Algoritmi per la Bioinformatica

Zsuzsanna Lipták

Laurea Magistrale Bioinformatica e Biotechnologie Mediche (LM9)
a.a. 2014/15, spring term

Computational efficiency II

Computational efficiency of an algorithm is measured in terms of running time and storage space.

To abstract from

- specific computers (processor speed, computer architecture, ...)
- specific programming languages
- ...

we measure

- running time in number of (basic) operations
  (e.g. additions, multiplications, comparisons, ...),
- storage space in number of storage units
  (e.g. 1 unit = 1 integer, 1 character, 1 byte, ...).

Example DP algorithm for global alignment (Needleman-Wunsch), variant which outputs only $sim(s, t)$.

Algorithm *DP algorithm for global alignment*
Input: strings s, t, with $|s| = n, |t| = m$; scoring function (p, g)
Output: value $sim(s, t)$

1. **for** $j = 0$ to m **do** $D(0, j) \leftarrow j \cdot g$;
2. **for** $i = 1$ to n **do** $D(i, 0) \leftarrow i \cdot g$;
3. **for** $i = 1$ to n **do**
4. **for** $j = 1$ to m **do**

$$D(i, j) \leftarrow \max \begin{cases} D(i-1, j) + g \\ D(i-1, j-1) + p(s_i, t_j) \\ D(i, j-1) + g \end{cases}$$

5. **return** $D(n, m)$;

Analysis of DP algorithm for global alignment:

Time

- for first row: $m+1$ operations (line 1.)
- for first column: n operations (line 2.)
- for each entry $D(i,j)$, where $1 \leq i \leq n, 1 \leq j \leq m$: 3 operations; there are $n \cdot m$ such entries: $3nm$ operations (lines 3.,4.)
- Altogether: $3nm + n + m + 1$ operations

Analysis of DP algorithm for global alignment:

Time

- for first row: $m + 1$ operations (line 1.)
- for first column: n operations (line 2.)
- for each entry $D(i, j)$, where $1 \leq i \leq n, 1 \leq j \leq m$: 3 operations; there are $n \cdot m$ such entries: $3nm$ operations (lines 3.,4.)
- Altogether: $3nm + n + m + 1$ operations

Space

- matrix of size $(n + 1)(m + 1) = nm + n + m + 1$ entries (units)

Analysis of DP algorithm for global alignment:

Time

- for first row: $m+1$ operations (line 1.)
- for first column: n operations (line 2.)
- for each entry $D(i,j)$, where $1 \leq i \leq n, 1 \leq j \leq m$: 3 operations; there are $n \cdot m$ such entries: $3nm$ operations (lines 3.,4.)
- Altogether: $3nm + n + m + 1$ operations

Space

- matrix of size $(n+1)(m+1) = nm + n + m + 1$ entries (units)

Equal length strings

If $n = m$ then time $= 3n^2 + 2n + 1$, space $= n^2 + 2n + 1$

Let's compare this with the other algorithm we saw for global alignment:

Exhaustive search

1. consider every possible alignment of s and t
2. for each of these, compute its score
3. output the maximum of these

Algorithm *Exhaustive search for global alignment*
Input: strings s, t, with $|s| = n, |t| = m$; scoring function (p, g)
Output: value $sim(s, t)$

int $max = (n + m)g$;
for each alignment A of s and t (in some order)
    do if $score(A) > max$
        then $max \leftarrow score(A)$;
return max;

Note:

1. The variable *max* is needed for storing the highest score so far seen.
2. The initial value of *max* is the score of *some* alignment of s, t (which one?)

Analysis of Exhaustive search:

- Time: next slides
- Space: exercise

Analysis of Exhaustive search (time):

- for every alignment (line 2.)
- compute its score (line 3.)

Analysis of Exhaustive search (time):

- for every alignment (line 2.) no. of al's
- compute its score (line 3.)

Analysis of Exhaustive search (time):

- for every alignment (line 2.) — no. of al's
- compute its score (line 3.) — length of al.

Analysis of Exhaustive search (time):

- for every alignment (line 2.) — no. of al's
- compute its score (line 3.) — length of al.

$$\text{time} = \underbrace{\text{no. of alignments}}_{N(n,m)} \cdot \underbrace{\text{length of alignment}}_{\text{between } \max(n,m) \text{ and } n+m}$$

Analysis of Exhaustive search (time):

- for every alignment (line 2.) — no. of al's
- compute its score (line 3.) — length of al.

$$\text{time} = \underbrace{\text{no. of alignments}}_{N(n,m)} \cdot \underbrace{\text{length of alignment}}_{\text{between } \max(n,m) \text{ and } n+m}$$

Simplify analysis: Let's look at two equal length strings $|s| = |t| = n$:

$$N(n,n) \cdot n \leq \text{time} \leq N(n,n) \cdot 2n$$

We have seen: $N(n,n) > 2^n$, so $\text{time} \geq 2^n \cdot n$.

