Algoritmos de Engenharia

Fundamentos (Caps. 1, 2 e 3)

Exercícios

1. (Exercício 2.1-3 do livro do Cormen) Escreva um pseudocódigo para a busca linear e mostre, usando invariância de laço, que o seu algoritmo está correto.
2. Implemente o algoritmo de ordenação por inserção e crie uma cópia anotada dele que mede o número de operações no modelo da *Random Access Machine* (RAM, seção 2.2 livro do Cormen). Usando entradas de tamanho crescente, mostre em um gráfico quando o tempo de execução no modelo RAM diverge de medições feitas em uma máquina real.
3. Mostre numericamente com suas implementações dos algoritmos de insertion-sort e merge-sort como se comporta o desempenho de cada algoritmo utilizando entradas de tamanho crescente, considerando entradas de pior caso, melhor caso e caso médio. Analise, para cada tipo de entrada, se existe algum ponto a partir do qual um algoritmo passa a ser mais rápido que o outro.

Algorithms

- A well-defined computational procedure that takes some value, or set of values, as **input** and produces some value, or set of values, as **output** in a **finite** amount of **time**.
- A sequence of **computational steps** that transform the input into the output.
- A tool for **solving** a well-defined **computational problem**.
- Sorting problem:

  Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

  Output: A permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

  Thus, given the input sequence $\langle 31, 41, 59, 26, 41, 58 \rangle$, a correct sorting algorithm returns as output the sequence $\langle 26, 31, 41, 41, 58, 59 \rangle$.

Algorithms

- Data structures—ways to store and organize data in order to facilitate access and modifications.
  - e.g., arrays, queues, stacks, lists, hash tables, heaps.
- Techniques
  - e.g., greedy, divide-and-conquer, dynamic programming.
- Efficiency
  - Speed: how long does an algorithm take to produce its result?
  - Example: *insertion sort*, takes time roughly equal to $c_1 n^2$ to sort n items, while *merge sort*, takes time roughly equal to $c_2 n \lg n$ (usually, $c_1 < c_2$).
  - What other characteristics can be considered to measure the efficiency of an algorithm?

Algorithms

Example: sort an array of 10 million numbers

- Computer A:
  - Runs insertion sort;
  - Executes 10 billion instructions per second;
  - Requires $2n^2$ instructions to sort n numbers;
- Computer B:
  - Runs merge sort;
  - Executes only 10 million instructions per second;
  - Requires $50n \lg n$ instructions to sort n numbers;

Computer A takes

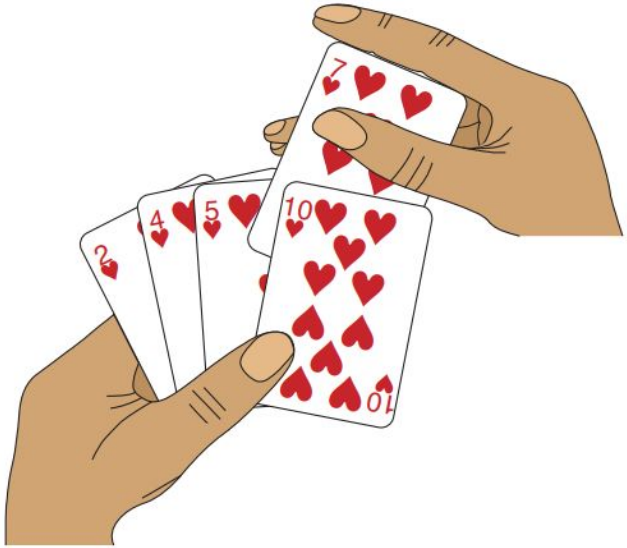
$$\frac{2 \cdot (10^7)^2 \text{ instructions}}{10^{10} \text{ instructions/second}} = 20{,}000 \text{ seconds (more than 5.5 hours)},$$

