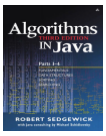
Week 3: references and exercises

- Efficiency of an algorithm: best case, worst case, average case
- Asymptotic expansion, O-Theta, O-O
- Complexity classes: constant, logarithmic, linear, linearithmic, quadratic, exponential
- Empirical performance tests
- Analysis of iterative and recursive algorithms
- Proof by induction

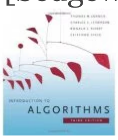
References

(in non-exclusive disjunction)

- 

[Sedgewick & Wayne 2011] §1.4
- 

[Sedgewick 2003] Chapter 2
- 



[Cormen, Leiserson, Rivest & Stein 2009] Chapter 3
- Wikipedia: harmonic number , Landau notation (grand-O)

1 Exercises: asymptotic notation

1.1 Growth Comparison

Fill in the following table. For each pair f, g, write "=" if $f = \Theta(g)$, "" "if $f = o(g)$," "" if $g = o(f)$, and "??? If none of the three apply. There is no need to justify your answers. $\lg n$ denotes the binary logarithm of n.

a	$f(n) = 2^{3^n}$	$g(n) = 3^{2^n}$
b	$f(n) = \lg(\sqrt{n})$	$g(n) = \sqrt{\lg n}$
c	$f(n) = \log_3(\log_4 n)$	$g(n) = \log_4(\log_3 n)$
d	$f(n) = 1 + 3n + 3n^2 + n^3$	$g(n) = n^2 \lg n$
e	$f(n) = \sqrt{n}$	$g(n) = \sqrt[3]{n}$
f	$f(n) = \sum_{i=0}^{n} 2015^i$	$g(n) = 2015^n$
g	$f(n) = \sum_{i=1}^{n} n/i$	$g(n) = n \lg n$
h	$f(n) = n^{2015}$	$g(n) = n!$
i	$f(n) = (n+1)!$	$g(n) = n!$
j	$f(n) = 2.015^{n+1}$	$g(n) = 2.015^n$
k	$f(n) = 2.015^{2n}$	$g(n) = 2.015^n$
l	$f(n) = 2.015^{\lg n}$	$g(n) = 2^{\lg n}$
m	$f(n) = \lg(n!)$	$g(n) = n \lg n$
n	$f(n) = \begin{cases} 1 & \{n < 2\} \\ g(\lceil n/2 \rceil) + g(\lfloor n/2 \rfloor) + O(1) & \{n \geq 2\} \end{cases}$	$g(n) = n \lg n$
o	$f(n) = \begin{cases} 1 & \{n = 0\} \\ 2n \cdot g(n-1) & \{n > 0\} \end{cases}$	$g(n) = n!$

1.2 Evidence

- Show that $2015^{O(\log n)} = n^{O(1)}$.
- Show that $2^{O(n)} = (O(1))^n$.

1.3 Sub-exponential growth

In this exercise, we study the notions of " ***sub-exponential*** " and " ***super-polynomial*** ", defined thus for a function $f(n)$on $n = 0, 1, 2, \ldots$.

- **(SE)** f is sub-exponential ssi$f(n) = 2^{o(n)}$
- **(SP)** f is super-polynomial ssi$n^{O(1)} = o(f(n))$

i. Give definitions equivalent to (SE) and (SP), without asymptotic notation (O, o, ...)
Hint : Discover the relation between f and the functions hidden by the asymptotic terms and express the imposed constraint either as an inequality that is worth almost everywhere, or as a limit $\lim_{n\to\infty}$.

ii. Show that the function $g(n) = (n+1)^{n/\log_{2015}(n+1)}$is super-polynomial but not sub-exponential.

iii. Is there a function that is super-polynomial and sub-exponential at the same time? (Give an example, or show that it's impossible.)

1.4 Harmonic number

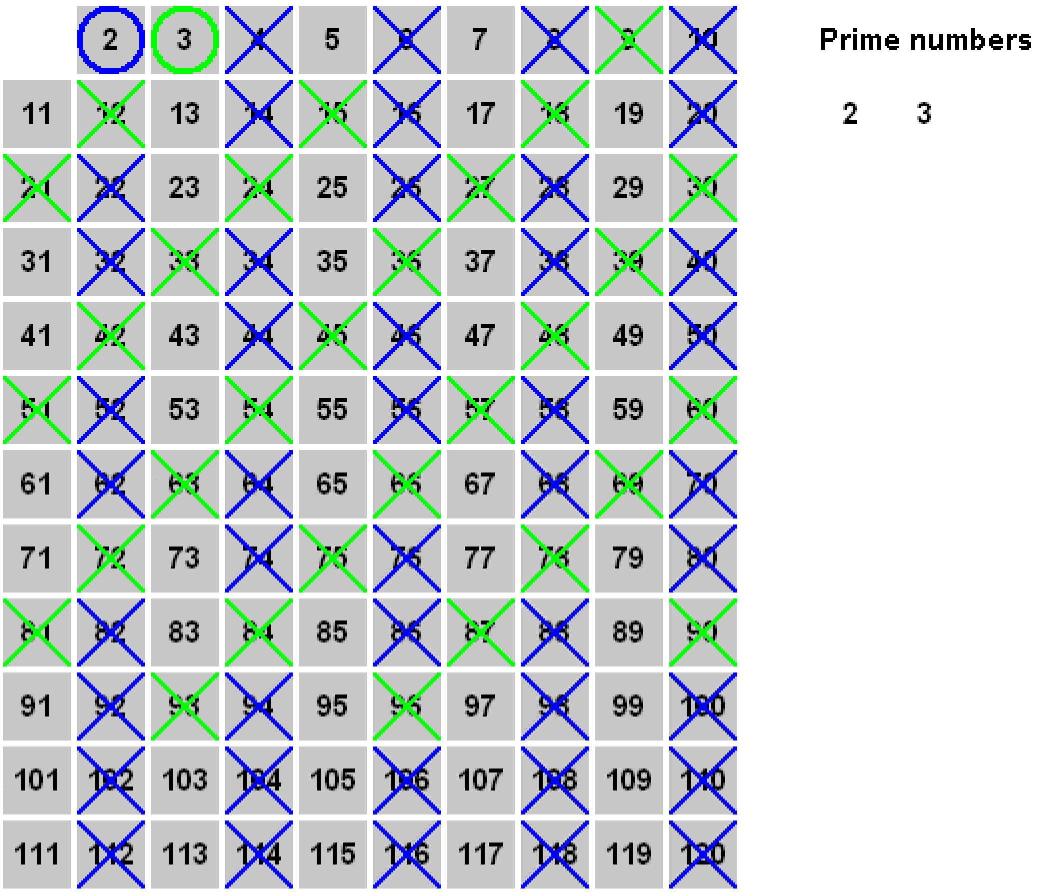
This exercise shows that the harmonic number is at least logarithmic.

