SFU CMPT-307 2008-2 Lecture: Week 1

Ján Maňuch

E-mail: jmanuch@sfu.ca

Lecture on May 6, 2008, 5.30pm-8.20pm

2008 Ján Maňuch

CMPT 307 – Data Structures and Algorithms

Instructor: Jan (Jano) Maňuch (jmanuch@sfu.ca)

TA: Louisa Harutyunyan (lha22@fas.sfu.ca)

Homepage: www.cs.sfu.ca/CC/307/jmanuch

check for lecture notes, assignments, solutions, important dates!!

2008 Ján Maňuch

Organization of lectures

- **Option 1.** 50 minutes lecture + 10 minutes break + 50 minutes lecture + 10 minutes break + 50 minutes lecture
- **Option 2.** 75 minutes lecture + 10 minutes break + 75 minutes lecture
- **Option 3.** 150 minutes lecture
- **Option X.** Other suggestions?

Content

- Mathematical preliminaries (asymptotic notation, recurrences)
- Sorting and selecting
- Dictionaries
- Search trees
- Greedy algorithms
- Dynamic programming
- Approximation algorithms

Textbook:

- *Introduction to Algorithms* (2nd edition), T.H. Cormen, C.E. Leiserson, R.L. Rivest, McGraw Hill

Analysing Algorithms

Usually interested in **running time** (but sometimes also memory requirements).

Example: One of he simplest simplest sorting algorithms

SELECTION-SORT

Input: n numbers in array $A[1], \ldots, A[n]$

```
1:  for i ← 1 to n − 1 do
2:      /* 3–10: find min in A[i], ..., A[n] */
3:      sm_elem ← A[i]
4:      sm_pos ← i
5:      for j ← i + 1 to n do
6:          if A[j] < sm_elem then
7:              sm_elem ← A[j]
8:              sm_pos ← j
9:          end if
10:     end for
11:     swap A[i] and A[sm_pos]
12: end for
```

Example: $n = 6$, $A = [14, 13, 12, 15, 16, 11]$

i	sm_elem	sm_pos	"new" $A[]$
1	11	6	$[11, 13, 12, 15, 16, 14]$
2	12	3	$[11, 12, 13, 15, 16, 14]$
3	13	3	$[11, 12, 13, 15, 16, 14]$
4	14	6	$[11, 12, 13, 14, 16, 15]$
5	15	6	$[11, 12, 13, 14, 15, 16]$

Correctness

the loop invariant: **after i-th loop, the elements $A[1] \ldots A[i]$ are sorted and are all smaller than the elements $A[i+1] \ldots A[n]$**

initialization: after 1st step $A[1]$ contains the minimal element $\Longrightarrow$ LI is true

maintenance: if it is true after i-th iteration, it remains true after $(i+1)$-th iteration

termination: after $(n-1)$-th loop $A[1] \ldots A[n-1]$ are sorted and all smaller than $A[n]$

SELECTION-SORT

Input: n numbers in array $A[1], \ldots, A[n]$

```
1:  for i ← 1 to n − 1 do
2:      /* 3–10: find min in A[i], ..., A[n] */
3:      sm_elem ← A[i]
4:      sm_pos ← i
5:      for j ← i + 1 to n do
6:          if A[j] < sm_elem then
7:              sm_elem ← A[j]
8:              sm_pos ← j
9:          end if
10:     end for
11:     swap A[i] and A[sm_pos]
12: end for
```

Maintenance

the loop invariant: **after i-th loop, the elements $A[1] \ldots A[i]$ are sorted and are all smaller than the elements $A[i+1] \ldots A[n]$**

maintenance: if it is true after i-th iteration, it remains true after $(i+1)$-th iteration

Proof.

- before $(i+1)$-th loop, $A[1] \ldots A[i]$ are sorted and all smaller than the elements $A[i+1] \ldots A[n]$
- in the $(i+1)$-th loop, we find the smallest element in $A[i+1] \ldots A[n]$ and put it to the position $A[i+1]$
- since, this element was greater than any element in $A[1] \ldots A[i]$, $A[1] \ldots A[i+1]$ is still sorted
- obviously, $A[1] \ldots A[i]$ are still smaller than $A[i+2] \ldots A[n]$
- since, $A[i+1]$ is at the end of the $(i+1)$-th loop the smallest element in $A[i+1] \ldots A[n]$, $A[1] \ldots A[i+1]$ are smaller than $A[i+2] \ldots A[n]$
- hence, the loop invariant holds after $(i+1)$-th loop as well.