while computer B takes

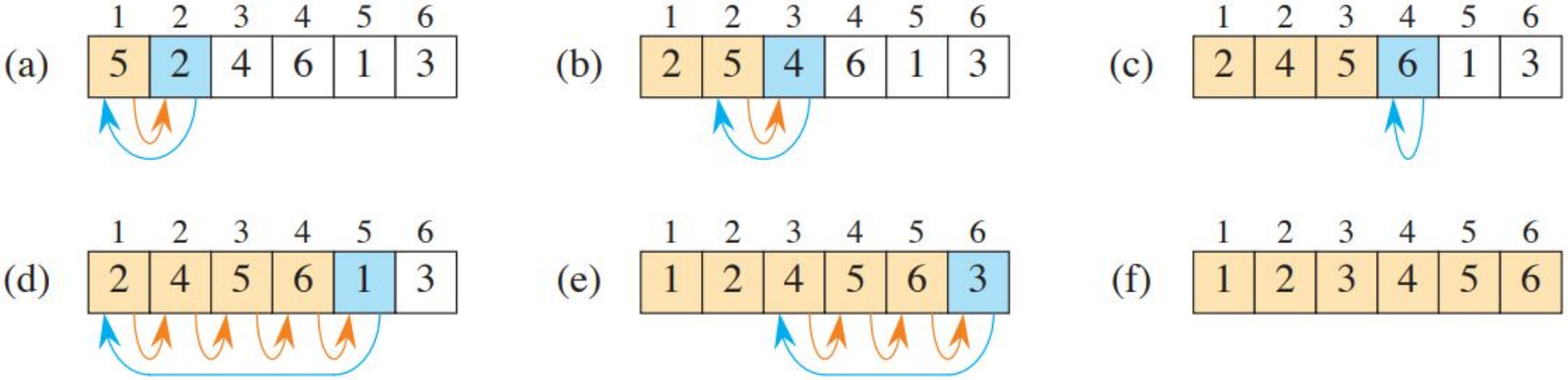
$$\frac{50 \cdot 10^7 \lg 10^7 \text{ instructions}}{10^7 \text{ instructions/second}} \approx 1163 \text{ seconds (under 20 minutes)}.$$

Insertion sort

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```



Insertion sort



How to make sure this algorithm works?

Loop invariants

Loop invariant properties help us understand why an algorithm is *correct*. They are properties that hold for:

- **Initialization:** it is true prior to the first iteration of the loop.
- **Maintenance:** if it is true before an iteration of the loop, it remains true before the next iteration.
- **Termination:** the loop terminates, and when it terminates, the invariant—usually along with the reason that the loop terminated—gives us a useful property that helps show that the algorithm is correct.

Resembles proof by induction.

Loop invariants – correctness of insertion sort

"The subarray $A[1 : i-1]$ consists of the elements *originally* in $A[1 : i-1]$, but in *sorted order*."

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- **Initialization:**

  The subarray $A[1 : i-1]$ consists of just the **single element** $A[1]$, which is in fact the *original* element in $A[1]$. Moreover, *this subarray is sorted*, which shows that the loop invariant holds prior to the first iteration of the loop.

Loop invariants – correctness of insertion sort

"The subarray $A[1 : i-1]$ consists of the elements *originally* in $A[1 : i-1]$, but in *sorted order*."

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- **Maintenance:**

  Informally, the body of the **for** loop works by **moving** the values in $A[i-1]$, $A[i-2]$, $A[i-3]$, and so on **until it finds the proper position** for $A[i]$, at which point it inserts the value of $A[i]$ (line 8). The subarray $A[1 : i]$ then consists of the elements originally in $A[1 : i]$, but in sorted order. Incrementing i for the next iteration of the **for** loop then preserves the loop invariant.

Loop invariants – correctness of insertion sort

"The subarray $A[1 : i-1]$ consists of the elements *originally* in $A[1 : i-1]$, but in *sorted order*."

