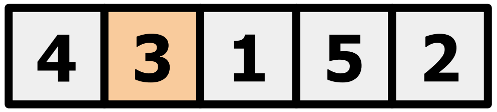
The formal problem of sorting

- How to solve this problem?

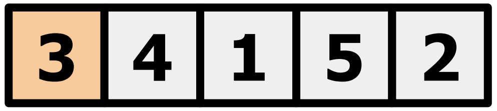
Insertion sort



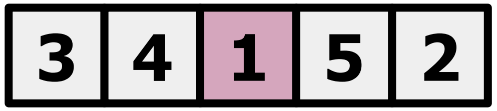
Let’s sort an unsorted list of numbers **`A`**. The sublist **`A[0:0]`** is trivially sorted.



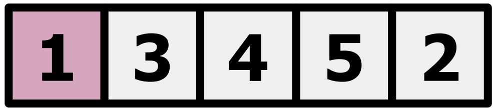
Look at the second element, **`A[1]`**.



Insert the element into a new position such that the sublist **`A[0:1]`** is sorted.

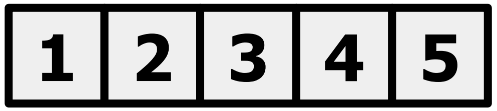


Now look at the third element, **`A[2]`**.



Insert it such that the sublist **`A[0:2]`** is sorted.

. . .



The entire array **`A[0:4]`** is sorted.

Let’s implement this in Python

Insertion sort: Python code

```python
def insertion_sort(A):
    for i in range(0,len(A)):
        cur_value = A[i]
        j = i - 1
        while j >= 0 and A[j] > cur_value:
            A[j+1] = A[j]
            j = j - 1
        A[j+1] = cur_value
```

Insertion sort: Pseudocode

```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let j = i - 1
    while j ≥ 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

Can you spot the similarities with Python?

Major questions regarding the algorithm

Question 1 How do we prove this algorithm always sorts the input list?

Question 2 How efficiently does this algorithm sort the input list?

Short recap: Proofs

- How to prove the following statement?
  - For each natural number **n, we have** that

$$\mathbf{1 + 2 + \ldots + n = n(n + 1)/2}$$

Short recap: Proofs

Proof by induction:

- Base case: n=1
  - 1 = 1(1+1)/2
- Inductive case: n -> n+1
  - To show: 1 + 2 + ... + n + (n + 1) = (n + 1)((n + 1) + 1)/2
  - **IMPORTANT**: We can use 1 + 2 + ... + n = n(n + 1)/2 !
  - Steps:

    1 + 2 + ... + n + (n + 1)

    = n(n + 1)/2 + (n + 1)

    = (n + 1) * (n + 2)/2

    = (n + 1) * ((n + 1) + 1)/2

Proving Correctness

Algorithms often initialize, modify, or delete new data.

- In the case of insertion sort, it might be challenging for an untrained observer to formalize the notion of correctness since the manner in which the algorithm behaves depends on the input list.
- Is there a way to prove the algorithm works, without checking it for all (infinitely many) input lists?
- **To reason about the behavior of algorithms, it often helps to look for things that *don't* change.**
  - Notice that insertion sort maintains a sorted sublist, the length of which grows each iteration.
- This unchanging property is called an **invariant**.

Where is an invariant here?

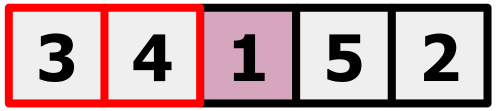
```
algorithm insertion_sort(list A):
  for i = 0 to length(A)-1:
    let cur_value = A[i]
    let j = i - 1
    while j ≥ 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

Proving Correctness

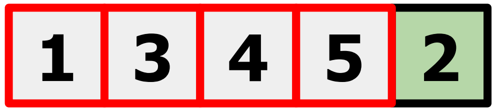
For example, an **invariant** of the outer for-loop of insertion sort: At the start of iteration i of the outer for-loop, the first i elements of the list are sorted.