Assignment 1

Assignment Problem 1.1. (deadline: May 13, 5:25pm)
Consider the following sorting algorithm:

BUBBLE-SORT

Input: n numbers in array $A[1], \ldots, A[n]$

```
1: for i ← 1 to n do
2:     for j ← n downto i + 1 do
3:         if A[j − 1] > A[j] then
4:             swap A[j − 1] and A[j]
5:         end if
6:     end for
7: end for
```

Find a loop invariant of the main loop and use it to prove that algorithm is correct!

Running time

Clearly depending on n (loops depend on n)

– Body of outer loop (over i) is executed $n-1$ times

– Each increment takes constant time, say c_1

– Lines 3 and 4 both take constant time, say c_2

– Body of inner loop (over j) is executed $n-i$ times

– Again, each increment takes time c_1

– Suppose comparison in line 6 takes constant time, c_3

– If condition is true, then another $2 \cdot c_2$, otherwise 0

– Thus lines 6–9 take at most $c_3 + 2 \cdot c_2$

– Swap in line 11 takes constant time, say c_4

SELECTION-SORT

Input: n numbers in array $A[1], \ldots, A[n]$

```
1:  for i ← 1 to n − 1 do
2:     /* 3–10: find min in A[i], ..., A[n] */
3:     sm_elem ← A[i]
4:     sm_pos ← i
5:     for j ← i + 1 to n do
6:        if A[j] < sm_elem then
7:           sm_elem ← A[j]
8:           sm_pos ← j
9:        end if
10:    end for
11:    swap A[i] and A[sm_pos]
12: end for
```

Putting everything together: (the worst case)

$$\sum_{i=1}^{n-1}\left[c_1 + 2\cdot c_2 + \left(\sum_{j=i+1}^{n} c_1 + c_3 + 2\cdot c_2\right) + c_4\right]$$

that's rather ugly for such a trivial algorithm...

Let's simplify this expression a bit:

Let $d_1 = c_1 + 2\cdot c_2 + c_4$
Let $d_2 = c_1 + c_3 + 2\cdot c_2$

Note: d_1 and d_2 are constants

Now we have

$$\sum_{i=1}^{n-1}\left[d_1 + \left(\sum_{j=i+1}^{n} d_2\right)\right]$$

Note that $\sum_{j=i+1}^{n} d_2 = (n-i)\cdot d_2 = n\cdot d_2 - i\cdot d_2$

This results in

$$\sum_{i=1}^{n-1} (d_1 + n \cdot d_2 - i \cdot d_2)$$

But we're not quite done: terms d_1 and $n \cdot d_2$ do not depend on i, so this is equal to

$$(n-1) \cdot d_1 + (n-1) \cdot n \cdot d_2 - d_2 \cdot \sum_{i=1}^{n-1} i$$

We know that

$$\sum_{i=1}^{k} i = 1 + 2 + 3 + \cdots + k = \frac{k \cdot (k+1)}{2}$$

Thus "our" expression becomes

$$\begin{aligned} & (n-1)\cdot d_1 + (n-1)\cdot n\cdot d_2 - d_2\cdot\frac{(n-1)\cdot n}{2} \\ = \quad & (n-1)\cdot d_1 + d_2\cdot\frac{(n-1)\cdot n}{2} \\ = \quad & n\cdot d_1 - d_1 + n^2\cdot d_2/2 - n\cdot d_2/2 \\ = \quad & n^2\cdot d_2/2 + n\cdot(d_1 - d_2/2) - d_1 \end{aligned}$$