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- **Termination:**

  The loop **terminates** once i equals $n + 1$. Substituting $n + 1$ for i in the wording of the loop invariant yields that the subarray $A[1 : n]$ consists of the elements *originally* in $A[1 : n]$, but in *sorted order*. Hence, the algorithm is correct.

Analyzing algorithms

Predicting resources that the algorithm requires

- Memory
- Cache hit/miss
- Disk usage
- Communication bandwidth
- Energy consumption
- Computational time

Analyzing algorithms

Random-Access Machine model

- Measuring the number of operations.
- Instructions execute one after another, with no concurrent operations.
- Instruction takes the **same amount of time** as any other instruction and that each data access takes the same amount of time as any other data access. In other words, in the RAM model each instruction or data access takes a constant amount of time—even indexing into an array.
- Instructions commonly found in real computers:
  - Arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling);
  - Data movement (load, store, copy);
  - Control (conditional and unconditional branch, subroutine call and return).

Analyzing algorithms

Random-Access Machine model

```
INSERTION-SORT(A, n)
1  for i = 2 to n                                              3 op.
2      key = A[i]                                              2 op.
3      // Insert A[i] into the sorted subarray A[1 : i − 1].   0 op.
4      j = i − 1                                               2 op.
5      while j > 0 and A[j] > key                              5 op.
6          A[j + 1] = A[j]                                     4 op.
7          j = j − 1                                           2 op.
8      A[j + 1] = key                                          4 op.
```

Analyzing algorithms

Inferring the run time of an algorithm

INSERTION-SORT(A, n)		*cost*	*times*
1	**for** $i = 2$ **to** n	c_1	n
2	$key = A[i]$	c_2	$n-1$
3	// Insert $A[i]$ into the sorted subarray $A[1:i-1]$.	0	$n-1$
4	$j = i-1$	c_4	$n-1$
5	**while** $j > 0$ and $A[j] > key$	c_5	$\sum_{i=2}^{n} t_i$
6	$A[j+1] = A[j]$	c_6	$\sum_{i=2}^{n} (t_i - 1)$
7	$j = j-1$	c_7	$\sum_{i=2}^{n} (t_i - 1)$
8	$A[j+1] = key$	c_8	$n-1$

Worst case?
Best case?
Average case?

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1) .$$

Designing algorithms

- Insertion-sort: *incremental* algorithm
  - For each element $A[i]$, insert it into its proper place in the subarray $A[1 : i]$, having already sorted the subarray $A[1 : i-1]$.
- Divide-and-conquer algorithms
  - Recursively solve subproblems. If the problem is small enough—the *base case*—you just solve it directly without recursing.

