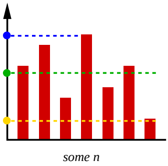
Algorithms and Datastructures

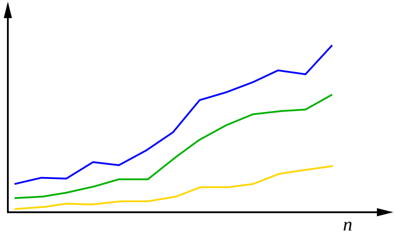
Lecture 2

Manfred Jaeger

AALBORG UNIVERSITET

Recap





$T(n) =$ *worst-case runtime for instances of size n*

Runtime is measured in terms of computation steps under the **RAM (random-access machine)** model:

- Sequential computation, no parallelism
- The following operations take one time unit (step):
  - reading/writing the value of a variable from/to memory
  - performing a simple arithmetic operation (addition, multiplication, exponentiation,...)
  - testing a simple Boolean condition

```
INSERTIONSORT(I)
   /* n = I.length */
1  for j = 2..n do
2      key=I[j]
3      i = j − 1
4      while i > 0 and I[i] > key do
5          I[i + 1] = I[i]
6          i = i − 1
7      I[i + 1] = key
```

Line No.	Number of times executed: Best	Worst	Average
1	$n-1$	$n-1$	$n-1$
2	$n-1$	$n-1$	$n-1$
3	$n-1$	$n-1$	$n-1$
4	$n-1$	$2+3+\ldots+(n+1)$	$\frac{2+3+\ldots+(n+1)}{2}$
5	0	$1+2+\ldots+n$	$\frac{1+2+\ldots+n}{2}$
6	0	$1+2+\ldots+n$	$\frac{1+2+\ldots+n}{2}$
7	$n-1$	$n-1$	$n-1$
total	$\sim n$	$\sim n^2$	$\sim n^2$

Also important:

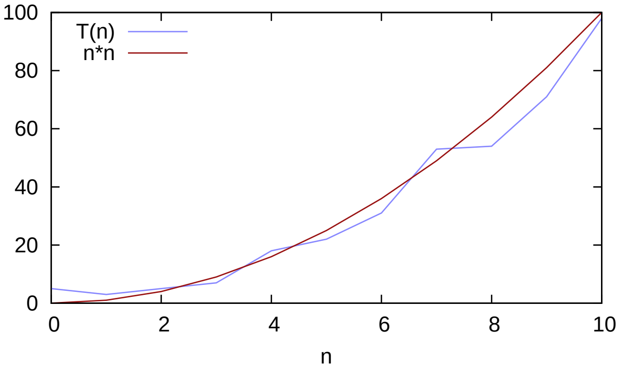
$$S(n) = \textit{worst-case memory consumption for instances of size } n$$

- for some applications critical: can run out of space faster than out of time!
- not the main concern for most of the “classical” problems we consider
- often a tradeoff: fast algorithm that uses much space vs. slower algorithm that uses less space

Growth of Functions

For insertion sort we found: $T(n) \sim n^2$.

What does that mean exactly? (Hypothetical) plot of precise values:



- $T(n)$ approximately follows n^2, but
  - sometimes above, sometimes below n^2.

Upper/Lower Bound

Divide characterization of $T(n)$'s growth into two parts:

- Lower bound
- Upper bound (Most attention on this – in line with worst-case perspective!)

Constants Don't Matter ...

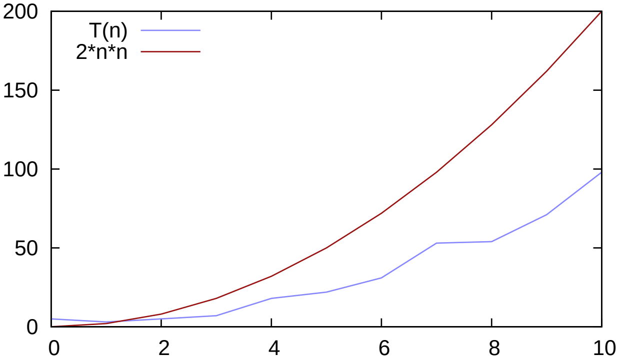
$T(n)$ characterizes the actual runtime (CPU-cycles, seconds, ...) only up to a constant factor.