Sanity checks:

At the start of the third iteration (i.e. the iteration when i = 2), the first 2 elements of the list are sorted. True.



At the start of the fifth iteration (i.e. the iteration when i = 4), the first 4 elements of the list are sorted. True.



Proving Correctness

Less formally...

1. At the start of the first iteration, the first element of the array is sorted.
2. By construction, the ith iteration puts element **`A[i]`** in the right place.
3. At the start of the i = length(A)th iteration (aka the end of the algorithm), the first length(A) elements are sorted.

Proving Correctness

More formally (rigorously) …

Outer invariant (for-loop): At the start of iteration i of the outer for-loop, the first i elements of the list are sorted.

Inner invariant (while-loop): At the start of iteration j of the inner while-loop, `A[0:j,j+2:i]` contains the same elements as the original sublist `A[0:i-1]`, still sorted, such that all of the values in the right sublist `A[j+2:i]` are greater than `cur_value`.

Proving Correctness

The theorem follows a consistent format:

Initialization:
The loop invariant starts out as true.

Maintenance:
If the loop invariant is true at step i, then it’s true at step i+1.

Termination:
If the loop invariant is true at the end of the algorithm, this tells you something about what you’re trying to prove.

Insertion sort

Question 1 How do we prove this algorithm always sorts the input list?

Question 2 How efficiently does this algorithm sort the input list?

Analyzing runtime

```
algorithm insertion_sort(list A):
  for i = 1 to length(A):
    let cur_value = A[i]
    let j = i - 1
    while j > 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

Total work: ???

Analyzing runtime

