Asymptotic analysis of algorithm runtimes

Analysis of runtime

Recall: We want to evaluate (analyze) algorithms in advance before we spend a lot of time implementing them.

The two main **questions**:

- Does the algorithm solve the problem (is it correct)?
- Is the algorithm efficient (what is the runtime)?

Analysis of runtime

Recall: We want to evaluate (analyze) algorithms in advance before we spend a lot of time implementing them.

The two main **questions**:

- Does the algorithm solve the problem (is it correct)?
- Is the algorithm efficient (what is the runtime)?

In these slides, we focus on the second question.

Analysis of runtime

Recall: In our analysis, the running time of an algorithm is equal to the number of basic operations it performs in the RAM model (for worst case input). This number of operations is a *growing function f(n)* of the input size n.

Analysis of runtime

Recall: In our analysis, the running time of an algorithm is equal to the number of basic operations it performs in the RAM model (for worst case input). This number of operations is a *growing function f(n)* of the input size n.

1. We will start by examining how well theoretical analyses in the RAM model seem to fit with the observed runtime of algorithms on real computers.

Analysis of runtime

Recall: In our analysis, the running time of an algorithm is equal to the number of basic operations it performs in the RAM model (for worst case input). This number of operations is a *growing function f(n)* of the input size n.

1. We will start by examining how well theoretical analyses in the RAM model seem to fit with the observed runtime of algorithms on real computers.

2. Next, we will introduce a tool called asymptotic analysis, to compare f(n) for different algorithms in a fairly (u) precise way.

Analysis of runtime

Recall: In our analysis, the running time of an algorithm is equal to the number of basic operations it performs in the RAM model (for worst case input). This number of operations is a *growing function f(n)* of the input size n.

1. We will start by examining how well theoretical analyses in the RAM model seem to fit with the observed runtime of algorithms on real computers.

2. Next, we will introduce a tool called asymptotic analysis, to compare f(n) for different algorithms in a fairly (u) precise way.

Goal: To roughly classify algorithms based on the growth rate of their runtimes, allowing us to avoid implementing those that have no chance of being the fastest.

Analysis vs. Reality

```python
import sys
import time
def main():
    start_time = time.time()
    n = int(sys.argv[1])
    total = 0
    for i in range(1, n+1):
        total += 1
    print("The total:",total)
    print("The execution time:",(time.time() - start_time))

if __name__ == "__main__":
    main()
```

Analyse

$$\boxed{\text{Time}(n) = c_1 \cdot n + c_0}$$

Analysis vs. Reality

Function adds 1 up to n

```
import sys
import time
def main():
    start_time = time.time()
    n = int(sys.argv[1])
    total = 0
    for i in range(1, n+1):
        total += 1
    print("The total:",total)
    print("The execution time:",(time.time() - start_time))

if __name__ == "__main__":
    main()
```

The Time the algorithm takes

Analyse

$$\text{Time}(n) = c_1 \cdot n + c_0$$

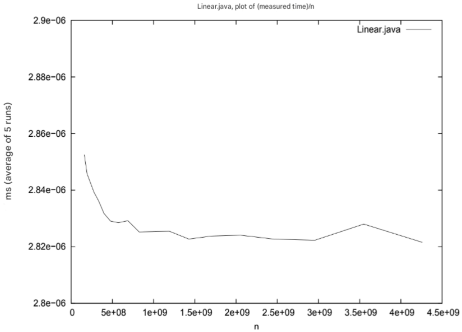
when dividing by n

$$\frac{\text{Time}(n)}{n} = \frac{c_1 \cdot n + c_0}{n} = c_1 + \frac{c_0}{n}$$

The bigger n is, the smaller C0/n becomes and there for the algorithm primarly depend on C1

This shows that for very large 'n', Time(n) / n is approximately equal to C1, a constant.
This is another way to see that the runtime is l inear with respect to 'n'.

Analysis vs. Reality



Reality

x-axis:

input size n

y-axis:

(measured time)/n

As 'n' becomes very large (towards the right side of the graph), the c0 / n term approaches zero. Therefore, Time(n) / n becomes approximately equal to c1

The start is a bit higher, becuase the c0 / n term is more significant when 'n' is small.