☞ Instead of comparing $T(n)$ to n^2, we may as well compare

- $c \cdot T(n)$ to n^2, or
- $T(n)$ to $c \cdot n^2$

(c any positive constant).

Comparing $T(n)$ to $2 \cdot n^2$:



- $2 \cdot n^2$ is an upper bound for $T(n)$ except for $n = 0$ and $n = 1$
- $c \cdot n^2 = 0$ for all $c > 0$, so the "problem" at $n = 0$ can not be fixed by any larger multiplicative factor c.
- We are interested in time complexity as size of input instances becomes really large (*asymptotic behavior*). Just ignore violation of upper bound by a few n values at the beginning!

- $T(n)$: any function (but usually the worst-case running time of an algorithm)
- $f(n)$: any function

$$T(n) = O(f(n)) \Leftrightarrow \text{there exist constants } c > 0 \text{ and } n_0 > 0 \text{, so that for all } n \geq n_0\text{:}$$
$$T(n) \leq c \cdot f(n)$$

- $T(n)$: any function (but usually the worst-case running time of an algorithm)
- $f(n)$: any function

$$\color{red}{T(n) = O(f(n))} \Leftrightarrow \text{there exist constants } c > 0 \text{ and } n_0 > 0 \text{, so that for all } n \geq n_0\text{:} \quad T(n) \leq c \cdot f(n)$$

Also: $O(f(n))$ is the *set* of all functions $T(n)$ with $T(n) = O(f(n))$.

(and therefore: $T(n) = O(f(n))$ can also be written as $T(n) \in O(f(n))$)

☞ In the following all functions $T(n), f(n), g(n), \ldots$ are assumed to be *non-negative*.

If $T(n) \in O(f(n))$, then for every positive constant d: $d \cdot T(n) \in O(f(n))$.

Robustness w.r.t. time measure

big-O characterizations of time complexity remain the same, independent of whether one measures the actual time of a computation as

- number of execution steps of RAM
- number of clock cycles on (single processor) computer
- number of seconds on (single processor) computer
- ...

Robustness w.r.t. accuracy of counting RAM execution steps

When counting the number of RAM execution steps, it does not matter whether we count one execution of

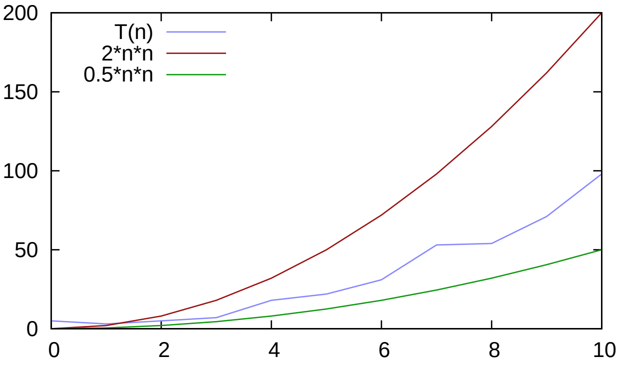
4 **while** $i > 0$ *and* $l[i] > key$ **do**

as requiring 1,2,3, or any constant number of steps.

$T(n) = \Omega(f(n)) \quad \Leftrightarrow \quad$ there exist constants $c > 0$ and $n_0 > 0$, so that for all $n \geq n_0$: $T(n) \geq c \cdot f(n)$

$T(n) = \Theta(f(n)) \quad \Leftrightarrow \quad T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Example: $T(n) = \Theta(n^2)$:



An equation of the form

$$T(n) = h(n) + O(f(n)) \qquad (h(n), f(n) \text{ some functions})$$

means: there exists $g(n) \in O(f(n))$, so that

$$T(n) = h(n) + g(n).$$

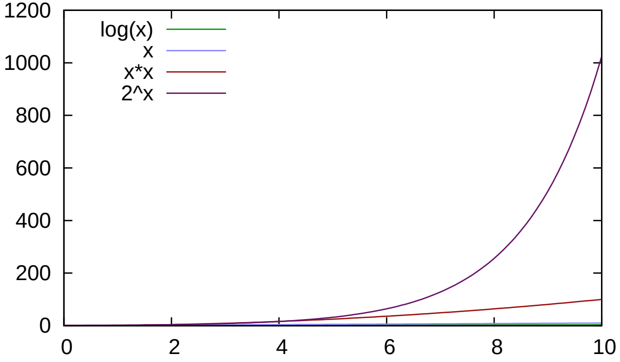
(the same for $T(n) = h(n) + \Omega(f(n))$ and $T(n) = h(n) + \Theta(f(n))$)

Rules and Calculations

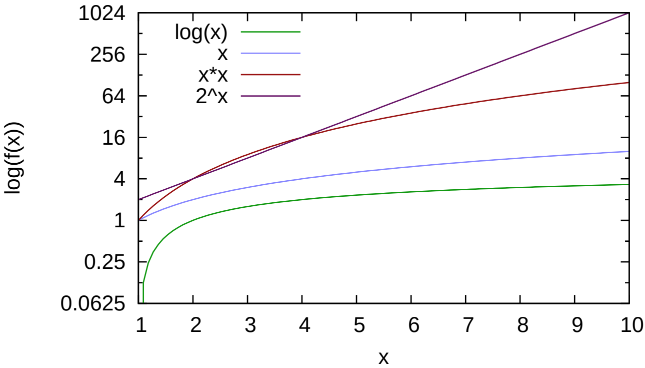
If $f(n) \leq g(n)$ stands for $f(n) \in O(g(n))$, then the following is true:

- A $n \leq n^2 \leq lg\, n \leq n^3 \leq 2^n$
- B $n \leq lg\, n \leq n^2 \leq 2^n \leq n^3$
- C $n \leq 2^n \leq n^2 \leq lg\, n \leq n^3$
- D $lg\, n \leq n \leq n^2 \leq n^3 \leq 2^n$
- E $lg\, n \leq n \leq n^2 \leq 2^n \leq n^3$

- $c \cdot f(n) = O(f(n))$ (robustness: ignore constants)
- $n^k = O(n^l)$ if $l \geq k$
- $n^k \neq O(n^l)$ if $l < k$
- $\log(n) = O(n)$ and $n \neq O(\log(n))$
- $n^k = O(2^n)$ and $2^n \neq O(n^k)$



- $c \cdot f(n) = O(f(n))$ (robustness: ignore constants)
- $n^k = O(n^l)$ if $l \geq k$
- $n^k \neq O(n^l)$ if $l < k$
- $\log(n) = O(n)$ and $n \neq O(\log(n))$
- $n^k = O(2^n)$ and $2^n \neq O(n^k)$



Actual computation time assuming 1G steps/second

Complexity	Input size 10	50	100	200	500
$\log(n)$	$3 \cdot 10^{-6} ms$	$5 \cdot 10^{-6} ms$	$6 \cdot 10^{-6} ms$	$7 \cdot 10^{-6} ms$	$9 \cdot 10^{-6} ms$
n	$10^{-5} ms$	$5 \cdot 10^{-5} ms$	$10^{-4} ms$	$2 \cdot 10^{-4} ms$	$5 \cdot 10^{-4} ms$
n^2	$10^{-4} ms$	$2.5 \cdot 10^{-3} ms$	$10^{-2} ms$	$4 \cdot 10^{-2} ms$	$0.25 ms$
n^3	$10^{-3} ms$	$0.12 ms$	$1 ms$	$8 ms$	$0.12 s$
2^n	$10^{-3} ms$	$312h : 44m$	10^{13} y	$10^{43} y$	$10^{133} y$

Considering the Ratio

If $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, then $f(n) = O(g(n))$ (stronger: $f(n) = o(g(n))$), and $g(n) \neq O(f(n))$.