```
algorithm insertion_sort(list A):
  for i = 1 to length(A):
    let cur_value = A[i]
    let j = i - 1
    while j > 0 and A[j] > cur_value:
      A[j+1] = A[j]
      j = j - 1
    A[j+1] = cur_value
```

O(n) work per iteration

O(n) iterations

Total work: $O(n^2)$

The Big-O notation

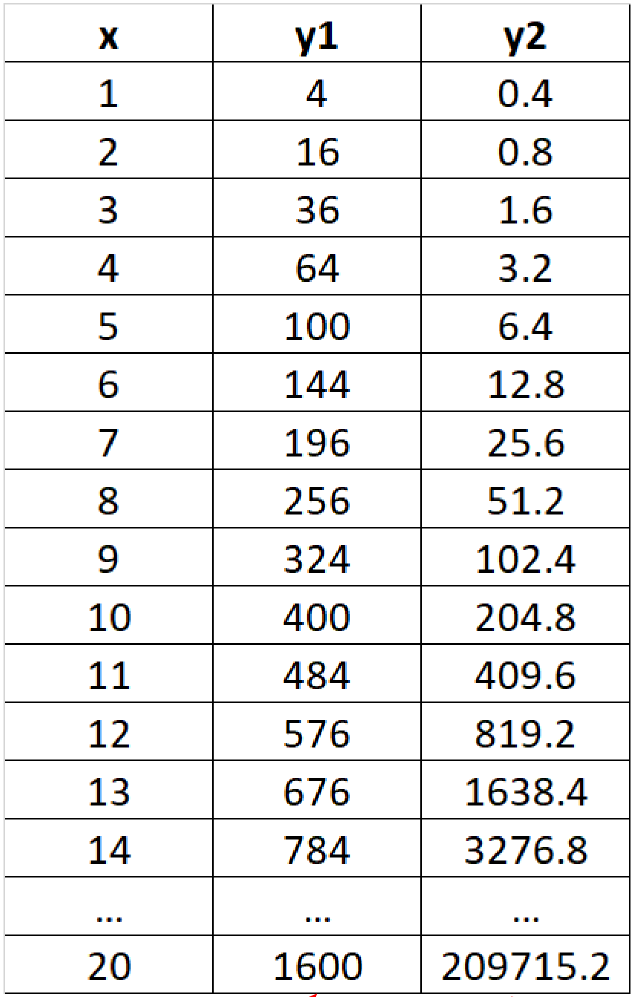
Which series grows faster?

x	y1	y2
1	4	0.4
2	16	0.8
3	36	1.6
4	64	3.2
5	100	6.4
6	144	12.8
7	196	25.6

Which series grows faster?

x	y1	y2
1	4	0.4
2	16	0.8
3	36	1.6
4	64	3.2
5	100	6.4
6	144	12.8
7	196	25.6
8	256	51.2
9	324	102.4
10	400	204.8
11	484	409.6
12	576	819.2
13	676	1638.4
14	784	3276.8
…	…	…
20	1600	209715.2

Which series grows faster?

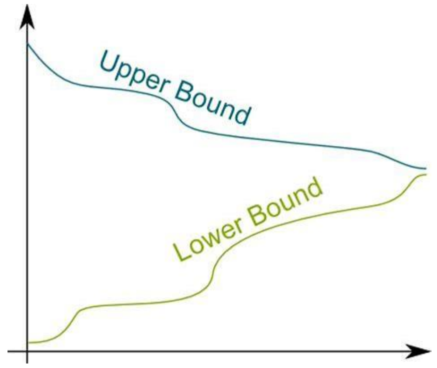


x	y1	y2
1	4	0.4
2	16	0.8
3	36	1.6
4	64	3.2
5	100	6.4