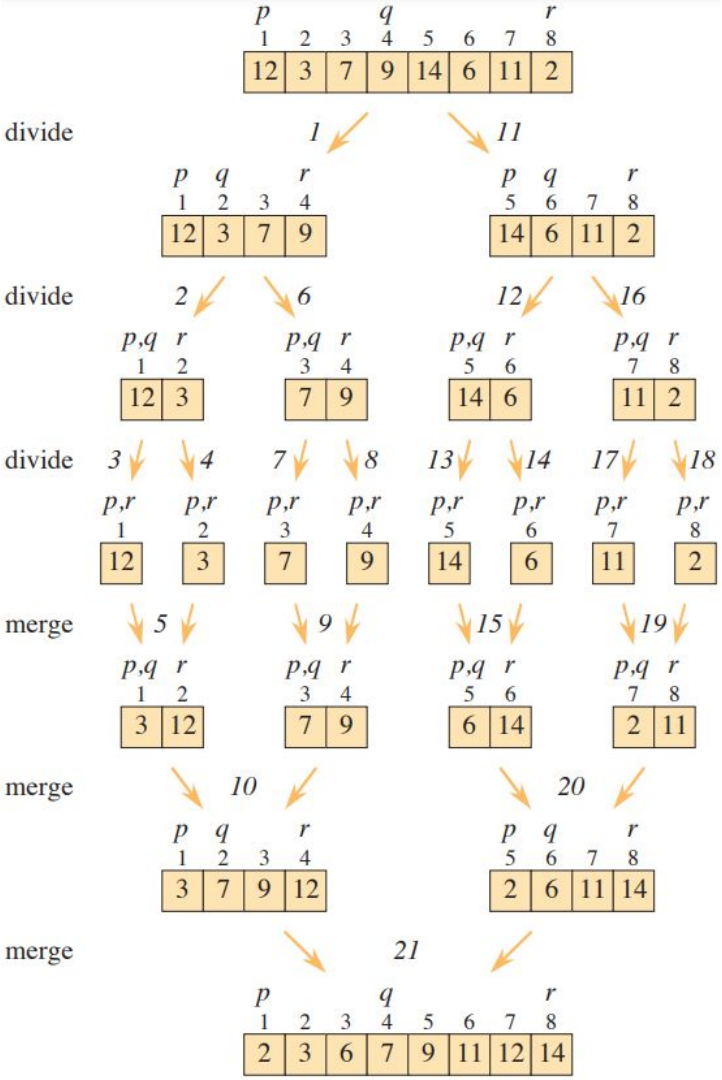
The divide-and-conquer method

Divide the problem into one or more subproblems that are smaller instances of the same problem.

Conquer the subproblems by solving them recursively.

Combine the subproblem solutions to form a solution to the original problem.

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```



What is the time complexity?

```
MERGE(A, p, q, r)
 1  n_L = q − p + 1          // length of A[p : q]
 2  n_R = r − q              // length of A[q + 1 : r]
 3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
 4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
 5      L[i] = A[p + i]
 6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
 7      R[j] = A[q + j + 1]
 8  i = 0                    // i indexes the smallest remaining element in L
 9  j = 0                    // j indexes the smallest remaining element in R
10  k = p                    // k indexes the location in A to fill
11  // As long as each of the arrays L and R contains an unmerged element,
    //      copy the smallest unmerged element back into A[p : r].
12  while i < n_L and j < n_R
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
18      k = k + 1
19  // Having gone through one of L and R entirely, copy the
    //      remainder of the other to the end of A[p : r].
20  while i < n_L
21      A[k] = L[i]
22      i = i + 1
23      k = k + 1
24  while j < n_R
25      A[k] = R[j]
26      j = j + 1
27      k = k + 1
```

Characterizing Running Times

Asymptotic notations

1. A way to describe **behavior of functions** in the limit.
2. Describe **growth** of functions.
3. **Focus** on what's important by **abstracting** away **low-order terms** and **constant factors**.
4. How we indicate **running times** of algorithms.
5. A way to compare "**sizes**" of functions:

$$
\begin{array}{lll}
O & \approx & \leq \\
\Omega & \approx & \geq \\
\Theta & \approx & = \\
o & \approx & < \\
\omega & \approx & >
\end{array}
$$

O-notation

O-notation characterizes an ***upper bound*** on the asympototic behavior of a function: it says that a function grows ***no faster*** than a certain rate. This rate is based on the highest order term.

For example:

$f(n) = 7n^3 + 100n^2 - 20n + 6$ is $O(n^3)$, since the highest order term is $7n^3$, and therefore the function grows no faster than n^3.

The function $f(n)$ is also $O(n^5)$, $O(n^6)$, and $O(n^c)$ for any constant $c \geq 3$.

Ω-notation

Ω-notation characterizes a ***lower bound*** on the asymptotic behavior of a function.

For example:

$f(n) = 7n^3 + 100n^2 - 20n + 6$ is $\Omega(n^3)$, since the highest-order term, n^3, grows at least as fast as n^3.

