Growth Functions

Chapter 3

Reading Assignment

- p.43-49
- p. 50-51: know the definitions of o-notation and ω-notation, that's all.
- p.53-59: know what monotonicity is (p. 53), equation (3.3), (3.8) & (3.9) (p. 54), properties of exponentials (middle of p. 55), and properties of logarithms (second half of p.56). <u>These will not be discussed in class.</u>

Asymptotic notations of complexity orders

$$T(n) = \Theta(g(n))$$

$$\Theta(g(n)) = \{T(n) \mid \exists c_1, c_2, n_0 \text{ s.t. } 0 \le c_1 g(n) \le T(n) \le c_2 g(n) \text{ for all } n \ge n_0\}$$

Note: there exists three constants

Example:

$$\frac{n^2}{14} \le \frac{n^2}{2} - 3n \le \frac{n^2}{2} \; if \;\; n>7.$$

$$6n^3 \ne \Theta(n^2)$$

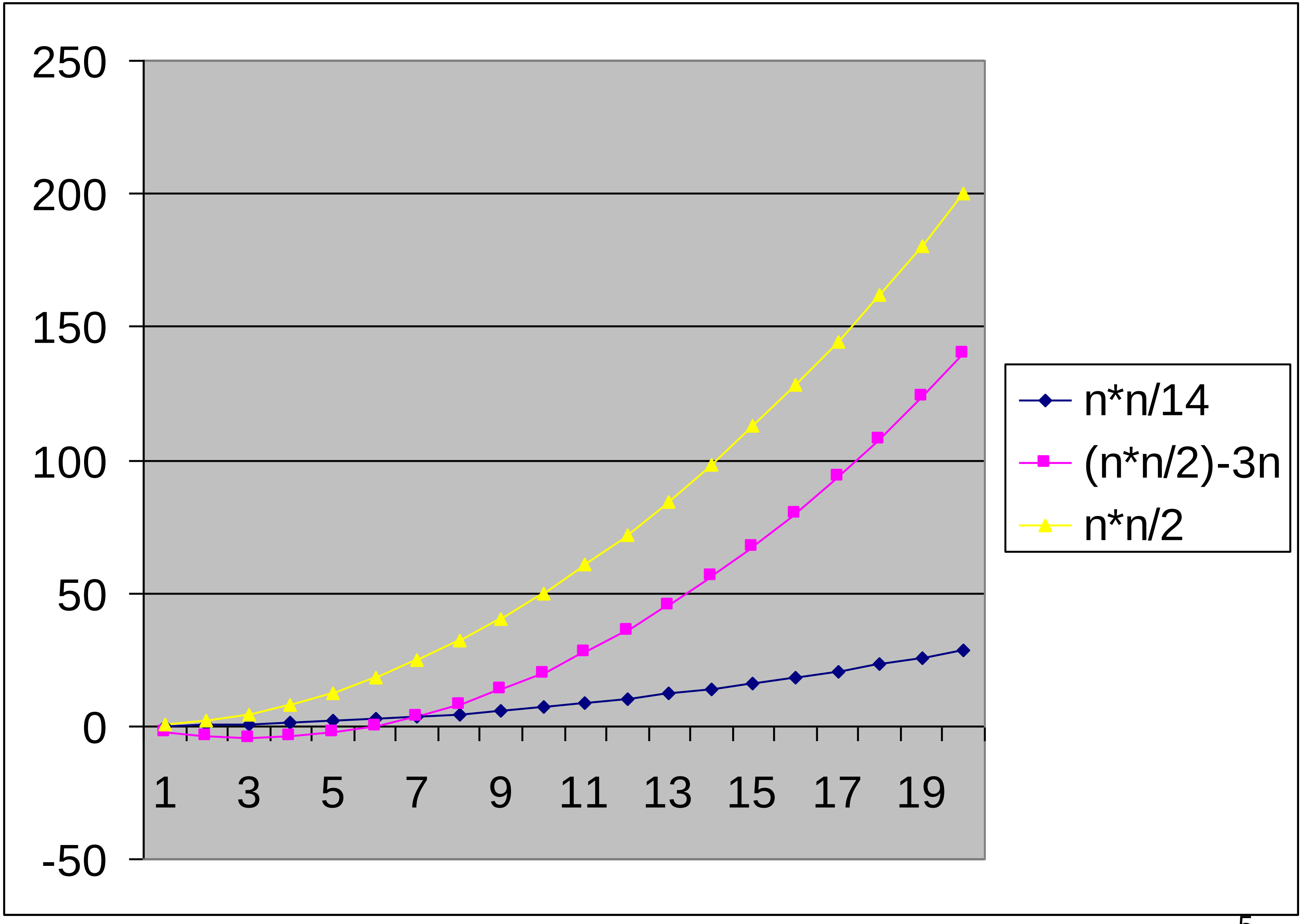
$$f(n) = an^2 + bn + c, \;\; a, b, c \text{ constants, } \; a>0.$$

$$\Rightarrow \; f(n)=\Theta(n^2).$$

- In general,

$$p(n) = \sum_{i=0}^{d} a_i n^i \text{ where } a_i \text{ are constant with } a_d > 0.$$

$$\text{Then } P(n) = \Theta(n^d).$$


250
200
150
100
50
0
-50
1
3
5
7
9
11
13
15
17
19
n*n/14
(n*n/2)-3n
n*n/2

upper bound

$$T(n) = O(g(n))$$

$$O(g(n)) = \{T(n) \mid \exists c, n_0 \text{ s.t. } 0 \le T(n) \le cg(n)\ \forall n \ge n_0\}$$