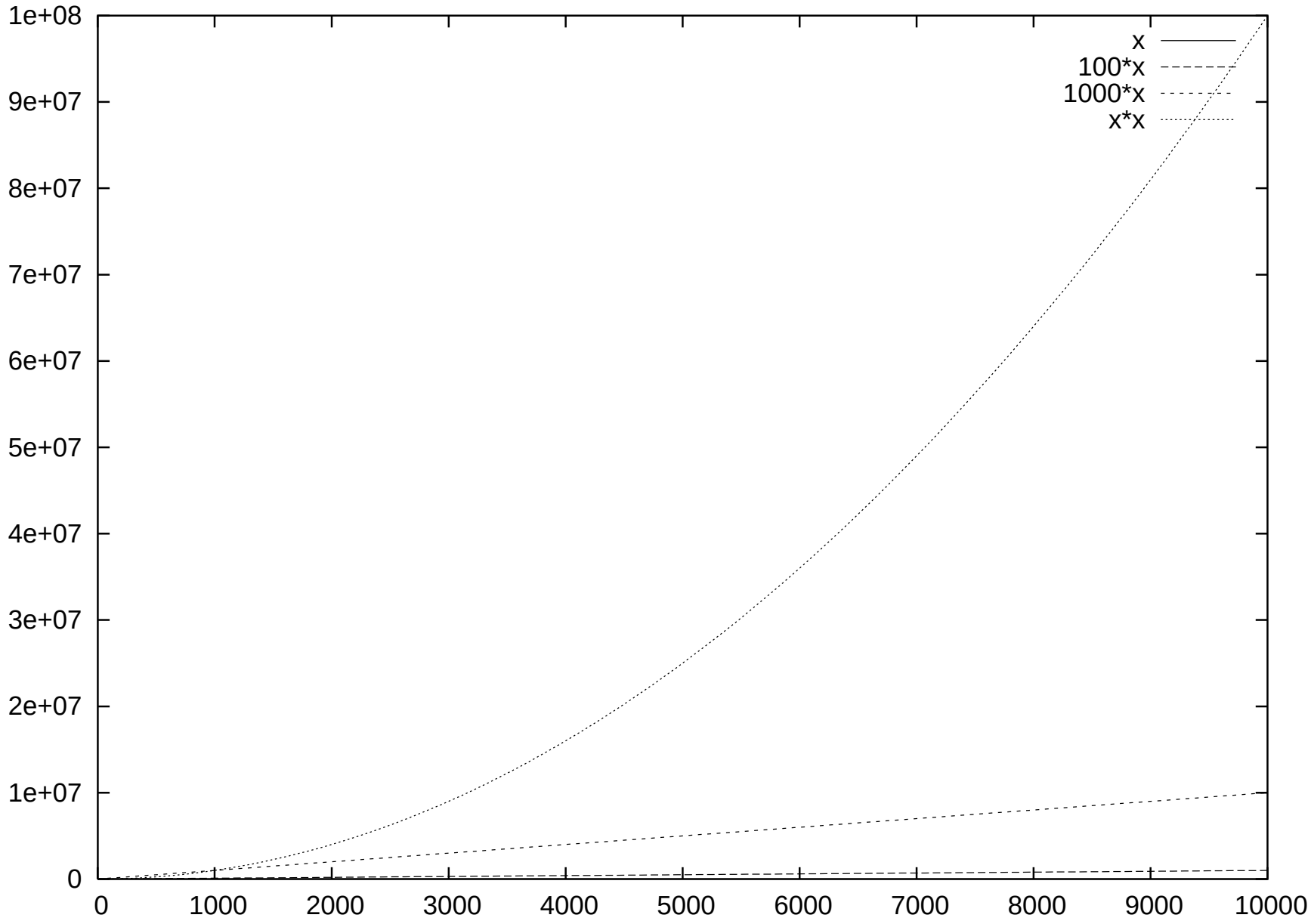
With $e_1 = d_2/2$ and $e_2 = d_1 - d_2/2$ (note: e_1 and e_2 are constants) we obtain