Polynomials vs. Exponentials

For all $a > 1$ and $b > 0$:

$$n^b = O(a^n) \text{ and } a^n \neq O(n^b)$$

Polylogarithms vs. Polynomials

For all $a, b > 0$:

$$\log^b n = O(n^a) \text{ and } n^a \neq O(\log^b n)$$

If

$$g_1(n) = O(f_1(n)), \quad \ldots \quad , g_k(n) = O(f_k(n)),$$

then

$$g_1(n) + \ldots + g_k(n) = O(\max_{i=1}^{k} f_i(n)).$$

where *max* is taken with respect to the (partial) order $f_i \leq f_j \Leftrightarrow f_i(n) \in O(f_j(n))$.

For products:

$$g_1(n) \cdots g_k(n) = O(f_1(n) \cdots f_k(n))$$

If

$$g_1(n) = O(f_1(n)), \ \ldots \ , g_k(n) = O(f_k(n)),$$

then

$$g_1(n) + \ldots + g_k(n) = O(\max_{i=1}^{k} f_i(n)).$$

where *max* is taken with respect to the (partial) order $f_i \leq f_j \Leftrightarrow f_i(n) \in O(f_j(n))$.

For products:

$$g_1(n) \cdots g_k(n) = O(f_1(n) \cdots f_k(n))$$

Careful!

- k must be a fixed constant that does not grow with n. Example: $g_i(n)$ the number of times the ith line of the algorithm is executed.
- The rule for sums only applies if the *max* actually exist (see next slide!).

If

$$g_1(n) = O(f_1(n)), \quad \dots \quad , g_k(n) = O(f_k(n)),$$

then

$$g_1(n) + \ldots + g_k(n) = O(\max_{i=1}^{k} f_i(n)).$$

where *max* is taken with respect to the (partial) order $f_i \leq f_j \Leftrightarrow f_i(n) \in O(f_j(n))$.

For products:

$$g_1(n) \cdots g_k(n) = O(f_1(n) \cdots f_k(n))$$

Careful!

- k must be a fixed constant that does not grow with n. Example: $g_i(n)$ the number of times the ith line of the algorithm is executed.
- The rule for sums only applies if the *max* actually exist (see next slide!).

Examples:

For polynomials, only the highest exponent counts:

$$23 \cdot n^3 + 54 \cdot n^2 + 32 \cdot n + 3421 = O(n^3).$$

A more messy function:

$$15 \cdot \log(n) - 23 \cdot n^2 + 12 \cdot n^3 + 43 \cdot n^2 \cdot \log(n) = O(n^3).$$

Transitivity

If $g(n) = O(f(n))$ and $f(n) = O(h(n))$, then $g(n) = O(h(n))$.

Careful!

It is **not** the case that one of $g(n) = O(f(n))$ and $f(n) = O(g(n))$ must be true.

Example: $g(n) = n$; $f(n) = n^{n \bmod 3}$; then neither $g(n) = O(f(n))$ nor $f(n) = O(g(n))$.

Also: $max\{f(n), g(n)\}$ is not defined.

Max-Subarray Example

The Problem

Given an array with integers, find the subarray with maximal sum of entries.

Example: an array

$$[13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]$$

The Problem

Given an array with integers, find the subarray with maximal sum of entries.

Example: an array

$$[13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]$$

Maximal subarray

The Problem

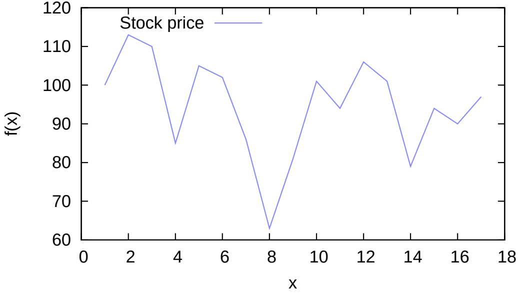
Given an array with integers, find the subarray with maximal sum of entries.