6	144	12.8
7	196	25.6
8	256	51.2
9	324	102.4
10	400	204.8
11	484	409.6
12	576	819.2
13	676	1638.4
14	784	3276.8
…	…	…
20	1600	209715.2

4*(x**2) **0.2*(2**x)**

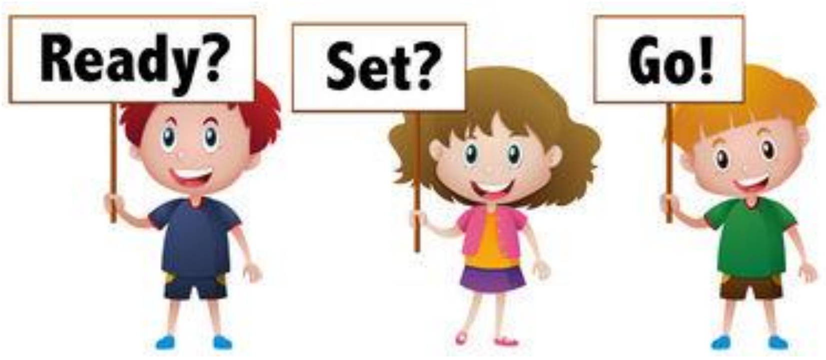
Big-O Notation

Big-O notation is a mathematical notation for providing an upper-bound of a function's rate of growth. Informally, it can be determined by ignoring constants and non-dominant growth terms.



Examples

- $n + 137 = O(n)$
- $3n + 42 = O(n)$
- $n^2 + 3n - 2 = O(n^2)$
- $n^3 + 10n^2 \log n - 15n = ??$
- $2^n + n^2 = ??$



Big-O Notation

Formally speaking, let f, g: $N \rightarrow N$.

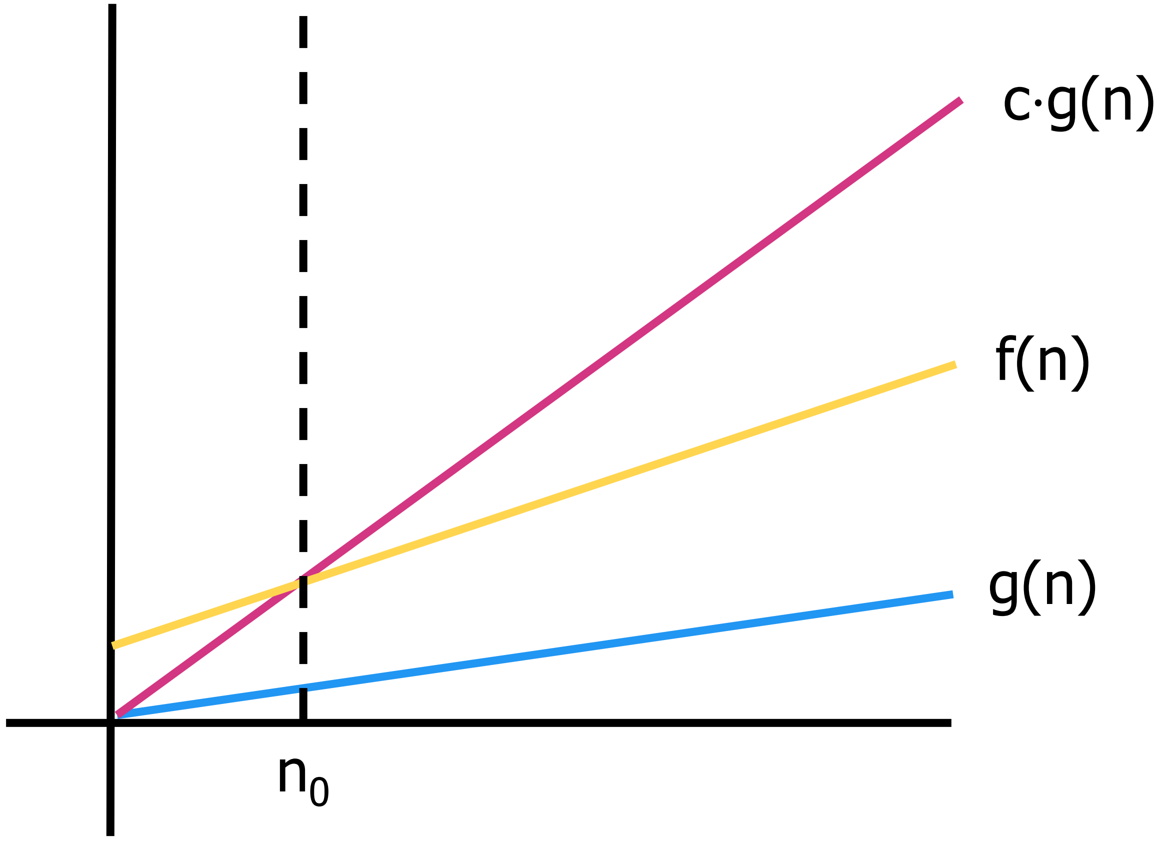
Then **f(n) = O(g(n)) iff**

$$\exists n_0 \in \mathbf{N}, c \in \mathbf{R}.$$
$$\forall n \in \mathbf{N}.$$