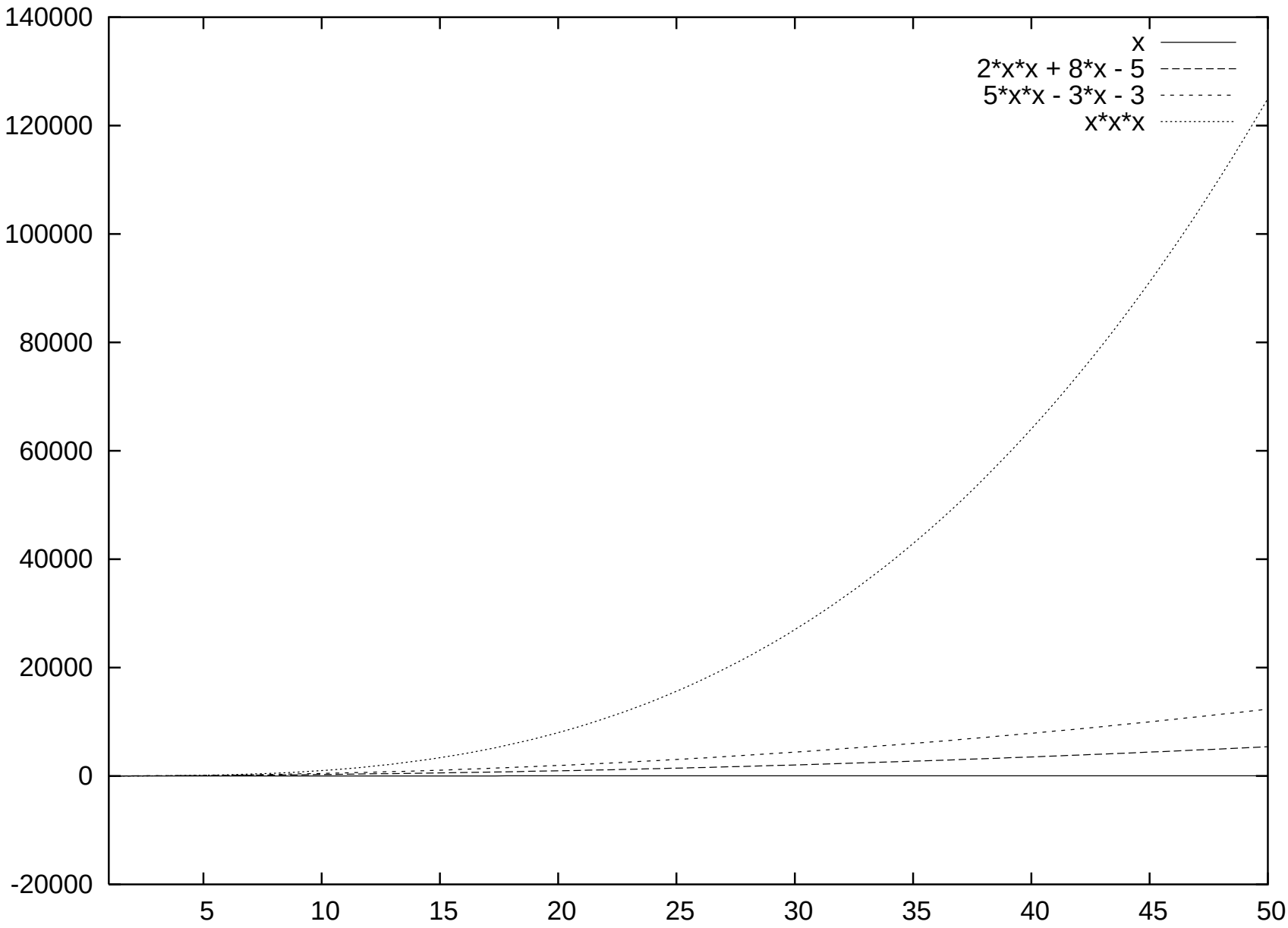
$$e_1\cdot n^2 + e_2\cdot n - d_1$$

Now, that was rather. . . cumbersome

Since e_1, e_2, and d_1 are constants, the statement here seems to be that the running time depends **quadratically** on n (modulo some "smaller terms")

This is the idea behind **asymptotic analysis**: we don't care about constants (either multiplicative or additive), or about lower-order terms.


1e+08
9e+07
8e+07
7e+07
6e+07
5e+07
4e+07
3e+07
2e+07
1e+07
0
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
x
100*x
1000*x
x*x


140000
120000
100000
80000
60000
40000
20000
0
-20000
5
10
15
20
25
30
35
40
45
50
x
2*x*x + 8*x - 5
5*x*x - 3*x - 3
x*x*x

Somewhat more formal:

If $T(n)$ is some algorithm's **running time function**, i.e., given input of size n, it runs $T(n)$ steps, we are interested in

asymptotic behaviour
(read: as n approaches infinity)

rather than

the **exact functions**.

We want compare the **growth** of $T(n)$ with the growth of some simple function $f(n)$.

In example: the running time function of SELECTION-SORT grows pretty much like n^2.

We're going to formalise this notion in the following.

Theta-notation

For a given function $g(n)$, $\Theta(g(n))$ denotes the set

$$\Theta(g(n)) = \{f(n) : \quad \text{there exist positive constants } c_1, c_2, n_0 \text{ such that } c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0\}$$

Intuition: $f(n)$ belongs to the family $\Theta(g(n))$ if $\exists$ constants c_1, c_2 s.t. $f(n)$ can fit between $c_1 \cdot g(n)$ and $c_2 \cdot g(n)$, for all n sufficiently large.