The flat part of the curve can be due to fluctionation:
* Measure noise
* system variation (CPU,operating system etc)

Analysis vs. Reality

```
for i in range(1, n+1):
    for j in range (1,n+1):
        total += 1
```

Analysis vs. Reality

```
for i in range(1, n+1):
    for j in range (1,n+1):
        total += 1
```

Time(n)

$$= (c_2 \cdot n + c_1) \cdot n + c_0$$

$$= c_2 \cdot n^2 + c_1 \cdot n + c_0$$

Analysis vs. Reality

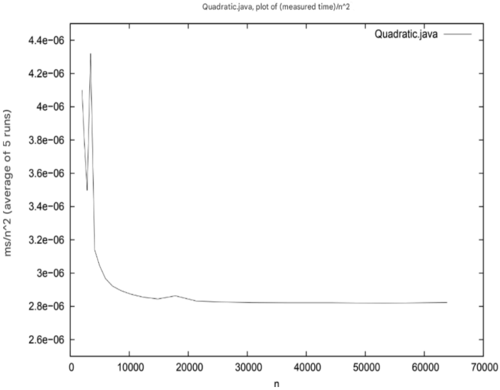
```python
for i in range(1, n+1):
    for j in range (1,n+1):
        total += 1
```

Time(n)

$$= (c_2 \cdot n + c_1) \cdot n + c_0$$

$$= c_2 \cdot n^2 + c_1 \cdot n + c_0$$

equal to n^2



x-axis:
input size n

y-axis:
(measured time)/n^2

Again inizial high because n is relative bigger

Analysis vs. Reality

```
for i in range(1, n+1):
    for j in range (1,n+1):
        for k in range (1,n+1):
            total += 1
```

Analysis vs. Reality

```python
for i in range(1, n+1):
    for j in range (1,n+1):
        for k in range (1,n+1):
            total += 1
```

Time(n)

$$= ((c_3 \cdot n + c_2) \cdot n + c_1) \cdot n + c_0$$

$$= c_3 \cdot n^3 + c_2 \cdot n^2 + c_1 \cdot n + c_0$$

Analysis vs. Reality

inner loop:
C3 * n
middel loop:
n+C1
outer loop:
n+C0

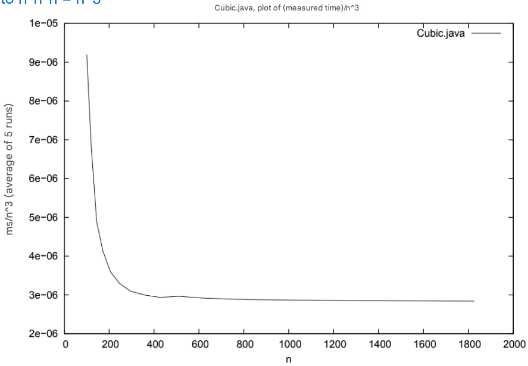
```python
for i in range(1, n+1):
    for j in range (1,n+1):
        for k in range (1,n+1):
            total += 1
```

Time(n)

$$= ((c_3 \cdot n + c_2) \cdot n + c_1) \cdot n + c_0$$

$$= c_3 \cdot n^3 + c_2 \cdot n^2 + c_1 \cdot n + c_0$$

similar to n*n*n = n^3



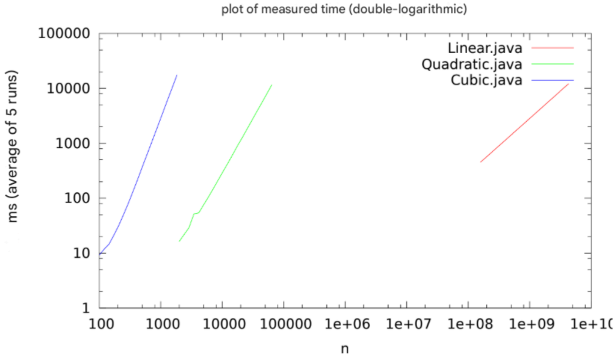
x-axis:
input size n

y-axis:
(measured time)/n^3

Analysis vs. Reality

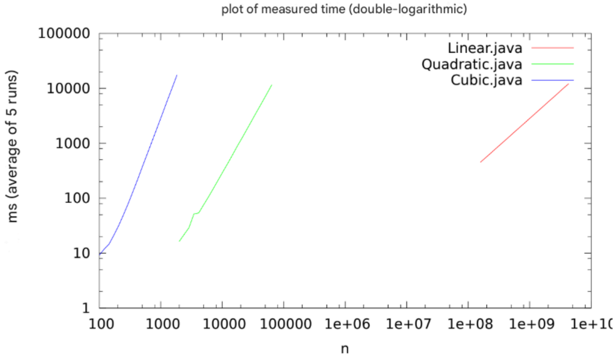
Conclusion: It seems that analyses in the RAM model predict the correct execution time quite well, at least for the tested examples.

Linear vs. Quadratic vs. Cubic



You can see that the functions n ,n^2 and n^3 represent very different efficiencies.