So we have, for $|s| = |t| = n$:

- DP algo: $3n^2 + 2n + 1$ operations
- Exhaustive search: at least $N(n, n) \cdot n$ operations

Let's compare the two functions for increasing n:

n	1	2	3	4	5	...	10	100	1000
$3n^2 + 2n + 1$	6	17	34	57	86	...	321	30 201	3 002 001
$N(n, n) \cdot n$	3	26	189	1284	8415	...	$\approx 80 \cdot 10^6$	$\approx 2 \cdot 10^{77}$	$\approx 10^{700}$

The DP algorithm is much faster than the exhaustive search algorithm, because its running time increases much slower as the input size increases. But how much?

Algorithm analysis

- We measure running time and storage space, measured in no. of operations and no. of storage units.

Algorithm analysis

- We measure running time and storage space, measured in no. of operations and no. of storage units.
- We want to know how our algo performs depending on the size of the input (bigger input = more time/space), i.e. as functions of the input size (usually denoted n, m).

Algorithm analysis

- We measure running time and storage space, measured in no. of operations and no. of storage units.
- We want to know how our algo performs depending on the size of the input (bigger input = more time/space), i.e. as functions of the input size (usually denoted n, m).
- We are interested in the algorithm's behaviour for large inputs.

Algorithm analysis

- We measure running time and storage space, measured in no. of operations and no. of storage units.
- We want to know how our algo performs depending on the size of the input (bigger input = more time/space), i.e. as functions of the input size (usually denoted n, m).
- We are interested in the algorithm's behaviour for large inputs.
- We want to know the growth behaviour, i.e. how time/space requirements change as input increases.

Algorithm analysis

- We measure running time and storage space, measured in no. of operations and no. of storage units.
- We want to know how our algo performs depending on the size of the input (bigger input = more time/space), i.e. as functions of the input size (usually denoted n, m).
- We are interested in the algorithm's behaviour for large inputs.
- We want to know the growth behaviour, i.e. how time/space requirements change as input increases.
- We want an upper bound, i.e. on any input how much time/space needed at most? (worst-case analysis)

Consider 3 algorithms $\mathcal{A}, \mathcal{B}, \mathcal{C}$:

		input size n		
	running t.	10	20	What happened when input doubled?
$\mathcal{A}$	n	10		
$\mathcal{B}$	n^2	100		
$\mathcal{C}$	2^n	1024		

Consider 3 algorithms $\mathcal{A}, \mathcal{B}, \mathcal{C}$:

		input size n		
	running t.	10	20	What happened when input doubled?
$\mathcal{A}$	n	10	20	
$\mathcal{B}$	n^2	100	400	
$\mathcal{C}$	2^n	1024	1 048 576	

Consider 3 algorithms $\mathcal{A}, \mathcal{B}, \mathcal{C}$:

		input size n		
	running t.	10	20	What happened when input doubled?
$\mathcal{A}$	n	10	20	doubled
$\mathcal{B}$	n^2	100	400	
$\mathcal{C}$	2^n	1024	1 048 576	

Consider 3 algorithms $\mathcal{A}, \mathcal{B}, \mathcal{C}$:

		input size n		
	running t.	10	20	What happened when input doubled?
$\mathcal{A}$	n	10	20	doubled
$\mathcal{B}$	n^2	100	400	quadrupled
$\mathcal{C}$	2^n	1024	1 048 576	

Consider 3 algorithms $\mathcal{A}, \mathcal{B}, \mathcal{C}$:

		input size n		
	running t.	10	20	What happened when input doubled?
$\mathcal{A}$	n	10	20	doubled
$\mathcal{B}$	n^2	100	400	quadrupled
$\mathcal{C}$	2^n	1024	1 048 576	squared

Consider 3 algorithms $\mathcal{A}, \mathcal{B}, \mathcal{C}$:

	running t.	input size n 10	20	What happened when input doubled?
$\mathcal{A}$	n	10	20	doubled
$\mathcal{B}$	n^2	100	400	quadrupled
$\mathcal{C}$	2^n	1024	1 048 576	squared

Now 3 algorithms $\mathcal{A}', \mathcal{B}', \mathcal{C}'$:

	running t.	input size n 10	20	What happened when input doubled?
$\mathcal{A}'$	$3n$	30	60	
$\mathcal{B}'$	$3n^2$	300	1200	
$\mathcal{C}'$	$3 \cdot 2^n$	3072	3 145 728	