The function $f(n)$ is also $\Omega(n^2)$, $\Omega(n)$ and $\Omega(nc)$ for any constant $c \leq 3$.

Θ-notation

Θ-notation characterizes a ***tight bound*** on the asympototic behavior of a function: it says that a function grows ***precisely*** at a certain rate, again based on the highest-order term.

If a function is is both $O(f(n))$ and $\Omega(f(n))$, then a function is $\Theta(f(n))$.

Example: Insertion sort

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

Example: Insertion sort

First, show that INSERTION-SORT is runs in $O(n^2)$ time, regardless of the input:

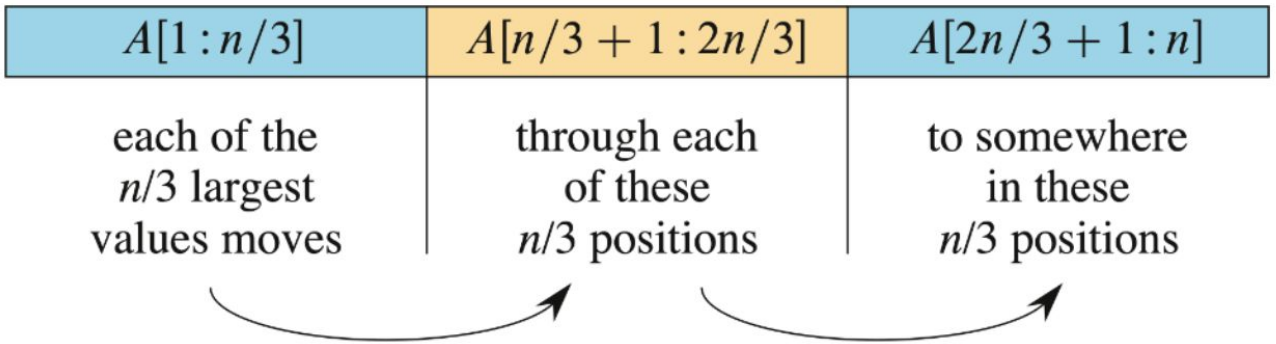
- The outer **for** loop runs $n - 1$ times regardless of the values being sorted.
- The inner **while** loop iterates at most $i - 1$ times.
- The exact number of iterations the **while** loop makes depends on the values it iterates over, but it will definitely iterate between 0 and $i - 1$ times.
- Since i is at most n, the total number of iterations of the inner loop is at most $(n - 1)(n - 1)$, which is less than n^2.

Each inner loop iteration takes constant time, for a total of at most cn^2 for some constant c, or $O(n^2)$.

Example: Insertion sort

Now show that INSERTION-SORT has a worst-case running time of $\Omega(n^2)$:

- Observe that for a value to end up k positions to the right of where it started, the line $A[j+1] = A[j]$ must have been executed k times.
- Assume that n is a multiple of 3 so that we can divide the array A into groups of $n/3$ positions.



Example: Insertion sort

Because at least $n/3$ values must pass through at least $n/3$ positions, the line $A[j+1] = A[j]$ executes at least $(n/3)(n/3) = n^2/9$ times, which is $\Omega(n^2)$. For this input, INSERTION-SORT takes time $\Omega(n^2)$.

Since we have shown that INSERTION-SORT runs in $O(n^2)$ time in all cases and that there is an input that makes it take $\Omega(n^2)$ time, we can conclude that the worst-case running time of INSERTION-SORT is $\Theta(n^2)$.

The constant factors for the upper and lower bounds may differ. That doesn't matter.