Linear vs. Quadratic vs. Cubic



You can see that the functions n ,n^2 and n^3 represent very different efficiencies. In the analysis, a number of constants actually appear (which we typically have difficulty knowing precisely), e.g., $c_1 .n + c_0$. Do these matter?

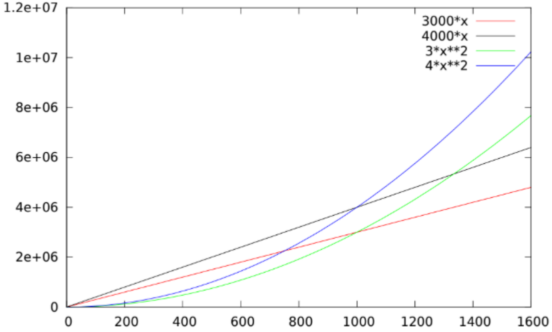
Yes they matter for small input values n

Multiplicative Constants

Multiplicative constants don’t matter if the growth rates are different.

$f(n) = 3000n$ $\quad$ $h(n) = 3n^2$

$g(n) = 4000n$ $\quad$ $k(n) = 4n^2$

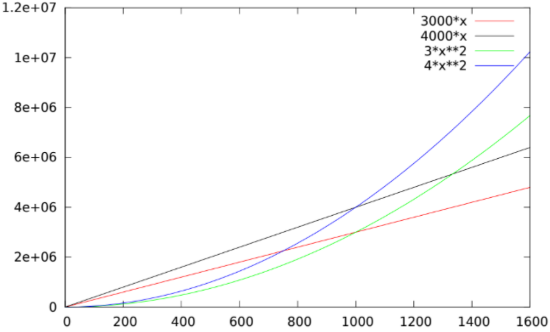


Multiplicative Constants

Multiplicative constants don't matter if the growth rates are different.

$f(n) = 3000n$
$g(n) = 4000n$

$h(n) = 3n^2$
$k(n) = 4n^2$



Does 3000n win over $4n^2$? Yes: $3000n < 4n^2 \Leftrightarrow 3000 < 4n \Leftrightarrow 750 < n$

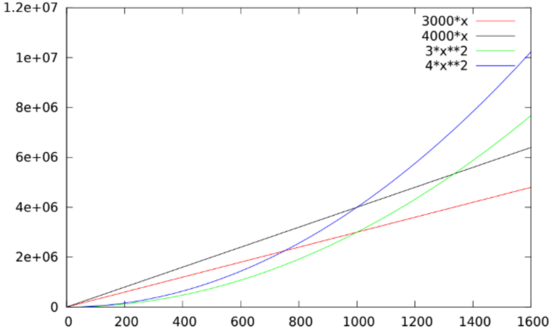
when n is bigger then 750, the quadratic function is slower

Multiplicative Constants

Multiplicative constants don't matter if the growth rates are different.

$f(n) = 3000n$ $\quad$ $h(n) = 3n^2$

$g(n) = 4000n$ $\quad$ $k(n) = 4n^2$



Does 3000n win over $4n^2$? Yes: $3000n < 4n^2 \Leftrightarrow 3000 < 4n \Leftrightarrow 750 < n$

In fact, $c_1 .n$ **always wins** over $c_2 .n^2$: $c_1 . n < c_2 . n^2 \Leftrightarrow c_1 / c_2 < n$

The Growth Rate

We therefore want to compare the essential growth rates of functions in a way that ignores multiplicative constants. Such a comparison can be used to make a rough sorting of algorithms before we do implementation work.

If two algorithms, A and B, have growth rates where algorithm B will always (for large n) be outperformed by algorithm A, regardless of the multiplicative constants in the growth rate expressions, then there will usually be no point in implementing algorithm B.

The Growth Rate

We therefore want to compare the essential growth rates of functions in a way that ignores multiplicative constants. Such a comparison can be used to make a rough sorting of algorithms before we do implementation work.

If two algorithms, A and B, have growth rates where algorithm B will always (for large n) be outperformed by algorithm A, regardless of the multiplicative constants in the growth rate expressions, then there will usually be no point in implementing algorithm B.

Note: In the above situation, we do not need to know the constants to make this assessment. We can therefore do runtime analysis without worrying about knowing the exact size of the input constants.

Asymptotic Notation

So we want a tool to compare the essential growth rate of functions in a way that disregards multiplicative constants.

The principle of our tool will be the following: for a function $f(n)$, we will consider all scalings of it.

$$\{c.\ f(n) \mid \text{for all } c > 0\}$$

seems just as good

Asymptotic Notation

So we want a tool to compare the essential growth rate of functions in a way that disregards multiplicative constants.