Correct notation: $f(n) \in \Theta(g(n))$

Usually used: $f(n) = \Theta(g(n))$.

We also say that "$f(n)$ is in $\Theta(g(n))$".

Examples of Θ-notation:

$f(n) = 2n^2 = \Theta(n^2)$
because with $g(n) = n^2$ and $c_1 = 1$ and $c_2 = 2$ we have
$0 \leq c_1 g(n) \leq f(n) = 2 \cdot n^2 \leq c_2 \cdot g(n)$.

$f(n) = 8n^5 + 17n^4 - 25 = \Theta(n^5)$
because $f(n) \geq 7 \cdot n^5$ for n large enough

n	$8n^5 + 17n^4 - 25$	n^5	$7n^5$
1	$8 \cdot 1 + 17 \cdot 1 - 25 = 0$	1	7
2	$8 \cdot 32 + 17 \cdot 16 - 25 = 503$	32	224

and $f(n) \leq 8n^5 + 17n^5 = 25n^5$, thus $c_1 = 7$, $c_2 = 25$ and $n_0 = 2$ are good enough.

More intuition: for all $n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor.

We say that $g(n)$ is an **asymptotically tight bound** for $f(n)$.

Back to sorting example

We had running time $T(n) = e_1 \cdot n^2 + e_2 \cdot n - d_1$. Now we can say: $T(n) = \Theta(n^2)$,

we have formal means to "get rid" of lower-order terms and constant coefficients.

Why is this true?

Must find positive c_1, c_2, n_0 such that

$$c_1 n^2 \leq e_1 n^2 + e_2 n - d_1 \leq c_2 n^2$$

for all $n \geq n_0$.

Dividing by n^2 gets us

$$c_1 \leq e_1 + \frac{e_2}{n} - \frac{d_1}{n^2} \leq c_2$$

Suppose e_1, e_2, d_1 are positive (other cases similar).

Obviously, for $n \geq e_2$ we have $e_2/n \leq 1$ and thus
$e_1 + e_2/n - d_1/n^2 \leq e_1 + 1$ and thus $c_2 = e_1 + 1$ and $n_0 = e_2$ does the job for the right-hand inequality.

Also, for $n^2 \geq d_1 \Leftrightarrow n \geq \sqrt{d_1}$ we have $d_1/n^2 \leq 1$ and thus
$e_1 + e_2/n - d_1/n^2 \geq e_1 - 1$ and therefore $c_1 = e_1 - 1$ is sufficient.

With $n_0 = \max\{e_2, \sqrt{d_1}\}$ both conditions are fulfilled simultaneously.

Exercise 1.1. Recall the formula for Selection-Sort:

$$\sum_{i=1}^{n-1}\left[c_1 + 2\cdot c_2 + \left(\sum_{j=i+1}^{n} c_1 + c_3 + 2\cdot c_2\right) + c_4\right]$$

(a) Assume that in Selection-Sort algorithm, $c_1 = 2$, $c_2 = 3$, $c_3 = 5$ and $c_4 = 4$ machine cycles. Find the exact formula for the running time (upper bound) of the algorithm.

(b) Show that the above running times $T(n)$ (with fixed c_1, c_2, c_3, c_4 above) is in $\Theta(n^2)$: find positive constants α_1, α_2, n_0 such that

$$\alpha_1.n^2 \leq T(n) \leq \alpha_2.n^2,$$

for all $n \geq n_0$.

Theta notation respects the main term

However, $n^3 \neq \Theta(n^2)$

Recall: for $n^3 = \Theta(n^2)$ we would have to find constants c_1, c_2, n_0 with

$$0 \leq c_1 n^2 \leq n^3 \leq c_2 n^2$$

for $n \geq n_0$.

Intuition: there's a factor of n between both functions, thus we **cannot** find a constant c_2!

Suppose, for purpose of contradiction, that there **are** constants c_2 and n_0 with $n^3 \leq c_2 \cdot n^2$ for $n \geq n_0$.

Dividing by n^2 yields $n \leq c_2$, which cannot possibly hold for arbitrarily large n (c_2 must be a constant).

Also: $n \neq \Theta(n^2)$

Same idea...