Note: there exists two constants

strict upper bound

$$T(n) = o(g(n))$$

$$o(g(n)) = \{T(n) \mid \forall c, \exists n_0 \forall n \geq n_0, 0 \leq T(n) < cg(n)\}$$

Note: for all c, there exists n_0

lower bound

$$T(n) = \Omega(g(n))$$

$$\Omega(g(n)) = \{T(n) \mid \exists c, n_0 \text{ s.t. } 0 \le cg(n) \le T(n)\ \forall n \ge n_0\}$$

Note: there exists two constants

strict lower bound

$$T(n) = \omega(g(n))$$

$$\omega(g(n)) = \{T(n) \mid \forall c, \exists n_0 \forall n \geq n_0, 0 \leq cg(n) < T(n)\}$$

Note: for all c, there exists n_0

Theorem 3.1.

- For any two functions *f*(*n*) and *g*(*n*), $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

math you should know

- Monotonicity:
  - A function f is *monotonically increasing* if $m \leq n$ implies $f(m) \leq f(n)$.
  - A function f is *monotonically decreasing* if $m \leq n$ implies $f(m) \geq f(n)$.
  - A function f is *strictly increasing* if $m < n$ implies $f(m) < f(n)$.
  - A function f is *strictly decreasing* if $m > n$ implies $f(m) > f(n)$.

floor and ceiling

$$x - 1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x + 1$$

$$\lceil n / 2 \rceil + \lfloor n / 2 \rfloor = n$$

modular arithmetic

– For any integer *a* and any positive integer *n*, the value *a* mod *n* is the **remainder** (or **residue**) of the quotient *a/n* :

$$a \bmod n = a - \lfloor a/n \rfloor n$$

$$0 \le a \bmod n \le n$$

exponentials

$a^0=1$

$a^1=a$

$a^{-1}=1/a$

$a^m.a^n=a^{m+n}$

$(a^m)^n=(a^n)^m=a^{mn}$

logarithms

$\lg n = \log_2 n$

$\ln n = \log_e n$ where $e = 2.72$

$\lg^k n = (\lg n)^k$

$\lg\lg n = \lg(\lg n)$

logarithms

for all real a,b,c > 0 and n,

$$a=b^{\log_b a}$$

$$\log_c(ab)=\log_c a+\log_c b$$

$$\log_b a^n=n\log_b a$$

$$\log_b a=\log_c a/\log_c b$$

$$\log_b(1/a)=-\log_b a$$

$$\log_b a=1/\log_a b$$

$$a^{\log_b c}=c^{\log_b a}$$

polynomial

a polynomial in n of degree d has the form

$$\sum_{i=0}^{i=d} a_i n^i$$

Algorithm Complexity

- If $T(n)=O(\lg^k n)$ for an algorithm, it is a polylogarithmic algorithm.

Algorithm Complexity

- If $T(n)=O(n^k)$ for an algorithm, $k \geq 1$, it is a polynomial algorithm.
- $k=1$ is a special case: linear algorithm
- $k=2$ is a special case: quadratic algorithm

Polylogarithmic vs. polynomial algorithms

- $\log^a n = o(n^b)$ for any constants *a,b > 0*.

- i.e., any positive polynomial function of n grows faster than any polylogarithmic function of n as n increases.

- So for large inputs, polylogarithmic algorithms will be more efficient than polynomial algorithms.

Exponentials

- Exponential functions: a function with a base greater than 1 (e.g. c^n where $c>1$)

- If $T(n)=O(c^n$ where $c>1)$ for an algorithm, it is an exponential algorithm.

Polynomial algorithms v.s. Exponential algorithms

- Any exponential function with a base greater than 1 (e.g. c^n where $c>1$) grows faster than any polynomial function n^b, where b and c are constants.

$$n^b = o(2^n)$$

- So for large inputs, polynomial algorithms will be more efficient than exponential algorithms.

Factorials

- Factorial function: a function of the form n!
- If $T(n)=O(n!)$ for an algorithm, it is an algorithm of factorial complexity.

Exponential algorithms v.s. Factorial algorithms

- $$2^n = o(n!)$$

- So for large inputs, exponential algorithms with a base of 2 will be more efficient than factorial algorithms.

$$n! = o(n^n)$$

- The function n^n grows even more quickly than the factorial function. Therefore factorial algorithms will be more efficient than any algorithm with complexity order $O(n^n)$.