$$(n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$$

Intuitively, this means that f(n) is upper-bounded by g(n) aka f(n) is "at most as big as" g(n).

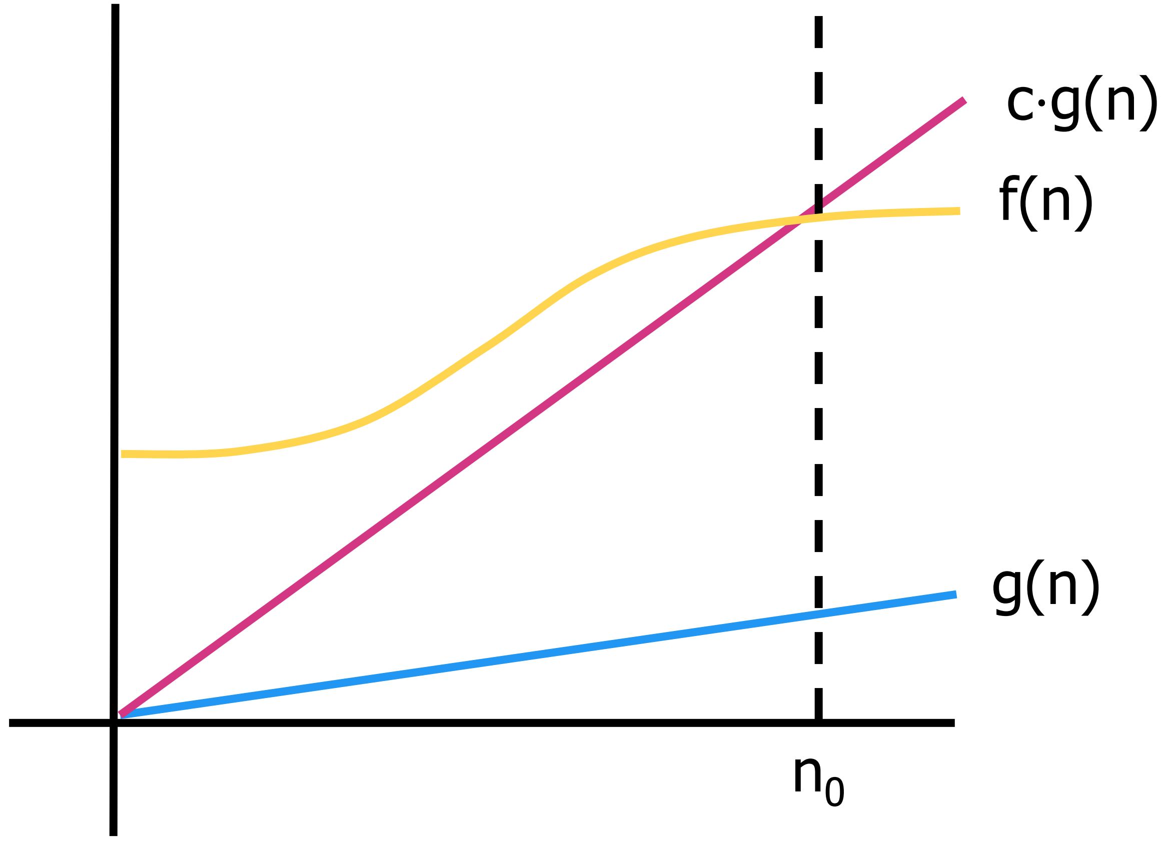
Big-O Notation

$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$



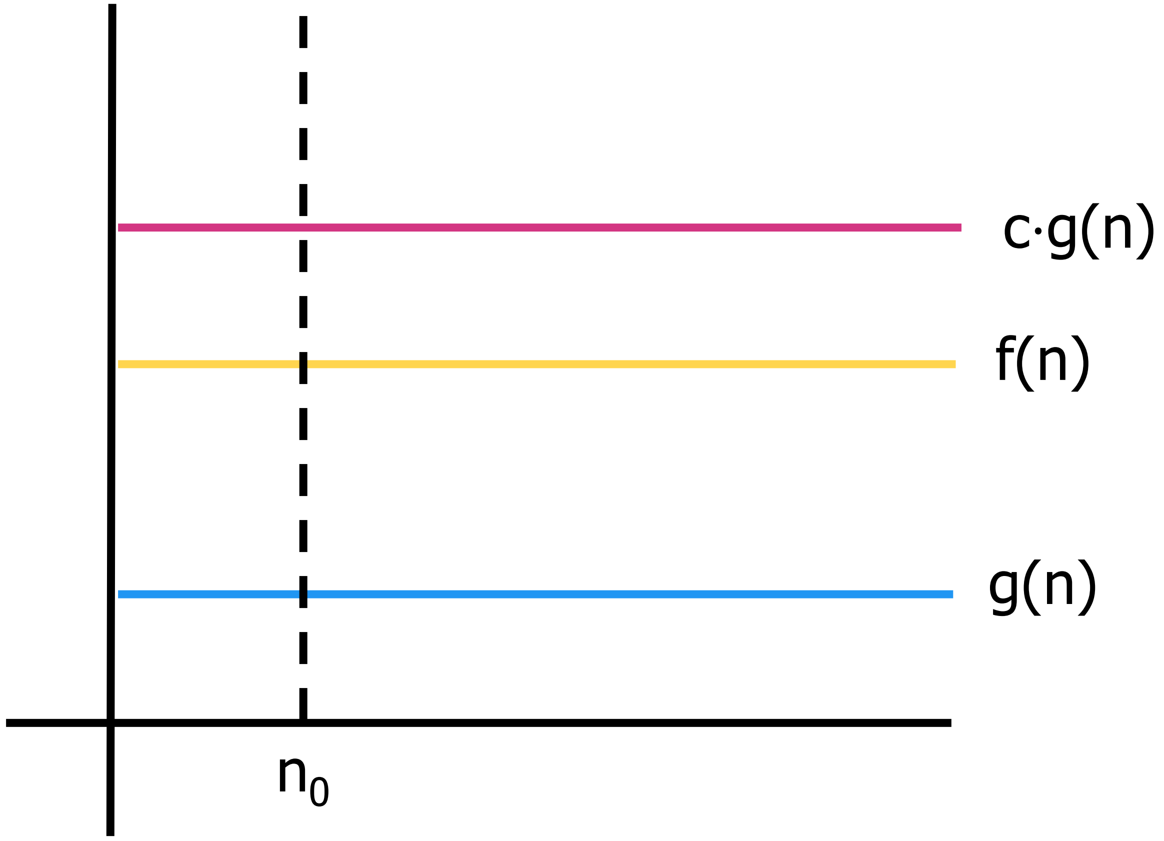
Big-O Notation

$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$



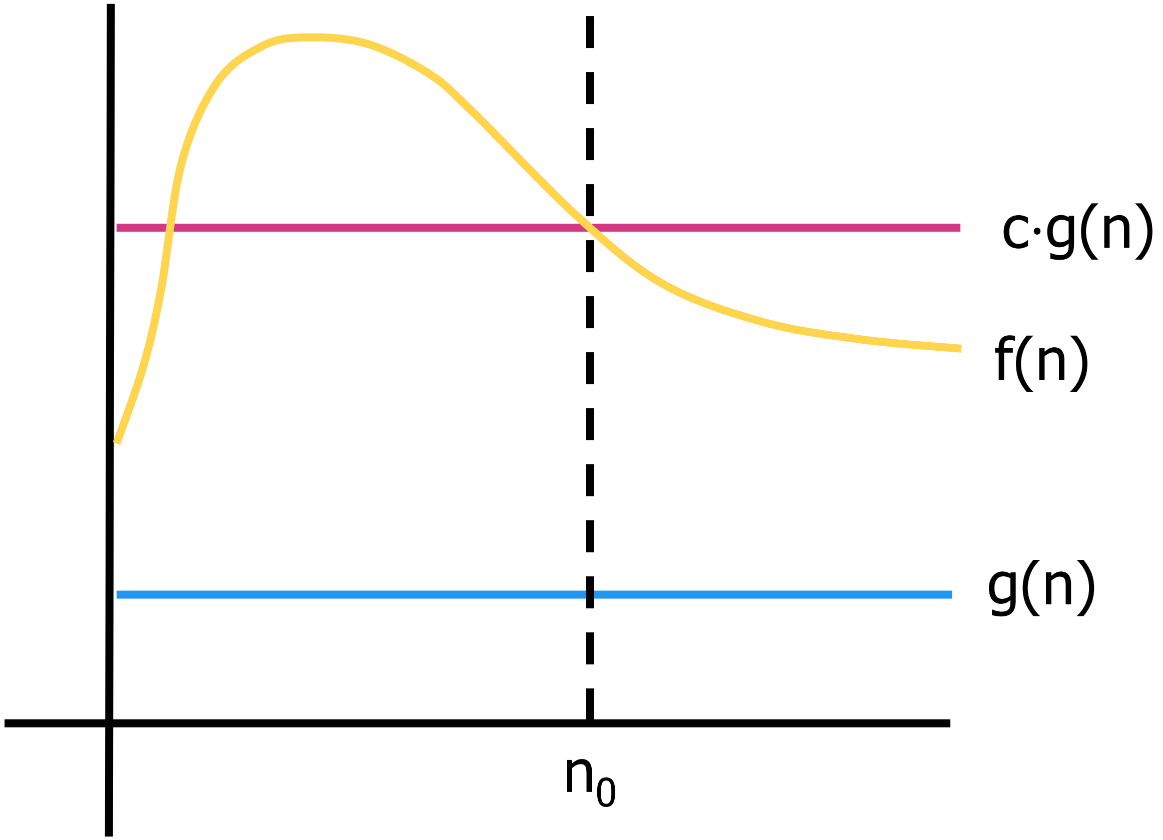
Big-O Notation

$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$



Big-O Notation

$f(n) = O(g(n)) \quad \text{iff} \quad \exists n_0 \in N, c \in R. \quad \forall n \in N. \quad (n \geq n_0 \rightarrow f(n) \leq c \cdot g(n))$



Big-O Notation

To prove $f(n) = O(g(n))$, show that there exists a c and n_0 that satisfies the definition.