The principle of our tool will be the following: for a function $f(n)$, we will consider all scalings of it.

$$\{c.\, f(n) \mid \text{for all } c > 0\}$$

we multiply any function with a constant

seems just as good

No matter the positive constant, the function remains the same, in terms of growth rate. So the multiplicative doesn't matter

In the following we will call this set of functions the class of $f(n)$.

Asymptotic Notation

Based on this principle, we define five relations for the growth rate of functions, corresponding to the five classical order relations.

$$\leq \qquad \geq \qquad = \qquad < \qquad >$$

They will, for historical reasons, be called:

compare growth rate as input(n) gets bigger

$$O \qquad \Omega \qquad \Theta \qquad o \qquad \omega$$

Which are pronounced as follows:

"O", "Omega", "Theta", "little o", "little omega"

The five definitions are described over the next five pages.

O

f(n) is Big O of g(n)

Definition: $f(n) = O(g(n))$

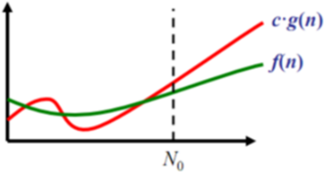
if f(n) and g(n) are functions $N \rightarrow R$ and $c > 0$

scaling constant c is bigger then 0

Maps from Natural numbers to real numbers

and N_0 exists then for all $n \geq N_0$:

$$f(n) \leq c \cdot g(n)$$



N_0 = threshold where c*g(n) >= f(n)

It means: f ≤ g in growth rate

Principle: $f(n)$ grows at most as fast as functions from the class of $g(n)$.

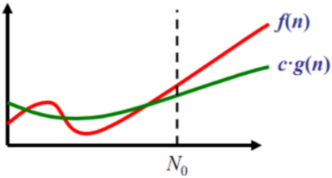
Omega

Definition: $f(n) = \Omega(g(n))$

if f(n) and g(n) are functions N → R and there

exists c > 0 and N_0 then for all n ≥ N_0 :

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



It means: f ≥ g in growth rate

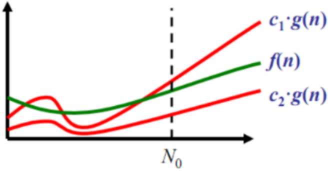
Principle: $f(n)$ grows at least as fast as functions from the class of g(n) .

Theta

Definition: $f(n) = \theta(g(n))$

if f(n)=O(g(n)) and f(n)= Ω(g(n)

if and only if f(n) < O, and f(n) > omega



It means: f = g in growth rate

Principle: $f(n)$ grows as fast as functions from the class of g(n) .

Little o

Definition: $f(n) = o(g(n))$

if f(n) and g(n) are functions $N \longrightarrow R$ and

for all $c > 0$, N_0 exists, so for all $n \geq N_0$:

No matter the scaling constant, f(n) is less then or equal to g(n)

$$f(n) \leq c \cdot g(n)$$

It means: f < g in growth rate

Principle: $f(n)$ grows more slowly than all functions from the class of $g(n)$.

Little omega

Definition: $f(n) = \omega(g(n))$

if f(n) and g(n) are functions N → R and

for all c > 0, N_0 then exists for all n ≥ N_0 :

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

It means: f > g in growth rate

Principle: $f(n)$ grows faster than all functions from the class of g(n) .

Asymptotic Notation

It can easily be shown that these definitions behave as expected for order relations. For example:

$$f(n) = o(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x < y \Rightarrow x \leq y)$$

Asymptotic Notation

It can easily be shown that these definitions behave as expected for order relations. For example:

$$f(n) = o(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x < y \Rightarrow x \leq y)$$

$$f(n) = \Theta(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x = y \Rightarrow x \leq y\ )$$

Asymptotic Notation

It can easily be shown that these definitions behave as expected for order relations. For example:

$$f(n) = o(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x < y \Rightarrow x \leq y)$$

$$f(n) = \Theta(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x = y \Rightarrow x \leq y\ )$$

$$f(n) = O(g(n)) \;\Leftrightarrow\; g(n) = \Omega(f(n)) \qquad (\text{jvf. } x \leq y \Leftrightarrow y \geq x\ )$$

Asymptotic Notation

It can easily be shown that these definitions behave as expected for order relations. For example:

$$f(n) = o(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x < y \Rightarrow x \leq y)$$

$$f(n) = \Theta(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x = y \Rightarrow x \leq y\ )$$

$$f(n) = O(g(n)) \;\Leftrightarrow\; g(n) = \Omega(f(n)) \qquad (\text{jvf. } x \leq y \Leftrightarrow y \geq x\ )$$

$$f(n) = o(g(n)) \;\Leftrightarrow\; g(n) = \omega(f(n)) \qquad (\text{jvf. } x < y \Leftrightarrow y > x\ )$$

