Algorithms for Data Science

CSOR W4246

Eleni Drinea

Computer Science Department

Columbia University

Tuesday, September 8, 2015

Outline

Today

1. Overview
2. A first algorithm: insertion sort
3. Analysis of algorithms
4. Efficient algorithms
5. Asymptotic notation

Algorithms

- An algorithm is a well-defined computational procedure that transforms the input (a set of values) into the output (a new set of values).
- The desired input/output relationship is specified by the statement of the **computational problem** for which the algorithm is designed.
- An algorithm is correct if, for every input, it **halts** with the correct output.

Efficient Algorithms

- In this course we are interested in algorithms that are correct and efficient.
- Efficiency is related to the resources an algorithm uses: time, space
  - *How much time/space are used?*
  - *How do they scale as the input size grows?*

We will primarily focus on efficiency in **running time**.

Running time

Running time = number of primitive computational steps performed; typically these are

1. arithmetic operations: add, subtract, multiply, divide fixed-size integers
2. data movement operations: load, store, copy
3. control operations: branching, subroutine call and return

We will use pseudocode for our algorithm descriptions.

Today

Sorting

- **Input:** A list A of n integers $x_1, \ldots, x_n$.
- **Output:** A permutation $x'_1, x'_2, \ldots, x'_n$ of the n integers where they are sorted in non-decreasing order, i.e., $x'_1 \leq x'_2 \leq \ldots \leq x'_n$

Sorting

- **Input:** A list A of n integers $x_1, \ldots, x_n$.
- **Output:** A permutation $x'_1, x'_2, \ldots, x'_n$ of the n integers where they are sorted in non-decreasing order, i.e., $x'_1 \leq x'_2 \leq \ldots \leq x'_n$

Example

- Input: $n = 6$, $A = \{9, 3, 2, 6, 8, 5\}$
- Output: $A = \{2, 3, 5, 6, 8, 9\}$

What data structure should we use to represent the list?

Sorting

- **Input:** A list A of n integers $x_1, \ldots, x_n$.
- **Output:** A permutation $x'_1, x'_2, \ldots, x'_n$ of the n integers where they are sorted in non-decreasing order, i.e., $x'_1 \leq x'_2 \leq \ldots \leq x'_n$

Example

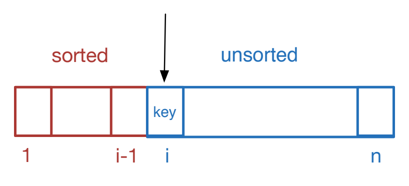
- Input: $n = 6$, $A = \{9, 3, 2, 6, 8, 5\}$
- Output: $A = \{2, 3, 5, 6, 8, 9\}$

What data structure should we use to represent the list?

Array: collection of items of the same data type

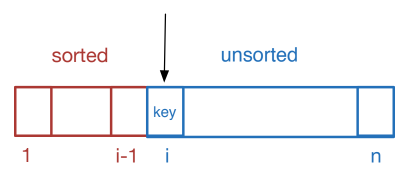
- allows for *random access*
- "zero" indexed in C++ and Java

Insertion sort in English



1. Start with a (trivially) sorted subarray of size 1 consisting of the first element of A.

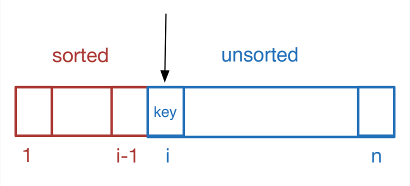
Insertion sort in English



1. Start with a (trivially) sorted subarray of size 1 consisting of the first element of A.

2. **Increase** the size of the sorted subarray by 1 by **inserting** the next element of A into its **correct** location.
   - Compare that next element, call it `key`, with every element x of the sorted subarray starting from the **right**.
     - If $x >$ `key`, move x one position to the right.
     - Else ($x \leq$ `key`), **insert** `key` after x.