Give a recursive definition of the harmonic number $H(n) = 1 + 1/2 + 1/3 + \cdots + 1/n$. Demonstrate by induction that $H(2^n) \leq 1 + n/2$. Motrez as result $H(n) = \Omega(\log n)$.
Hint: Use the terminal $x_1 + x_2 + \cdots + x_N \geq N \cdot \min\{x_1, \ldots, x_N\}$in the inductive case.

2. Analysis of algorithms

2.1 Eratosthenes screening

The Eratosthenes algorithm finds the prime numbers less than or equal to `n` by elimination. It is a question of marking all the multiples 2 `k` , 3 `k`, ... ≤ `n` for all integer values `k` = 2,3,4, ..., `n` in ascending order. If on arriving at `k` , it is unmarked, then it is a prime number.



i. Give the algorithm `sieve( n )` in pseudocode or Java to perform the calculation of the screen up to a `n` > 1. Use a Boolean array to mark integers.

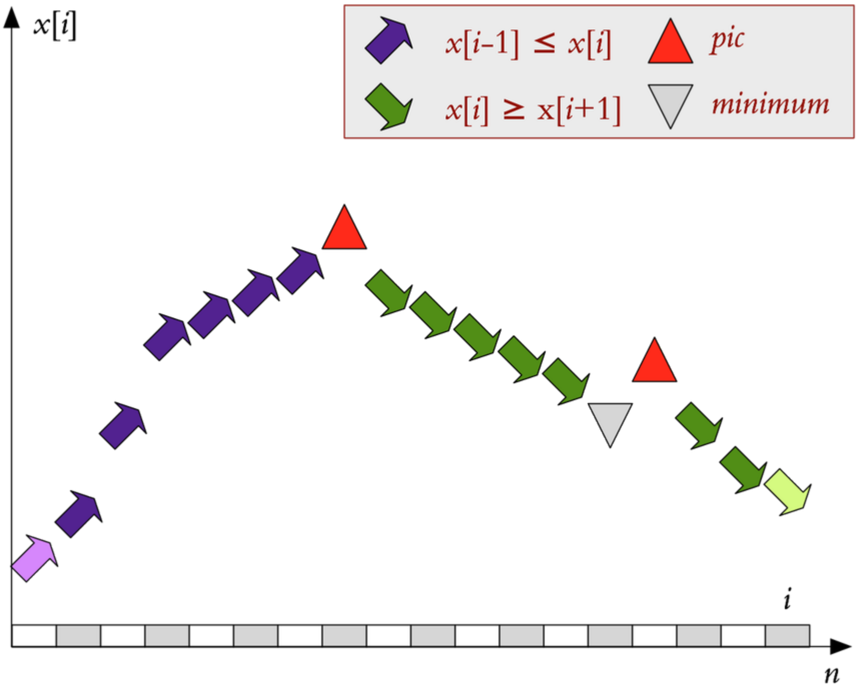
ii. Analyze the calculation time of `sieve`.

iii. Theorem of Prime Numbers (NPT) characterizes the growth in the number $\pi(n)$of primes less than or equal to `n` : $\pi(n) \sim \frac{n}{\ln n}$. Characterize the asymptotic growth of the damped time (by prime number) of `sieve`.

iv. Prove that it suffices to mark the multiples of `k` while $k^2 \leq n$. Analyze how the acceleration involved (that one executes the loop for $k = 2, 3, \ldots, \lfloor\sqrt{n}\rfloor$) changes the computation time.

v. Empirically check the result of **iii.** or **iv.**

2.2 Finding the Peak



The *peak* in an array `x[0..n-1]` is an element `x[i-1] ≤ x[i] ≥ x[i+1]`. Give an algorithm to find the peak. Analyze its calculation time.

Hint : Divide-to-reign with nesting (*bracketing*).