Asymptotic Notation

It can easily be shown that these definitions behave as expected for order relations. For example:

$$f(n) = o(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x < y \Rightarrow x \leq y)$$

$$f(n) = \Theta(g(n)) \;\Rightarrow\; f(n) = O(g(n)) \qquad (\text{jvf. } x = y \Rightarrow x \leq y\ )$$

$$f(n) = O(g(n)) \;\Leftrightarrow\; g(n) = \Omega(f(n)) \qquad (\text{jvf. } x \leq y \Leftrightarrow y \geq x\ )$$

$$f(n) = o(g(n)) \;\Leftrightarrow\; g(n) = \omega(f(n)) \qquad (\text{jvf. } x < y \Leftrightarrow y > x\ )$$

$$f(n) = O(g(n)) \text{ og } f(n) = \Omega(g(n)) \;\Rightarrow\; f(n) = \Theta(g(n))$$
$$(\text{jvf. } x \leq y \text{ og } x \geq y \Rightarrow x = y\ )$$

Asymptotic analysis in practice

The asymptotic relationships between most functions f and g can be clarified by the following two theorems (which can be proven from the definitions):

If: $$\frac{f(n)}{g(n)} \to k > 0 \text{ for } n \to \infty \quad \Rightarrow \quad f(n) = \Theta(g(n)) \qquad (1)$$

$$\frac{f(n)}{g(n)} \to 0 \text{ for } n \to \infty \quad \Rightarrow \quad f(n) = o(g(n)) \qquad (2)$$

Asymptotic analysis in practice

The asymptotic relationships between most functions f and g can be clarified by the following two theorems (which can be proven from the definitions):

If:

$$\frac{f(n)}{g(n)} \to k > 0 \text{ for } n \to \infty \quad \Rightarrow \quad f(n) = \Theta(g(n)) \qquad (1)$$

$$\frac{f(n)}{g(n)} \to 0 \text{ for } n \to \infty \quad \Rightarrow \quad f(n) = o(g(n)) \qquad (2)$$

Skriv tekst her

Examples:

$$\frac{20n^2 + 17n + 312}{n^2} = \frac{20 + 17/n + 312/n^2}{1} \to \frac{20 + 0 + 0}{1} = 20 \text{ for } n \to \infty$$

Asymptotic analysis in practice

The asymptotic relationships between most functions f and g can be clarified by the following two theorems (which can be proven from the definitions):

If the ratio of f(n) to g(n) approaches a positive constant (not zero, not infinity) as n gets very large,
it means that f(n) and g(n) are growing at the same rate. And "growing at the same rate"
is precisely what Big Theta means!

Find out if in theta

If: $$\frac{f(n)}{g(n)} \to k > 0 \text{ for } n \to \infty \quad \Rightarrow \quad f(n) = \Theta(g(n)) \qquad (1)$$

And the theorem says, if this limit
exists and is equal to some finite constant k
that is strictly greater than zero

or little o relationship

$$\frac{f(n)}{g(n)} \to 0 \text{ for } n \to \infty \quad \Rightarrow \quad f(n) = o(g(n)) \qquad (2)$$

if the limit is 0 as n approch infinity
then it is little o relation ship.

Examples:

In plain English: If the ratio of f(n) to g(n) approaches zero as n gets very large, i
t means that f(n) is growing significantly slower than g(n).
And "growing strictly slower" is what little o notation is all about!

$$\frac{20n^2 + 17n + 312}{n^2} = \frac{20 + 17/n + 312/n^2}{1} \to \frac{20 + 0 + 0}{1} = 20 \text{ for } n \to \infty$$

because when n gets big
17/n and 312/n, gets small

since 20 is bigger then 0, it is thetha

$$\frac{20n^2 + 17n + 312}{n^3} = \frac{20/n + 17/n^2 + 312/n^3}{1} \to \frac{0 + 0 + 0}{1} = 0 \text{ for } n \to \infty$$

since it is 0, it is little o

Asymptotic analysis in practice

In addition, it is useful to know the following fact from mathematics:

For all $a > 0$ and $b > 1$ it means: $\frac{n^a}{b^n} \longrightarrow 0$ for $n \rightarrow \infty$ (3)

Asymptotic analysis in practice

In addition, it is useful to know the following fact from mathematics:

For all a > 0 and b >1 it means: $\frac{n^a}{b^n} \longrightarrow 0$ for $n \rightarrow \infty$ (3)

$\boxed{\text{That is, any polynomial is o() of any exponential function}}$

Asymptotic analysis in practice

In addition, it is useful to know the following fact from mathematics:

For all a > 0 and b >1 it means: $\frac{n^a}{b^n} \longrightarrow 0$ for $n \longrightarrow \infty$ (3)