Insertion sort in English

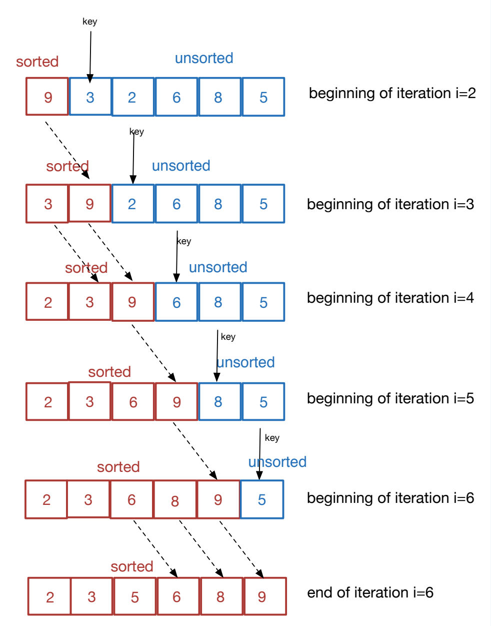


1. Start with a (trivially) sorted subarray of size 1 consisting of the first element of A.

2. **Increase** the size of the sorted subarray by 1 by **inserting** the next element of A into its **correct** location.
   - Compare that next element, call it `key`, with every element x of the sorted subarray starting from the **right**.
     - If $x >$ `key`, move x one position to the right.
     - Else ($x \leq$ `key`), **insert** `key` after x.

3. Repeat Step 2. until the sorted subarray has size n.

Example of insertion sort: $n = 6, A = \{9, 3, 2, 6, 8, 5\}$



Pseudocode

Let A be an array of n integers.

`insertion-sort`(A)

for $i = 2$ to n **do**

  `key` $= A[i]$

  //Insert $A[i]$ into the sorted subarray $A[1, i-1]$

  $j = i - 1$

  while $j > 0$ and $A[j] >$ `key` **do**

    $A[j+1] = A[j]$

    $j = j - 1$

  end while

  $A[j+1] =$ `key`

end for

Today

Analysis of algorithms

- **Correctness**
- **Running time**
- **(Space)**

Analysis of algorithms

- **Correctness:** formal proof often by induction
- **Running time:** number of primitive computational steps
  - Not the same as **time** it takes to execute the algorithm.
  - We want a measure that is independent of hardware.
  - We want to know how running time scales with the size of the input.
- **Space:** how much space is required by the algorithm

Analysis of `insertion-sort`

Notation: $A[i, j]$ is the subarray of A that starts at position i and ends at position j.

- **Correctness:** follows from the key observation that after loop i, the subarray $A[1, i]$ is sorted
- **Running time:** number of primitive computational steps
- **Space:** in place algorithm (at most a constant number of elements of A are stored outside A at any time)

Example of induction

Fact 1.

For all $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Example of induction

Fact 1.

For all $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Proof.

- **Base case:** $n = 1$
- **Inductive hypothesis:** Assume that the statement is true for $n \geq 1$, that is, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.
- **Inductive step:** We show that the statement is true for $n + 1$. That is, $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$. (Show this!)
- **Conclusion:** It follows that the statement is true for all n since we can apply the inductive step for $n = 2, 3, \ldots$.

$\square$

Correctness of `insertion-sort`

Notation: $A[i, j]$ is the subarray of A that starts at position i and ends at position j.

Minor change in the pseudocode: in line 1, start from $i = 1$ rather than $i = 2$. *How does this change affect the algorithm?*

Claim 1.

Let $n \geq 1$ be a positive integer. For all $1 \leq i \leq n$, after the i-th loop, the subarray $A[1, i]$ is sorted.

Correctness of `insertion-sort` follows if we show Claim 1 (*why?*).

Proof of Claim 1

By induction on i.

- **Base case:** $i = 1$, trivial.
- **Induction hypothesis:** assume that the statement is true for some $1 \leq i < n$.
- **Inductive step:** Show it true for $i + 1$.

  In loop $i + 1$, element `key` $= A[i + 1]$ is inserted into $A[1, i]$, which is sorted (by the induction hypothesis). Since `key` is inserted after the first element $A[\ell]$ for $1 \leq \ell \leq i$ such that `key` $\geq A[\ell]$, and all elements in $A[\ell + 1, j]$ are pushed one position to the right with their order preserved, the statement is true for $i + 1$.