~~Insertion-sort's running time is $\Theta(n^2)$~~

Insertion-sort's running time is $O(n^2)$ 👍

Insertion-sort's running time is $\boldsymbol{\Omega}(n)$ 👍

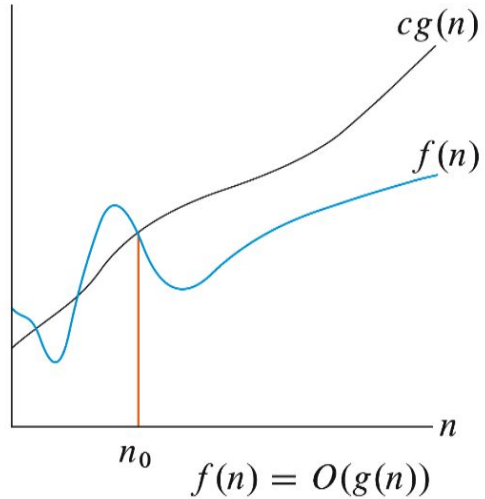
Example: Merge-sort

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

Merge-sort's running time is $O(n \lg n)$ 👍
Merge-sort's running time is $\Omega(n \lg n)$ 👍
Merge-sort's running time is $\Theta(n \lg n)$ 👍

O-notation

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$

Example

$2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

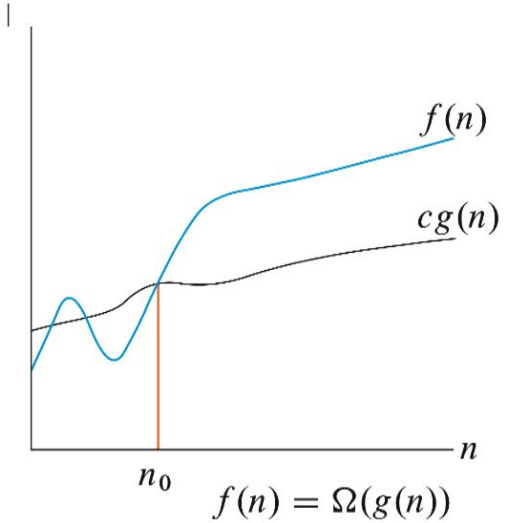
n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$

Also,

n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Ω-notation

$\Omega(g(n)) = \{f(n) :$ there exist positive constants c and n_0 such that
$0 \le cg(n) \le f(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Example

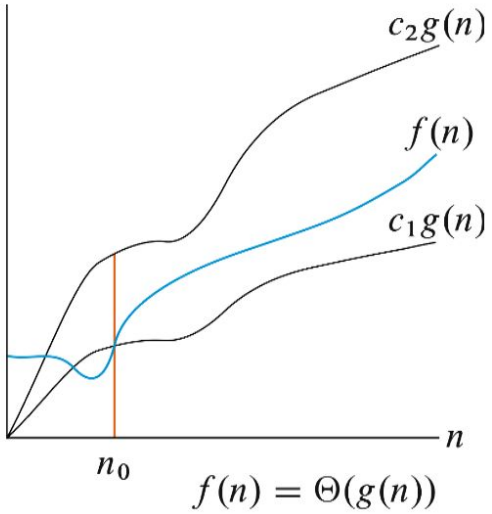
$\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\} .$$



Example

$n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Theorem

$f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$.

Leading constants and low-order terms don't matter.

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

Exercícios

1. (Exercício 2.1-3 do livro do Cormen) Escreva um pseudocódigo para a busca linear e mostre, usando invariância de laço, que o seu algoritmo está correto.
2. Implemente o algoritmo de ordenação por inserção e crie uma cópia anotada dele que mede o número de operações no modelo da *Random Access Machine* (RAM, seção 2.2 livro do Cormen). Usando entradas de tamanho crescente, mostre em um gráfico quando o tempo de execução no modelo RAM diverge de medições feitas em uma máquina real.
3. Mostre numericamente com suas implementações dos algoritmos de insertion-sort e merge-sort como se comporta o desempenho de cada algoritmo utilizando entradas de tamanho crescente, considerando entradas de pior caso, melhor caso e caso médio. Analise, para cada tipo de entrada, se existe algum ponto a partir do qual um algoritmo passa a ser mais rápido que o outro.