Suppose $f(n) = n$ and $g(n) = n \log n$. We prove that $f(n) = O(g(n))$.

Consider the values $c = 1$ and $n_0 = 2$. We have $n \leq cn \log n$ for $n \geq n_0$ since n is positive and $1 \leq \log n$ for $n \geq 2$.

To prove $f(n) \neq O(g(n))$, proceed by contradiction.

Suppose $f(n) = n^2$ and $g(n) = n$. We prove that $f(n) \neq O(g(n))$.

Suppose there exists some c and n_0 such that for all $n \geq n_0$, $n^2 \leq cn$. Consider $n = \max\{c, n_0\} + 1$. Then $n \geq n_0$, but we have $n > c$, which implies that $n^2 > cn$. Contradiction!

The Big-Ω notation

(omega)

Big-Ω Notation

Let f, g: $N \rightarrow N$.

Then $f(n) = \Omega(g(n))$ iff

$\exists n_0 \in N, c \in R.$

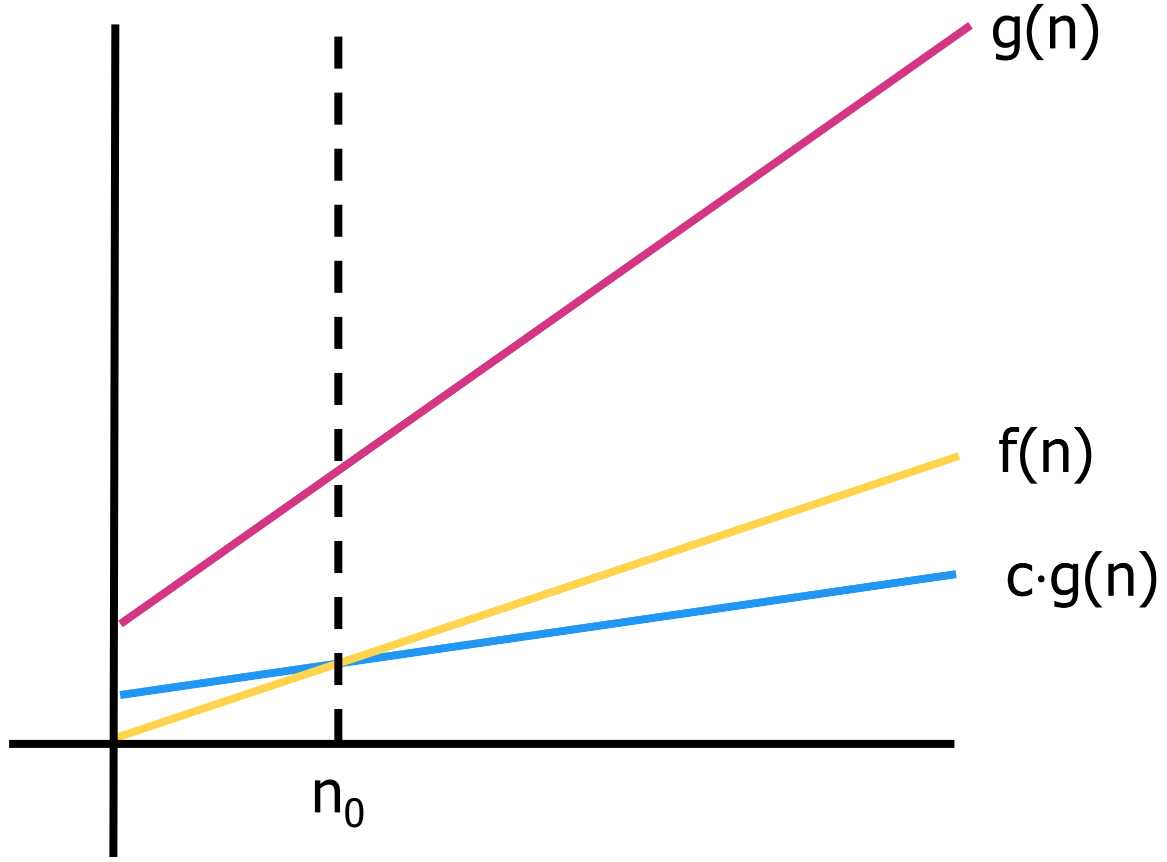
$\forall n \in N.$

$(n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$