Running time $T(n)$ of `insertion-sort`

for $i = 2$ to n **do**
    `key` $= A[i]$
    //Insert $A[i]$ into the sorted subarray $A[1, i-1]$
    $j = i - 1$
    while $j > 0$ and $A[j] >$ `key` **do**
        $A[j+1] = A[j]$
        $j = j - 1$
    end while
    $A[j+1] =$ `key`
end for

- *How many primitive computational steps are executed by the algorithm?*
- *Equivalently, what is the running time $T(n)$? Bounds on $T(n)$?*

Running time $T(n)$ of `insertion-sort`

for $i = 2$ to n **do** — line 1

  `key` $= A[i]$ — line 2

  //Insert $A[i]$ into the sorted subarray $A[1, i-1]$

  $j = i - 1$ — line 3

  while $j > 0$ and $A[j] >$ `key` **do** — line 4

    $A[j+1] = A[j]$ — line 5

    $j = j - 1$ — line 6

  end while

  $A[j+1] =$ `key` — line 7

end for

- For $2 \leq i \leq n$, let $t_i = \#$ times line 4 is executed.

Running time $T(n)$ of `insertion-sort`

```
for i = 2 to n do                                        line 1
    key = A[i]                                           line 2
    //Insert A[i] into the sorted subarray A[1, i-1]
    j = i - 1                                            line 3
    while j > 0 and A[j] > key do                        line 4
        A[j+1] = A[j]                                    line 5
        j = j - 1                                        line 6
    end while
    A[j+1] = key                                         line 7
end for
```

- For $2 \leq i \leq n$, let $t_i = \#$ times line 4 is executed. Then

$$T(n) = n + 3(n-1) + \sum_{i=2}^{n} t_i + 2\sum_{i=2}^{n}(t_i - 1) = 3\sum_{i=2}^{n} t_i + 2n - 1$$

- *Which input yields the smallest* (best-case) *running time?*
- *Which input yields the largest* (worst-case) *running time?*

Running time $T(n)$ of `insertion-sort`

```
for i = 2 to n do                                      line 1
    key = A[i]                                         line 2
    //Insert A[i] into the sorted subarray A[1, i − 1]
    j = i − 1                                          line 3
   while j > 0 and A[j] > key do                       line 4
       A[j + 1] = A[j]                                 line 5
       j = j − 1                                       line 6
   end while
    A[j + 1] = key                                     line 7
end for
```

- For $2 \leq i \leq n$, let $t_i = \#$ times line 4 is executed. Then

$$T(n) = 3 \sum_{i=2}^{n} t_i + 2n - 1$$

- Best-case running time: $5n - 4$
- Worst-case running time: $\frac{3n^2}{2} + \frac{7n}{2} - 4$

Worst-case analysis

Definition 2.

Worst-case running time: largest possible running time of the algorithm over all inputs of a given size n.

Why worst-case analysis?

- It gives well-defined computable bounds.
- Average-case analysis can be tricky: how do we generate a "random" instance?

The worst-case running time of `insertion-sort` is quadratic.
Is `insertion-sort` *efficient?*

Today

Efficiency of insertion-sort and the brute force solution

Compare to brute force solution:

- At each step, generate a new permutation of the n integers.
- If sorted, stop and output the permutation.

Efficiency of insertion-sort and the brute force solution

Compare to brute force solution:

- At each step, generate a new permutation of the n integers.
- If sorted, stop and output the permutation.

Worst-case analysis: generate $n!$ permutations. *Is brute force solution efficient?*

Efficiency of insertion-sort and the brute force solution

Compare to brute force solution:

- At each step, generate a new permutation of the n integers.
- If sorted, stop and output the permutation.

Worst-case analysis: generate $n!$ permutations. *Is brute force solution efficient?*

- Efficiency relates to the performance of the algorithm as n grows.
- Stirling's approximation formula: $n! \approx \left(\frac{n}{e}\right)^n$.
  - For $n = 10$, generate $3.67^{10} \geq 2^{10}$ permutations.
  - For $n = 50$, generate $18.3^{50} \geq 2^{200}$ permutations.
  - For $n = 100$, generate $36.7^{100} \geq 2^{700}$ permutations!

$\Rightarrow$ Brute force solution is not efficient.

Efficient algorithms –Attempt 1

Definition 3 (Attempt 1).

An algorithm is efficient if it achieves better worst-case performance than brute-force search.

Efficient algorithms –Attempt 1

Definition 3 (Attempt 1).

An algorithm is efficient if it achieves better worst-case performance than brute-force search.

Caveat: fails to discuss the scaling properties of the algorithm.

- If the input size grows by a constant factor, we would like the running time $T(n)$ of the algorithm to increase by a constant factor as well.