Suppose it's true, i.e., there are constants c_1 and n_0 such that

$$c_1 n^2 \leq n \Leftrightarrow c_1 n \leq 1 \Leftrightarrow n \leq 1/c_1$$

Once more:

Θ-notation is about **asymptotic equality**

("disregarding" lower-order terms and constant coefficients by choosing suitable c_1, c_2, n_0)

We've just seen one of the **most important** (and sometimes most annoying) gaps between theory and practice:

In **theory** a factor of 1,000 doesn't make one bit of a difference (just choose your c_2 accordingly),

whereas in **practice** it does (there, even a factor of 2 may decide on whether the graphics run smoothly or not).

There's this nice saying:

> "*In theory, practice and theory are the same. In practice, they're not.*"
>
> (source unknown)

Exercise 1.2. Consider positive functions $f(n)$, $g(n)$ and $h(n)$. Prove that

(a) $f(n) \in \Theta(f(n))$;

(b) if $f(n) \in \Theta(g(n))$ then $g(n) \in \Theta(f(n))$;

(c) if $f(n) \in \Theta(g(n))$ and $g(n) \in \Theta(h(n))$ then $f(n) \in \Theta(h(n))$.

Assignment 1 (continued)

Assignment Problem 1.2. (deadline: May 13, 5:25pm)
Show that for any real constants a and b, where $b > 0$,

(a) $b^{n+a} \in \Theta(b^n)$;

(b) $(n+a)^b \in \Theta(n^b)$.

Big-O-notation

We've seen: Θ-notation asymptotically bounds **from above and below.**

When we're interested in **asymptotic upper bounds** only, we use $\mathcal{O}$-notation (read: "big-O").

For given function $g(n)$, define $\mathcal{O}(g(n))$ (read: "big-O of g of n" or also "order g of n") as follows:

$$\mathcal{O}(g(n)) = \{f(n) : \quad \text{there exist positive constants } c, n_0 \text{ such that } f(n) \leq c \cdot g(n) \text{ for all } n \geq n_0\}$$

We write $f(n) = \mathcal{O}(g(n))$ to indicate that $f(n)$ is member of set $\mathcal{O}(g(n))$.

Obviously, $f(n) = \Theta(g(n))$ implies $f(n) = \mathcal{O}(g(n))$; we just drop the left inequality in the definition of $\Theta(g(n))$.

Back to our **sorting example** with running time $T(n) = e_1 n^2 + e_2 n - d_1$, we can say $T(n) = \mathcal{O}(n^2)$.

This means:
the running time of SELECTION-SORT is asymptotically at most n^2, i.e., the "real" running time of SELECTION-SORT is at most a constant factor greater than n^2.

Also: now we have, e.g., $n = O(n^2)$ because $n \leq 1 \cdot n^2$ for all n (thus $c = n = 1$ does the job).

Intuition: $\mathcal{O}$-notation is used to denote upper bounds on running times, memory requirements, etc.

Saying "the running time is $\mathcal{O}(n \log n)$" means: the running time is not greater than $n \log n$ times some constant factor, for n large enough.

Computing upper bound

SELECTION-SORT

Input: n numbers in array $A[1], \ldots, A[n]$

```
1:  for i ← 1 to n − 1 do
2:      /* 3–10: find min in A[i], ..., A[n] */
3:      sm_elem ← A[i]
4:      sm_pos ← i
5:      for j ← i + 1 to n do
6:          if A[j] < sm_elem then
7:              sm_elem ← A[j]
8:              sm_pos ← j
9:          end if
10:     end for
11:     swap A[i] and A[sm_pos]
12: end for
```

– lines 3–4: $\mathcal{O}(1)$

– lines 6–9: $\mathcal{O}(1)$

– lines 5–10: $\mathcal{O}(n)$ times $\mathcal{O}(1)$, which is $\mathcal{O}(n)$

– lines 3–11: $\mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n) = \mathcal{O}(n)$

– the whole algorithms: $\mathcal{O}(n)$ times $\mathcal{O}(n)$, which is $\mathcal{O}(n^2)$

Assignment 1 (continued)

Assignment Problem 1.3. (deadline: May 13, 5:25pm)
Does $f(n) \in \mathcal{O}(g(n))$ implies

(a) $g(n) \in \mathcal{O}(f(n))$?

(b) $\frac{1}{g(n)} \in \mathcal{O}(\frac{1}{f(n)})$?