Intuitively, this means that f(n) is lower-bounded by g(n) aka f(n) is “at least as big as” g(n).

Big-Ω Notation

$f(n) = \Omega(g(n))$ iff $\exists n_0 \in N, c \in R.\ \ \forall n \in N.\ \ (n \geq n_0 \rightarrow f(n) \geq c \cdot g(n))$



The Big-Θ notation
(theta)

Big-Θ Notation

$f(n) = \Theta(g(n))$ iff both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

More verbosely, let $f, g\colon N \to N$.

Then $f(n) = \Theta(g(n))$ iff

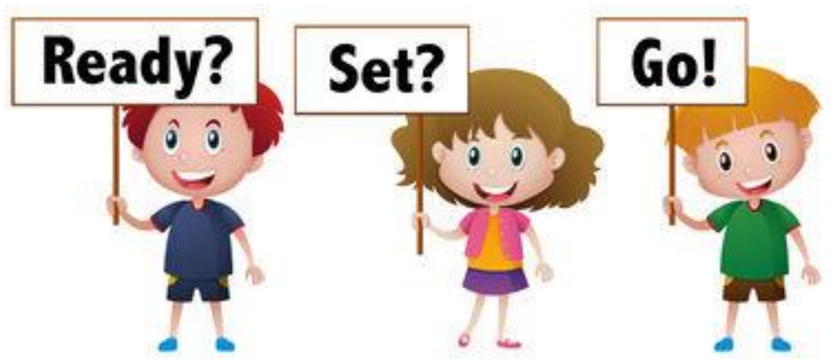
$\exists n_0 \in N$, c_1 and $c_2 \in R$.

$\forall n \in N$.

$(n \geq n_0 \to c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n))$

Intuitively, this means that f(n) is lower and upper-bounded by g(n) aka f(n) is “the same as” g(n).

Best case vs. Worst case



1	2	3	4	5	...	n

n	...	5	4	3	2	1

9	7	n	...	1	4	2

Total work: $O(n)$ or $O(n^2)$ or $\Omega(n)$ or $\Omega(n^2)$?

Best case vs. Worst case

The worst-case runtime of insertion sort is $\Theta(n^2)$.

The best-case runtime of insertion sort is $\Theta(n)$.

Usually, we care more about the worst-case time.

Why?

Best case vs. Worst case

The worst-case runtime of insertion sort is $\Theta(n^2)$.

The best-case runtime of insertion sort is $\Theta(n)$.

Usually, we care more about the worst-case time.

We do not know the user's input at runtime, so we need to expect the worst-case.

It's acceptable, albeit not entirely precise, to say the runtime of insertion sort is $\Theta(n^2)$.

Thank you very much!

If you have any questions, please get in touch with me: wandelt@buaa.edu.cn