Efficient algorithms –Attempt 1

Definition 3 (Attempt 1).

An algorithm is efficient if it achieves better worst-case performance than brute-force search.

Caveat: fails to discuss the scaling properties of the algorithm.

- If the input size grows by a constant factor, we would like the running time $T(n)$ of the algorithm to increase by a constant factor as well.
- Note that polynomial running times scale well: on input of size n, $T(n)$ is at most $c \cdot n^d$ for $c, d > 0$ constants.
  - the **smaller** the exponent of the polynomial the better

Efficient algorithms

Definition 4.

An algorithm is efficient if it has a polynomial running time.

Caveat

- What about huge constants in front of the leading term or large exponents?

However

- Small degree polynomial running times exist for most problems that can be solved in polynomial time.
- Conversely, problems for which no polynomial-time algorithm is known tend to be very hard in practice.
- So we can distinguish between easy and hard problems.

Remark 1.

Today's big data: even low degree polynomials might be too slow!

Are we done with sorting?

Insertion sort is efficient. *Are we done with sorting?*

Are we done with sorting?

Insertion sort is efficient. *Are we done with sorting?*

1. *Can we do better?*
2. *And what is better?*
   - *E.g., is* $T(n) = n^2 + n - 4$ *worth aiming for?*

Running time in terms of # primitive steps

To discuss this, we need a coarser classification of running times of algorithms; exact characterizations

- are **too detailed**;
- do not reveal similarities between running times in an immediate way as n grows large;
- are often **meaningless**: pseudocode steps will **expand** by a constant factor that depends on the hardware.

Aymptotic analysis

A framework that will allow us to compare the rate of growth of different running times as the input size n grows.

- We will express the running time as a function of the number of primitive steps.
- The number of primitive steps is itself a a function of the size of the input n.

$\Rightarrow$ The running time is a function of the size of the input n.

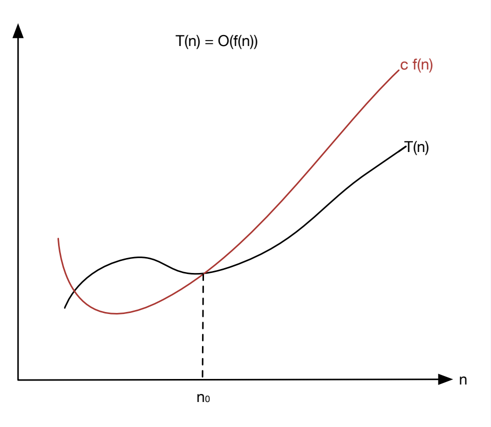
- To compare functions expressing running times, we will ignore their low-order terms and focus solely on the highest-order term.

Today

Asymptotic upper bounds: Big-O notation

Definition 5 (O).

We say that $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.



Asymptotic upper bounds: Big-O notation

Definition 6 (O).

We say that $T(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \leq c \cdot f(n)$.

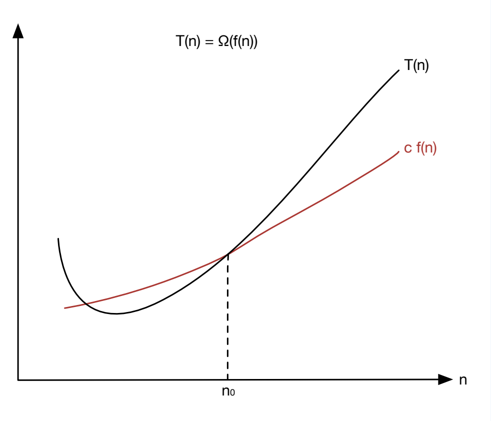
Examples:

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.
- $T(n) = an^2 + b$, $f(n) = n^3$.

Asymptotic lower bounds: Big-Ω notation

Definition 7 (Ω).

We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.



Asymptotic lower bounds: Big-Ω notation

Definition 8 (Ω).

We say that $T(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) \geq c \cdot f(n)$.

Examples:

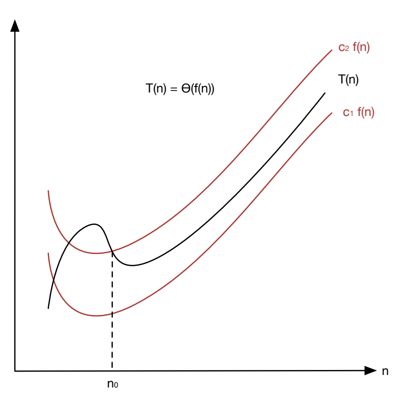
- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.
- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n$.