Consider 3 algorithms $\mathcal{A}, \mathcal{B}, \mathcal{C}$:

	running t.	input size n 10	20	What happened when input doubled?
$\mathcal{A}$	n	10	20	doubled
$\mathcal{B}$	n^2	100	400	quadrupled
$\mathcal{C}$	2^n	1024	1 048 576	squared

Now 3 algorithms $\mathcal{A}', \mathcal{B}', \mathcal{C}'$:

	running t.	input size n 10	20	What happened when input doubled?
$\mathcal{A}'$	$3n$	30	60	doubled
$\mathcal{B}'$	$3n^2$	300	1200	quadrupled
$\mathcal{C}'$	$3 \cdot 2^n$	3072	3 145 728	1/3 of squared

The O-notation allows us to abstract from constants ($3n$ vs. n) and other details which are not important for the growth behaviour of functions.

Definition (O-classes)

Given a function $f : \mathbb{N} \to \mathbb{R}$, then $O(f(n))$ is the class (set) of functions $g(n)$ s.t.:

There exists a $c > 0$ and an $n_0 \in \mathbb{N}$ s.t. for all $n \geq n_0$: $g(n) \leq c \cdot f(n)$.

The O-notation allows us to abstract from constants ($3n$ vs. n) and other details which are not important for the growth behaviour of functions.

Definition (O-classes)

Given a function $f : \mathbb{N} \to \mathbb{R}$, then $O(f(n))$ is the class (set) of functions $g(n)$ s.t.:

There exists a $c > 0$ and an $n_0 \in \mathbb{N}$ s.t. for all $n \geq n_0$: $g(n) \leq c \cdot f(n)$.

We then say that

$$g(n) \in O(f(n)) \qquad \text{or} \qquad \underbrace{g(n) = O(f(n))}_{\text{Careful, this is not an "equality"!}}$$

The O-notation allows us to abstract from constants ($3n$ vs. n) and other details which are not important for the growth behaviour of functions.

Definition (O-classes)

Given a function $f : \mathbb{N} \to \mathbb{R}$, then $O(f(n))$ is the class (set) of functions $g(n)$ s.t.:

There exists a $c > 0$ and an $n_0 \in \mathbb{N}$ s.t. for all $n \geq n_0$: $g(n) \leq c \cdot f(n)$.

We then say that

$$g(n) \in O(f(n)) \qquad \text{or} \qquad \underbrace{g(n) = O(f(n))}_{\text{Careful, this is not an "equality"!}}$$

Meaning: "g is smaller or equal than f (w.r.t. growth behaviour)"
"g does not grow faster than f"

Example

$3n^2 + 2n + 1 \in O(n^2)$

Example

$3n^2 + 2n + 1 \in O(n^2)$

Recall definition

$g(n) \in O(f(n))$ if
there exists a $c > 0$ and an $n_0 \in \mathbb{N}$ s.t. for all $n \geq n_0$: $g(n) \leq c \cdot f(n)$.

Example

$3n^2 + 2n + 1 \in O(n^2)$

Recall definition

$g(n) \in O(f(n))$ if
there exists a $c > 0$ and an $n_0 \in \mathbb{N}$ s.t. for all $n \geq n_0$: $g(n) \leq c \cdot f(n)$.

Proof

n	1	2	3	4	5
$3n^2 + 2n + 1$	6	17	34	57	86
$4n^2$	4	16	36	64	100

Example

$3n^2 + 2n + 1 \in O(n^2)$

Recall definition

$g(n) \in O(f(n))$ if
there exists a $c > 0$ and an $n_0 \in \mathbb{N}$ s.t. for all $n \geq n_0$: $g(n) \leq c \cdot f(n)$.

Proof

Choose $c = 4$ and $n_0 = 3$. We have: $\forall n \geq 3 : \quad 3n^2 + 2n + 1 \leq 4n^2$.

n	1	2	3	4	5
$3n^2 + 2n + 1$	6	17	34	57	86
$4n^2$	4	16	36	64	100

Example

$3n^2 + 2n + 1 \in O(n^2)$

Recall definition

$g(n) \in O(f(n))$ if
there exists a $c > 0$ and an $n_0 \in \mathbb{N}$ s.t. for all $n \geq n_0$: $g(n) \leq c \cdot f(n)$.