If does prove it, if does not, show an example of two functions $f(n)$ and $g(n)$ which satisfy $f(n) \in \mathcal{O}(g(n))$, but do not satisfy the condition (a) (respectively, (b)).

Big-Omega-notation

Like $\mathcal{O}$-notation, but for lower bounds

For a given function $g(n)$, $\Omega(n)$ denotes the set

$$\Omega(g(n)) = \{f(n) : \quad \text{there exist positive constants } c, n_0 \text{ such that } c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0\}$$

Saying $T(n) = \Omega(n^2)$ means growth of $T(n)$ is at least the of n^2.

Clearly, $f(n) = \Theta(g(n))$ iff $f(n) = \Omega(g(n))$ and $f(n) = \mathcal{O}(g(n))$.

o-notation

Similar to $\mathcal{O}$

$f(n) = \mathcal{O}(g(n))$ means we can upper-bound the growth of f by the growth of g (up to a constant factor)

$f(n) = o(g(n))$ is the same, **except** we require the growth of f to be **strictly** smaller than the growth of g:

For a given function $g(n)$, $o(n)$ denotes the set

$$o(g(n)) = \{f(n) : \quad \text{for } \textbf{any} \text{ pos constant } c$$
$$\text{there exists a pos constant } n_0$$
$$\text{such that}$$
$$f(n) < c \cdot g(n)$$
$$\text{for all } n \geq n_0\}$$

Intuition: $f(n)$ becomes insignificant relative to $g(n)$ as n approaches infinity:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

In other words, f is $o(\text{something})$ if there is **no** constant factor between f and something.

Examples:

$n = o(n^2)$

$\log n = o(n)$

$n = o(2^n)$

$n^{1,000} = o(1.0001^n)$

$1 = o(\log n)$

omega-notation

ω is to Ω what o is to $\mathcal{O}$:

$f(n) = \omega(g(n))$ iff $g(n) = o(f(n))$

For a given function $g(n)$, $\omega(n)$ denotes the set

$$\omega(g(n)) = \{f(n) : \quad \text{for } \textbf{any} \text{ pos constant } c$$
$$\text{there exists a pos constant } n_0$$
$$\text{such that}$$
$$c \cdot g(n) < f(n)$$
$$\text{for all } n \geq n_0\}$$

In other words:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

if the limit exists.

I.e., $f(n)$ becomes **arbitrarily** large relative to $g(n)$.

Relational properties of asymptotic notation

So we have

- Θ: asymptotically "equal"
- $\mathcal{O}$: asymptotically "at most"
- Ω: asymptotically "at least"
- o: asymptotically "strictly smaller"
- ω: asymptotically "strictly greater"

Many **relational properties** of real numbers hold for functions as well.

Transitivity:

$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$

$f(n) = \mathcal{O}(g(n)) \wedge g(n) = \mathcal{O}(h(n)) \Rightarrow f(n) = \mathcal{O}(h(n))$

$f(n) = \Omega(g(n)) \wedge g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$

$f(n) = o(g(n)) \wedge g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$

$f(n) = \omega(g(n)) \wedge g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$

Reflexivity:

$f(n) = \Theta(f(n))$

$f(n) = \mathcal{O}(f(n))$

$f(n) = \Omega(f(n))$

Symmetry:

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

Transpose symmetry:

$f(n) = \mathcal{O}(g(n)$ iff $g(n) = \Omega(f(n))$

$f(n) = o(g(n)$ iff $g(n) = \omega(f(n))$

This implies an analogy between aymptotic comparison of functions f and g and comparison of real numbers a and b:

$$
\begin{aligned}
f(n) = O(g(n)) &\quad\approx\quad a \leq b \\
f(n) = \Omega(g(n)) &\quad\approx\quad a \geq b \\
f(n) = \Theta(g(n)) &\quad\approx\quad a = b \\
f(n) = o(g(n)) &\quad\approx\quad a < b \\
f(n) = \omega(g(n)) &\quad\approx\quad a > b
\end{aligned}
$$

Assignment 1 (continued)

Assignment Problem 1.4. (deadline: May 13, 5:25pm)
Prove that

(a) $\mathcal{O}(g(n)) \cap \Omega(g(n)) = \Theta(g(n))$;

(b) $o(g(n)) \cap \omega(g(n))$ is the empty set.

Assignment Problem 1.5. (deadline: May 13, 5:25pm)
Prove that $n! \in \omega(2^n)$ and $n! \in o(n^n)$.