Asymptotic tight bounds: Θ notation

Definition 9 (Θ).

We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$



Asymptotic tight bounds: Θ notation

Definition 10 (Θ).

We say that $T(n) = \Theta(f(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have

$$c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n).$$

Equivalent definition

$T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

Examples:

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^2$.
- $T(n) = n \log n + n$, and $f(n) = n \log n$.

Asymptotic upper bounds that are **not** tight: little-o

Definition 11 (o).

We say that $T(n) = o(f(n))$ if for any constant $c > 0$ there exists a constant $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have
$T(n) < c \cdot f(n)$.

Asymptotic upper bounds that are **not** tight: little-o

Definition 11 (o).

We say that $T(n) = o(f(n))$ if for any constant $c > 0$ there exists a constant $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.

- Intuitively, $T(n)$ becomes insignificant relative to $f(n)$ as $n \to \infty$.
- Proof by showing that $\lim\limits_{n\to\infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists).

Asymptotic upper bounds that are **not** tight: little-o

Definition 11 (o).

We say that $T(n) = o(f(n))$ if for any constant $c > 0$ there exists a constant $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) < c \cdot f(n)$.

- Intuitively, $T(n)$ becomes insignificant relative to $f(n)$ as $n \to \infty$.
- Proof by showing that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists).

Examples:

- $T(n) = an^2 + b$, $a, b > 0$ constants and $f(n) = n^3$.
- $T(n) = n \log n$, $a, b, d > 0$ constants and $f(n) = n^2$.

Asymptotic lower bounds that are **not** tight: little-ω

Definition 12 (ω).

We say that $T(n) = \omega(f(n))$ if for any constant $c > 0$ there exists $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

Asymptotic lower bounds that are **not** tight: little-ω

Definition 12 (ω).

We say that $T(n) = \omega(f(n))$ if for any constant $c > 0$ there exists $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

- Intuitively $T(n)$ becomes arbitrarily large relative to $f(n)$ as $n \to \infty$.
- $T(n) = \omega(f(n))$ implies that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$ if the limit exists. Then $f(n) = o(T(n))$.

Asymptotic lower bounds that are **not** tight: little-ω

Definition 12 (ω).

We say that $T(n) = \omega(f(n))$ if for any constant $c > 0$ there exists $n_0 \geq 0$ s.t. for all $n \geq n_0$, we have $T(n) > c \cdot f(n)$.

- Intuitively $T(n)$ becomes arbitrarily large relative to $f(n)$ as $n \to \infty$.
- $T(n) = \omega(f(n))$ implies that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$ if the limit exists. Then $f(n) = o(T(n))$.

Examples:

- $T(n) = n^2$ and $f(n) = n \log n$.
- $T(n) = 2^n$ and $f(n) = n^5$.

Basic rules for omitting low order terms from functions

1. ignore **multiplicative** factors: e.g., $10n^3$ becomes n^3
2. n^a dominates n^b if $a > b$: e.g., n^2 dominates n
3. exponentials dominate polynomials: e.g., 2^n dominates n^4
4. polynomials dominate logarithms: e.g., n dominates $\log^3 n$

$\Rightarrow$ for large enough n,

$$\log n < n < n \log n < n^2 < 2^n < n^n$$

Properties of asymptotic growth rates

Transitivity

1. If $f = O(g)$ and $g = O(h)$ then $f = O(h)$.
2. If $f = \Omega(g)$ and $g = \Omega(h)$ then $f = \Omega(h)$.
3. If $f = \Theta(g)$ and $g = \Theta(h)$ then $f = \Theta(h)$.

Sums of (up to a constant number of) functions

1. If $f = O(h)$ and $g = O(h)$ then $f + g = O(h)$.
2. Let k be a fixed constant, and let $f_1, f_2, \ldots, f_k, h$ be functions s.t. for all i, $f_i = O(h)$. Then $f_1 + f_2 + \ldots + f_k = O(h)$.

Transpose symmetry

- $f = O(g)$ if and only if $g = \Omega(f)$.
- $f = o(g)$ if and only if $g = \omega(f)$.