Example: an array

$$[13, -3, -25, 20, -3, -16, -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]$$

Maximal subarray

Application:



Find optimal buy and sell date for a stock.

input : An array A with integer values
output: The maximal subarray of A given by its start and end indices, and the sum s of the subarray

STUPIDMAXSUBARR(A)

```
1  MaxSum=−∞
2  MaxLeft=MaxRight=0
3  for j = 1..A.length do
4      for k = j..A.length do
5          sum=0
6          for i = j..k do
7              sum=sum+A[i]
8          if sum>MaxSum then
9              MaxSum=sum
10             MaxLeft=j
11             MaxRight=k
12 return [MaxLeft, MaxRight, MaxSum]
```

Line	# executed (worst case)
1	1
2	1
3	n
4	$n + (n-1) + \ldots + 1$
5	$n + (n-1) + \ldots + 1$
6	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$
7	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$
8	$n + (n-1) + \ldots + 1$
9	$n + (n-1) + \ldots + 1$
10	$n + (n-1) + \ldots + 1$
11	$n + (n-1) + \ldots + 1$
12	1

A formula to know (arithmetic series):

$$1 + 2 + \ldots + (n-1) + n = (n+1)\frac{n}{2}$$

A little more generally:

$$a + (a+1) + \ldots + (a+n) = (n+2a)\frac{n+1}{2}$$

Line	# executed	summed	O-ed
1	1		
2	1		
3	n		
4	$n + (n-1) + \ldots + 1$		
5	$n + (n-1) + \ldots + 1$		
6	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$		
7	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$		
8	$n + (n-1) + \ldots + 1$		
9	$n + (n-1) + \ldots + 1$		
10	$n + (n-1) + \ldots + 1$		
11	$n + (n-1) + \ldots + 1$		
12	1		

A formula to know (arithmetic series):

$$1 + 2 + \ldots + (n-1) + n = (n+1)\frac{n}{2}$$

A little more generally:

$$a + (a+1) + \ldots + (a+n) = (n+2a)\frac{n+1}{2}$$

Line	# executed	summed	O-ed
1	1	1	
2	1	1	
3	n	n	
4	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
5	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
6	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$		
7	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$		
8	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
9	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
10	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
11	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
12	1	1	

A formula to know (arithmetic series):

$$1 + 2 + \ldots + (n-1) + n = (n+1)\frac{n}{2}$$

A little more generally:

$$a + (a+1) + \ldots + (a+n) = (n+2a)\frac{n+1}{2}$$

Line	# executed	summed	O-ed
1	1	1	
2	1	1	
3	n	n	
4	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
5	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
6	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$	?	
7	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$	?	
8	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
9	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
10	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
11	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	
12	1	1	

A formula to know (arithmetic series):

$$1 + 2 + \ldots + (n-1) + n = (n+1)\frac{n}{2}$$

A little more generally:

$$a + (a+1) + \ldots + (a+n) = (n+2a)\frac{n+1}{2}$$

Line	# executed	summed	O-ed
1	1	1	$O(1)$
2	1	1	$O(1)$
3	n	n	$O(n)$
4	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
5	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
6	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$	?	
7	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$	?	
8	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
9	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
10	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
11	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
12	1	1	$O(1)$

A formula to know (arithmetic series):

$$1 + 2 + \ldots + (n-1) + n = (n+1)\frac{n}{2}$$

A little more generally:

$$a + (a+1) + \ldots + (a+n) = (n+2a)\frac{n+1}{2}$$

Line	# executed	summed	O-ed
1	1	1	$O(1)$
2	1	1	$O(1)$
3	n	n	$O(n)$
4	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
5	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
6	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$	?	$O(n^3)$
7	$n \cdot 1 + (n-1) \cdot 2 + (n-2) \cdot 3 + \ldots + 1 \cdot n$	?	$O(n^3)$
8	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
9	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
10	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
11	$n + (n-1) + \ldots + 1$	$(n+1)\frac{n}{2}$	$O(n^2)$
12	1	1	$O(1)$

Each level of for-looping adds $+1$ to the polynomial (if number of loop iterations is input-dependent)