Session 36: Growth of Functions

- Efficiency of Algorithms
- Characterizing growth of functions
- Big-O Notation

How much detail is needed?

When determining efficiency of algorithms we have two extremes

How much detail is needed?

When determining efficiency of algorithms we have two extremes

1. Precise count of everything involved
   - computer instructions, disk accesses, …
   - as a function of the problem size

   ➜ inconvenient, not always well-defined

How much detail is needed?

When determining efficiency of algorithms we have two extremes

1. Precise count of everything involved
   - computer instructions, disk accesses, …
   - as a function of the problem size

   ➜ inconvenient, not always well-defined
2. “it took a few seconds on my laptop”
   - what if size doubles?

   ➜ not sufficiently informative

Example

Assume it took ***s* seconds** to find the maximum among ***n* unsorted items**

How to predict the time required to find the maximum among
2*n*, 3*n*, or *m* items?

$2n$ items $\rightarrow$ $2s$ seconds

$3n$ items $\rightarrow$ $3s$ seconds

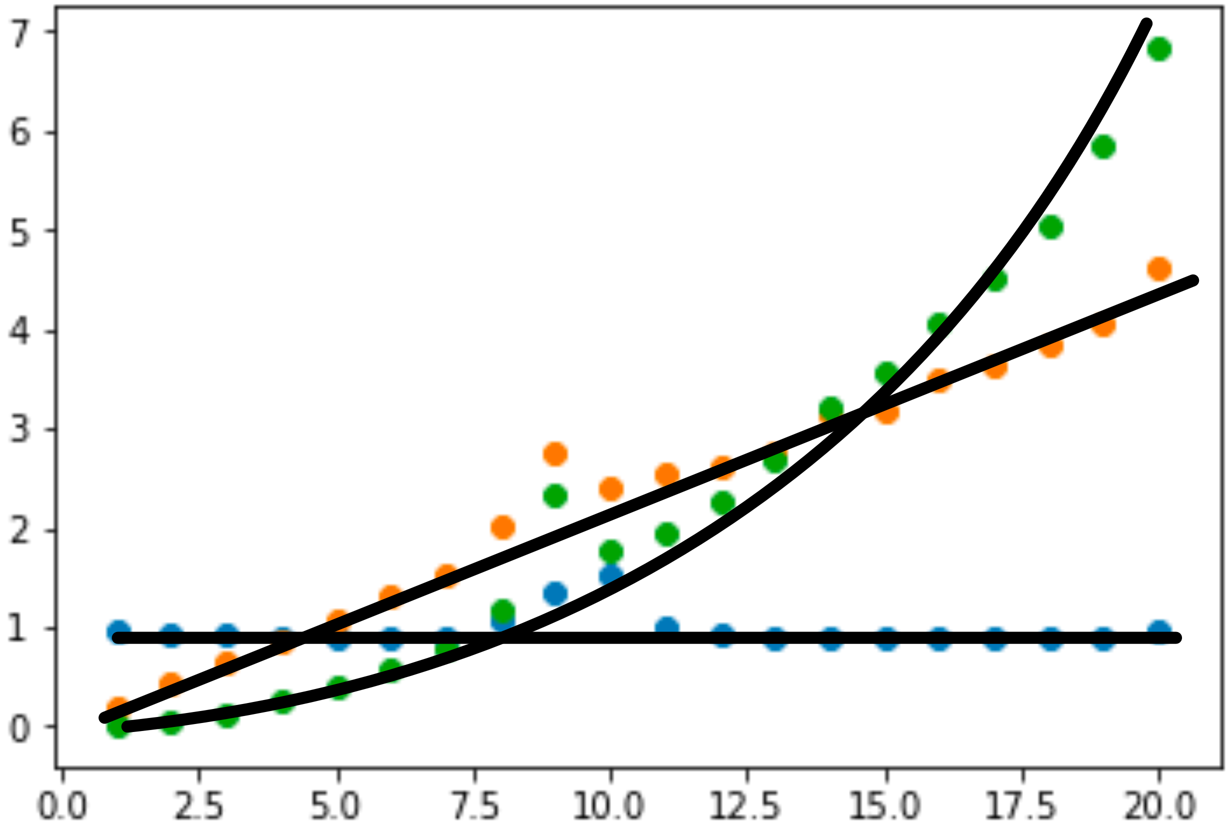
m items $\rightarrow$ $\frac{m}{n} s$ seconds

Example

- Assume you run the following algorithm

```
Procedure sort_tasks(n: integer)
   Create a list of 3000 random numbers and sort it using bubble sort (task 1)
   Create n lists of length 1500 and sort them using bubble sort (task 2)
   Create a list of length 400*n and sort it using bubble sort (task 3)
```

Experiment



Time spent on each task

- We measure how much time is spent on each task depending on n
  - n = 1, 2, 3, …, 20
- It is approximately
  - 1000 milli-seconds for task 1, independent of n
  - 200 n milli-seconds for task 2
  - 1.5 n^2 milli-seconds for task 3
- So in total we can estimate the time spent as

$$f(n) = 1.5\, n^2 + 200\, n + 1000$$

Example

Let $f(n) = 1.5\, n^2 + 200\, n + 1000$ be a function to estimate the time to solve problem of size n

Let $g(n) = 1.5\, n^2$, $h(n) = 200\, n$, $t(n) = 1000$

for small n: $t(n)$ most significant
then: $h(n)$ takes over
but ultimately: **only $g(n)$ is relevant**

Observation

Let $f(n)$ estimate the time to solve problem of size n

if

$$f(n) = g(n) + h(n) + \ldots + t(n)$$

for functions $g, h, \ldots, t$: $\mathbf{N} \to \mathbf{R}$

then the "ultimately largest" of $g, h, \ldots, t$ determines f's behavior when n gets large

Observation

Let g(n) be f(n)’s “ultimately most relevant part”

Then f(n)’s growth rate is **independent of multiplicative constants** in g(n):

$$\frac{g(m)}{g(n)} = \frac{cg(m)}{cg(n)}$$

Consequence

When considering a runtime function f(n)

- Focus on part that grows "fastest" (for $n \to \infty$)
- Forget about multiplicative constants

- We do not care about small values of n
- We do not care about the absolute value, but about growth

Big-*O* Notation

Definition: Let *f* and *g* be functions from the set of integers or the set of real numbers to the set of real numbers. We say that ***f*(*x*) is *O*(*g*(*x*))** if there are constants *C* and *k* such that

$$\left| f(x) \right| \leq C \left| g(x) \right|$$

whenever $x > k$.

- This is read as "*f*(*x*) is big-*O* of *g*(*x*)" or "*g* asymptotically dominates *f*.
- The constants C and k are called **witnesses** to the relationship *f*(*x*) is *O*(*g*(*x*)).
- Only one pair of witnesses is needed.

Remark on Notations:

We have been using for function $f : \mathbb{R} \to \mathbb{R}$ the notation $f(x)$ is $O(g(x))$

When we consider functions $f : \mathbb{N} \to \mathbb{N}$ we will alternatively use $f(n)$ is $O(g(n))$

Summary

- Big-O notation
  - Abstract from details of how a function grows
  - Considers the fastest growing part for large values
- Used to characterize the efficiency of algorithms