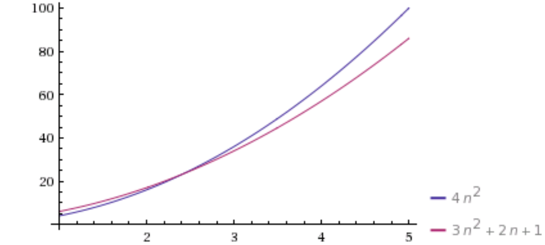
Proof

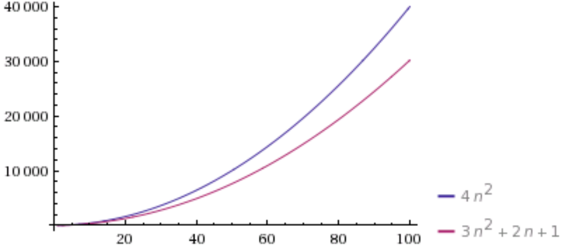
Choose $c = 4$ and $n_0 = 3$. We have: $\forall n \geq 3 : \quad 3n^2 + 2n + 1 \leq 4n^2$.

n	1	2	3	4	5
$3n^2 + 2n + 1$	6	17	34	57	86
$4n^2$	4	16	36	64	100

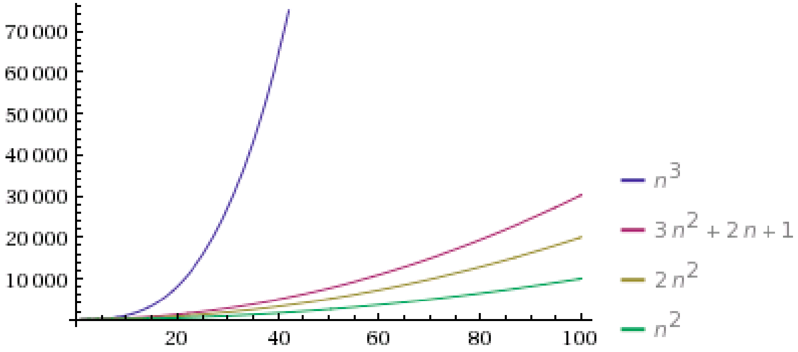
$$
\begin{aligned}
& 3n^2 + 2n + 1 \leq 4n^2 \\
\Leftrightarrow \quad & n^2 - 2n - 1 \geq 0 \\
\Leftrightarrow \quad & (n-1)^2 - 2 \geq 0 \\
\Leftrightarrow \quad & (n-1)^2 \geq 2 \\
\Leftrightarrow \quad & n \geq 3
\end{aligned}
$$

$3n^2 + 2n + 1 \in O(n^2)$: $\forall n \geq 3: \quad 3n^2 + 2n + 1 \leq 4n^2$

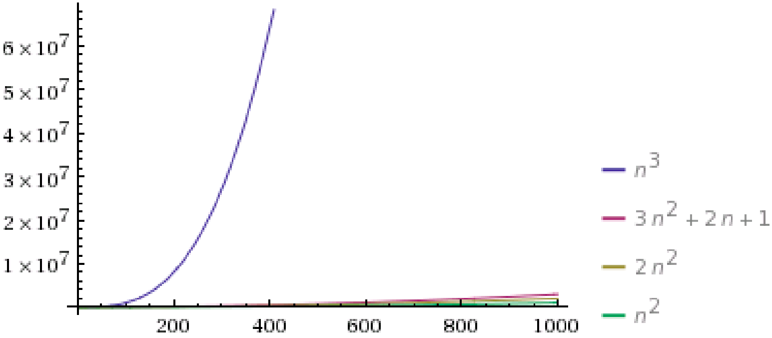




plot: WolframAlpha



plot: WolframAlpha



plot: WolframAlpha

In practice:

- identify which input parameters are important—no. months n for Fibonacci numbers; length of strings n, m for pairwise al.
- order additive terms according to these in decreasing growth order:
  $3n^5 + 2n^3 + n + 7$,
  $3nm + n + m + 1$
- take largest without multiplicative constant:
  $3n^5 + 2n^3 + n + 7 \in O(n^5)$,
  $3nm + n + m + 1 \in O(nm)$

Important O-classes

The most important functions, ordered by increasing O–classes: each function f_i is in the O–class of the next function f_{i+1}, but $f_{i+1}(n) \notin O(f_i(n))$.