$\boxed{\text{That is, any polynomial is o() of any exponential function}}$

For example, this gives:

$$\frac{n^{100}}{2^n} \to 0 \text{ for } n \to \infty$$

becuase the limit is zero, we know from last slide
it is little o

$$n^{100} = o(2^n)$$

Asymptotic analysis in practice

As well as the following fact:

For all a, d >0 and c >1 it means: $\frac{(log_c n)^a}{n^d} \to 0$ for n $\longrightarrow \infty$ (4)

Asymptotic analysis in practice

As well as the following fact:

For all a, d >0 and c >1 it means: $\frac{(\log_c n)^a}{n^d} \rightarrow 0$ for n $\longrightarrow \infty$ (4)

That is, any logarithm (even raised to any power) o() of any polynomial.

Asymptotic analysis in practice

As well as the following fact:

For all a, d >0 and c >1 it means: $\dfrac{(\log_c n)^a}{n^d} \rightarrow 0$ for n ⟶ ∞ (4)

That is, any logarithm (even raised to any power) o() of any polynomial.

In other words, polynomial functions grow asymptotically
faster than any logarithmic function, no matter how much you raise the logarithm to a power.

For example, this gives:

$$\frac{(\log\ n)^3}{n^{0.5}} \rightarrow 0 \quad \text{for } n \longrightarrow \infty \qquad \Rightarrow \qquad (\log\ n)^3 = o(n^{0.5})$$

Examples of growth rate functions

With rules (1)–(4), it can be shown that the following functions are arranged in increasing growth rate (more precisely, that one is o() of the next):

$$1, \quad \log n, \quad \sqrt{n}, \quad n, \quad n \log n,$$

$$n\sqrt{n}, \quad n^2, \quad n^3, \quad n^{10}, \quad 2^n$$

Examples of growth rate functions

With rules (1)–(4), it can be shown that the following functions are arranged in increasing growth rate (more precisely, that one is o() of the next):

$$1, \quad \log n, \quad \sqrt{n}, \quad n, \quad n \log n,$$

$$n\sqrt{n}, \quad n^2, \quad n^3, \quad n^{10}, \quad 2^n$$

These have quite different effectiveness in practice:

	n	$n \log n$	n^2	n^3	n^{10}	2^n
1 Minute	$6,0 \cdot 10^{10}$	$1,9 \cdot 10^{9}$	245.000	3.910	12	36
1 month	$2,6 \cdot 10^{15}$	$5,7 \cdot 10^{13}$	50.900.000	137.000	35	51

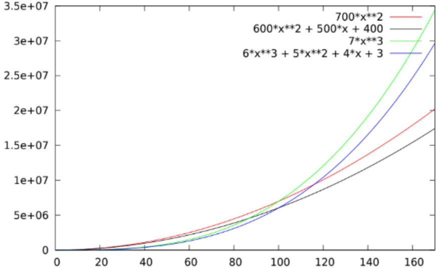
The table shows which input sizes n can be done if the algorithm has to perform the specified number of CPU operations and it has to finish after one minute and one month, respectively. It is assumed that a CPU can do 10^9 operations per second.

Dominant term

For functions with multiple terms (parts with a plus sign between them), the term(s) with the highest growth rate will determine the overall growth rate. Example:

$$f(n) = 700n^2 \qquad g(n) = 7n^3$$

$$h(n) = 600n^2 + 500n + 400 \qquad k(n) = 6n^3 + 5n^2 + 4n + 3$$

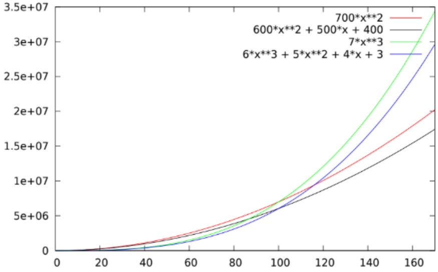


Dominant term

For functions with multiple terms (parts with a plus sign between them), the term(s) with the highest growth rate will determine the overall growth rate. Example:

$$f(n) = 700n^2 \qquad g(n) = 7n^3$$

$$h(n) = 600n^2 + 500n + 400 \qquad k(n) = 6n^3 + 5n^2 + 4n + 3$$



The figure fits with calculations:

Dominant term

$$\frac{6n^3 + 5n^2 + 4n + 3}{7n^3} = \frac{6 + 5/n + 4/n^2 + 3/n^3}{7} \to \frac{6 + 0 + 0 + 0}{7} = 6/7 \text{ for } n \to \infty$$

$$6n^3 + 5n^2 + 4n + 3 = \Theta(7n^3)$$

Dominant term

$$\frac{6n^3 + 5n^2 + 4n + 3}{7n^3} = \frac{6 + 5/n + 4/n^2 + 3/n^3}{7} \to \frac{6+0+0+0}{7} = 6/7 \text{ for } n \to \infty$$

$$6n^3 + 5n^2 + 4n + 3 = \Theta(7n^3)$$

$$\frac{600n^2 + 500n + 400}{700n^2} = \frac{600 + 500/n + 400/n^2}{700} \to \frac{600+0+0}{700} = 6/7 \text{ for } n \to \infty$$

$$600n^2 + 500n + 400 = \Theta(700n^2)$$

Dominant term

$$\frac{6n^3+5n^2+4n+3}{7n^3} = \frac{6+5/n+4/n^2+3/n^3}{7} \to \frac{6+0+0+0}{7} = 6/7 \text{ for } n \to \infty$$

grows equaly fast

$$6n^3+5n^2+4n+3 = \Theta(7n^3)$$

$$\frac{600n^2+500n+400}{700n^2} = \frac{600+500/n+400/n^2}{700} \to \frac{600+0+0}{700} = 6/7 \text{ for } n \to \infty$$

$$600n^2+500n+400 = \Theta(700n^2)$$

grows equally fast

$$\frac{600n^2+500n+400}{6n^3+5n^2+4n+3} = \frac{600/n+500/n^2+400/n^3}{6+5/n+4/n^2+3/n^3} \to \frac{0+0+0}{6+0+0+0} = 0 \text{ for } n \to \infty$$

$$600n^2+500n+400 = o(6n^3+5n^2+4n+3)$$

n^3 is little o(strictly upper bund) n^2 grows slower

Example 1 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME1(n)
  i = 1
  while i ≤ n
    i = i + 2
```

Example 1 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME1(n)
  i = 1
  while i ≤ n
    i = i + 2
```

The loop has $\lceil n/2 \rceil = \Theta(n)$ iterations.

Example 1 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME1(n)
  i = 1
  while i ≤ n
    i = i + 2
```

The loop has $\lceil n/2 \rceil = \Theta(n)$ iterations.

Each iteration takes between c_1 and c_2 time for (unknown) constants c_1 and c_2.
So, each iteration takes $\Theta(1)$ time.

Example 1 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

$\textsc{Algoritme1}(n)$
  $i = 1$
  while $i \leq n$
    $i = i + 2$

The loop has $\lceil n/2 \rceil = \Theta(n)$ iterations.

Each iteration takes between c_1 and c_2 time for (unknown) constants c_1 and c_2.
So, each iteration takes $\Theta(1)$ time.

Therefore, the runtime is $\Theta(n.1) = \Theta(n)$.

Example 1 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

$$
\begin{array}{l}
\textsc{Algoritme1}(n) \\
\quad i = 1 \\
\quad \textbf{while } i \leq n \\
\quad\quad i = i + 2
\end{array}
$$

In essence, this algorithm starts with i at 1 and keeps adding 2 to i until i becomes greater than n.

The loop has $\lceil n/2 \rceil = \Theta(n)$ iterations.

This is essiensially 1/2 * n, and since we dont care abot the constant, it is n. which is linear

Because the runtime of the algorithm grows linearly with the i nput size n, just like the function g(n) = n itself, we say it is Big Theta of n, or Θ(n)

Each iteration takes between c_1 and c_2 time for (unknown) constants c_1 and c_2.

So, each iteration takes $\Theta(1)$ time.

the time taking is bound by constant

c1 and c2 are just representing some unknown but fixed constants. The exact time for i = i + 2 might vary slightly depending on the specific computer, compiler, etc., but importantly, it doesn't depend on the value of n. It's always going to take roughly the same amount of time, regardless of how big n is.

Therefore, the runtime is $\Theta(n.1) = \Theta(n)$.

To get the total runtime of the loop, we multiply the number of it erations (Θ(n)) by the time per iteration (Θ(1)), which gives us a total loop runtime of Θ(n).

We have omitted to discuss the time for the first line and for initialization of the loop. A more precise analysis would give an expression of the type $c_1 .n+c_0$, but we omit to talk about terms that are clearly dominated by other terms .

Example 2 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME2(n)
    s = 0
    for i = 1 to n
        for j = i downto 1
            s = s + 1
```

Example 2 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

$$
\begin{array}{l}
\textsc{Algoritme2}(n) \\
\quad s = 0 \\
\quad \textbf{for } i = 1 \textbf{ to } n \\
\quad\quad \textbf{for } j = i \textbf{ downto } 1 \\
\quad\quad\quad s = s + 1
\end{array}
$$

There are n iterations of the outer loop. For each of these, there is at most n iterations of the inner loop. Each iteration of the inner loop takes $\Theta(1)$ time. So the runtime is $O(n.n.1) = O(n^2)$.