1	$\log\log n$	$\log n$	$\sqrt{n}$	n	$n \log n$	n^2	n^3	...	...	2^n	$n!$	n^n
constant		logarithmic		linear		quadratic	cubic			exponential		
			polynomial (of the form n^c for some constant c) (all except $n \log n$ are polynomials)									
EFFICIENT[1]									inefficient			

function grows slower $\longleftrightarrow$ *function grows faster*

faster algorithm — *slower algorithm*

[1] also called *feasible* vs. *infeasible*

Amount of time an algorithm of time complexity $f(n)$ would need on a computer that performs one million operations per second:

$f(n)$	$n = 50$	$n = 100$	$n = 200$
n	$5 \cdot 10^{-5}$ s	10^{-4} s	
n^2	0.0025 s	0.01 s	
n^3	0.125 s	1 s	
1.1^n	0.0001 s	0.014 s	
2^n	35.7 years	$4 \cdot 10^{16}$ years	

Amount of time an algorithm of time complexity $f(n)$ would need on a computer that performs one million operations per second:

$f(n)$	$n = 50$	$n = 100$	$n = 200$
n	$5 \cdot 10^{-5}$ s	10^{-4} s	$2 \cdot 10^{-4}$ s
n^2	0.0025 s	0.01 s	0.04 s
n^3	0.125 s	1 s	8 s
1.1^n	0.0001 s	0.014 s	190 s
2^n	35.7 years	$4 \cdot 10^{16}$ years	$5 \cdot 10^{46}$ years

On a 1000 times faster computer:

$f(n)$	$n = 50$	$n = 100$	$n = 200$
n	$5 \cdot 10^{-8}$ s	10^{-7} s	$2 \cdot 10^{-7}$ s
n^2	$2.5 \cdot 10^{-6}$ s	10^{-5} s	$4 \cdot 10^{-5}$ s
n^3	$1.25 \cdot 10^{-4}$ s	10^{-3} s	$8 \cdot 10^{-3}$ s
1.1^n	$1.1 \cdot 10^{-7}$ s	$1.4 \cdot 10^{-5}$ s	0.19 s
2^n	13 days	$4 \cdot 10^{13}$ years	$5 \cdot 10^{43}$ years

Looking at it in a different way ...

	1	2	3	4	5	...	10	20	100	1000	10^6
n	1	2	3	4	5	...	10	20	100	1000	10^6
n^2	1	4	9	16	25	...	100	400	10000	10^6	
2^n	2	4	8	16	32	...	1024	$\approx 10^6$	$\approx 10^{30}$	$\approx 10^{301}$	

Looking at it in a different way ...

	1	2	3	4	5	...	10	20	100	1000	10^6
n	1	2	3	4	5	...	10	20	100	1000	10^6
n^2	1	4	9	16	25	...	100	400	10000	10^6	
2^n	2	4	8	16	32	...	1024	$\approx 10^6$	$\approx 10^{30}$	$\approx 10^{301}$	

On a computer that can perform one million operations per second, in a second,

- a linear-time algorithm can solve a problem instance of size 10^6 (one million) (e.g. fib2, fib3),
- a quadratic-time algorithm one of size 1000 (one thousand),
- an exponential-time algorithm one of size 20 (e.g. fib1).

In fact, on any computer, these algorithms need always the same amount of time for problem instances of such different sizes!

Back to the global alignment algorithms:

- $A(n) := 3n^2 + 2n + 1$ running time of DP algo
- $B(n) := n \cdot N(n, n)$ running time of exhaustive search algo

	1	2	3	4	5	...	10	20	100	1000
$A(n)$	6	17	34	57	86	...	321	1241	30 201	3 002 001
$B(n)$	3	26	189	1284	8415	...	$\approx 80 \cdot 10^6$	$\approx 5 \cdot 10^{16}$	$\approx 2 \cdot 10^{77}$	$\approx 10^{700}$
n	1	2	3	4	5	...	10	20	100	1000
n^2	1	4	9	16	25	...	100	400	10 000	10^6
2^n	2	4	8	16	32	...	1024	$\approx 10^6$	$\approx 10^{30}$	$\approx 10^{301}$

- $A(n) \in O(n^2)$ a quadratic time algorithm
- $B(n)$ is super-exponential

Analysis of our alignment algorithms

algorithm	time	space
DP for global alignment, only $sim(s,t)$	$O(nm)$	$O(nm)$
[equal length strings	$O(n^2)$	$O(n^2)$]
computing an optimal alignment	$O(n+m)$	none[1]
[equal length strings	$O(n)$	none[1]]
space saving variant of DP for global alignment, only $sim(s,t)$	$O(nm)$	$O(\min(n,m))$
[equal length strings	$O(n^2)$	$O(n)$]
DP for local alignment	$O(nm)$	$O(nm)$
[equal length strings	$O(n^2)$	$O(n^2)$]

[1] assuming the $O(n^2)$ size DP-table is given