Example 2 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME2(n)
    s = 0
    for i = 1 to n
        for j = i downto 1
            s = s + 1
```

There are n iterations of the outer loop. For each of these, there is at most n iterations of the inner loop. Each iteration of the inner loop takes $\Theta(1)$ time. So the runtime is $O(n.n.1) = O(n^2)$.

For $i \geq n/2$ (i.e. for $n/2$ iterations of the outer loop) there are at least $n/2$ iterations of the inner loop. So the running time is $\Omega(n/2.\ n/2\ .1) = \Omega(n^2)$.

Example 2 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME2(n)
  s = 0
  for i = 1 to n
      for j = i downto 1
          s = s + 1
```

There are n iterations of the outer loop. For each of these, there is at most n iterations of the inner loop. Each iteration of the inner loop takes $\Theta(1)$ time. So the runtime is $O(n.n.1) = O(n^2)$.

For $i \geq n/2$ (i.e. for $n/2$ iterations of the outer loop) there are at least $n/2$ iterations of the inner loop. So the running time is $\Omega(n/2.\ n/2\ .1) = \Omega(n^2)$.

Therefore, the total runtime is $\Theta(n^2)$.

Example 2 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME2(n)
  s = 0
  for i = 1 to n
      for j = i downto 1
          s = s + 1
```

There are n iterations of the outer loop. For each of these, there is at most n iterations of the inner loop. Each iteration of the inner loop takes $\Theta(1)$ time. So the runtime is $O(n.n.1) = O(n^2)$.

For $i \geq n/2$ (i.e. for $n/2$ iterations of the outer loop) there are at least $n/2$ iterations of the inner loop. So the running time is $\Omega(n/2.\ n/2\ .1) = \Omega(n^2)$.

Therefore, the total runtime is $\Theta(n^2)$.

[Alternative analysis: the inner loop runs $(1 + 2 + 3 +\cdots+n) = (n+1)n/2 = \Theta(n^2)$ times.]

Example 3 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME3(n)
    s = 0
    for i = 1 to n
        for j = i to n
            for k = i to j
                s = s + 1
```

Example 3 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME3(n)
  s = 0
  for i = 1 to n
      for j = i to n
          for k = i to j
              s = s + 1
```

There are n iterations of the outer loop. For each of these there are at most n iterations of the middle loop. For each of these there are at most n iterations of the inner loop. Each iteration of the inner loop takes $\Theta(1)$ time. So the running time is $O(n.n.n.1) = O(n^3)$.

Example 3 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

$\text{ALGORITME3}(n)$
  $s = 0$
  for $i = 1$ **to** n
    for $j = i$ **to** n
      for $k = i$ **to** j
        $s = s + 1$

There are n iterations of the outer loop. For each of these there are at most n iterations of the middle loop. For each of these there are at most n iterations of the inner loop. Each iteration of the inner loop takes $\Theta(1)$ time. So the running time is $O(n.n.n.1) = O(n^3)$.

For $i \leq n/4$ (i.e. for $n/4$ iterations of the outer loop) there are $n/4$ iterations of the middle loop with $j \geq 3\,n/4$. For these, it holds that: $j - i \geq n/2$, so for these the inner loop has at least $n/2$ iterations. So the runtime is $\Omega(n/4 \,.\, n/4 \,.\, n/2 \,.1) = \Omega(n^3)$.

Example 3 of asymptotic analysis of algorithms

What is the asymptotic running time of the following algorithm?

```
ALGORITME3(n)
  s = 0
  for i = 1 to n
      for j = i to n
          for k = i to j
              s = s + 1
```

There are n iterations of the outer loop. For each of these there are at most n iterations of the middle loop. For each of these there are at most n iterations of the inner loop. Each iteration of the inner loop takes $\Theta(1)$ time. So the running time is $O(n.n.n.1) = O(n^3)$.

For $i \leq n/4$ (i.e. for $n/4$ iterations of the outer loop) there are $n/4$ iterations of the middle loop with $j \geq 3\,n/4$. For these, it holds that: $j - i \geq n/2$, so for these the inner loop has at least $n/2$ iterations. So the runtime is $\Omega(n/4 \,.\, n/4 \,.\, n/2 \,.1) = \Omega(n^3)$.

Therefore, the total runtime is $\Theta(n^3)$.

Summary

Work principle: First compare the growth rates of algorithms via asymptotic analysis, and usually implement only the one with the lowest growth rate. For two algorithms with the same growth rate, implement both and measure their running times.