Analysis of Algorithms, I
CSOR W4231

Eleni Drinea

Computer Science Department

Columbia University

Insertion sort, efficient algorithms
Outline

1. Overview
2. A first algorithm: insertion sort
3. Analysis of algorithms
4. Efficiency of algorithms
1. Overview

2. A first algorithm: insertion sort

3. Analysis of algorithms

4. Efficiency of 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.
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

1. Overview

2. A first algorithm: insertion sort

3. Analysis of algorithms

4. Efficiency of algorithms
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: $n = 6$, $A = \{9, 3, 2, 6, 8, 5\}$
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

2. **Insert** the next available element of $A$, call it **key**, into its **correct** location in the **sorted** subarray to its left, thus increasing the size of the sorted subarray by 1. *How?*

2. **Insert** the next available element of $A$, call it **key**, into its **correct** location in the **sorted** subarray to its left, thus increasing the size of the sorted subarray by 1. **How**?

   - Compare **key** with every element $x$ of the sorted subarray, starting from the **right**.
     - If $x > \text{key}$, move $x$ one position to the right.
     - Else ($x \leq \text{key}$), **insert** **key** after $x$. 

2. **Insert** the next available element of $A$, call it **key**, into its **correct** location in the **sorted** subarray to its left, thus increasing the size of the sorted subarray by 1. **How?**
   
   - Compare **key** with every element $x$ of the sorted subarray, starting from the **right**.
     
     - If $x > \text{key}$, move $x$ one position to the right.
     - Else ($x \leq \text{key}$), **insert** **key** after $x$.

3. Repeat Step 2. until the sorted subarray has size $n$. 
Example of insertion sort: \( n = 6, A = \{9, 3, 2, 6, 8, 5\} \)

\[
\begin{array}{c}
\text{beginning of iteration } i=2 \\
\begin{array}{cc}
\text{sorted} & \text{unsorted} \\
9 & 3 & 2 & 6 & 8 & 5
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{beginning of iteration } i=3 \\
\begin{array}{cc}
\text{sorted} & \text{unsorted} \\
3 & 9 & 2 & 6 & 8 & 5
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{beginning of iteration } i=4 \\
\begin{array}{cc}
\text{sorted} & \text{unsorted} \\
2 & 3 & 9 & 6 & 8 & 5
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{beginning of iteration } i=5 \\
\begin{array}{cc}
\text{sorted} & \text{unsorted} \\
2 & 3 & 6 & 9 & 8 & 5
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{beginning of iteration } i=6 \\
\begin{array}{cc}
\text{sorted} & \text{unsorted} \\
2 & 3 & 6 & 8 & 9 & 5
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\text{end of iteration } i=6 \\
\begin{array}{cc}
\text{sorted} & \text{unsorted} \\
2 & 3 & 5 & 6 & 8 & 9
\end{array}
\end{array}
\]
Let \( A \) be an array of \( n \) integers.

\[
\text{insertion-sort}(A) \\
\text{for} \ i = 2 \ \text{to} \ n \ \text{do} \\
\quad \text{key} = A[i] \\
\quad // \text{Insert } A[i] \text{ into the sorted subarray } A[1, i - 1] \\
\quad j = i - 1 \\
\quad \text{while } j > 0 \ \text{and} \ A[j] > \text{key} \ \text{do} \\
\quad \quad A[j + 1] = A[j] \\
\quad \quad j = j - 1 \\
\quad \text{end while} \\
\quad A[j + 1] = \text{key} \\
\text{end for}
\]
Today

1. Overview

2. A first algorithm: insertion sort

3. Analysis of algorithms

4. Efficiency of algorithms
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 $n \geq 1$, \( \sum_{i=1}^{n} i = \frac{n(n+1)}{2} \).
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 $n = 2, 3, \ldots$. 
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 $\text{key} = A[i + 1]$ is inserted into $A[1, i]$. By the induction hypothesis, $A[1, i]$ is sorted. Since

  1. $\text{key}$ is inserted after the last element $A[\ell]$ such that $0 \leq \ell \leq i$ and $A[\ell] \leq \text{key}$;
  2. all elements in $A[\ell + 1, i]$ are pushed one position to the right with their order preserved,

  the statement is true for $i + 1$. 

Proof of the inductive step in a picture

A[\ell] is the rightmost element of A[0,i] such that A[\ell] \leq \text{key}

End of i-th iteration:
A[1,i] is sorted

End of i+1-st iteration:
A[1,i+1] is sorted
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
        $j = j - 1$
    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

\begin{verbatim}
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  
    j = j – 1  
  end while  
  A[j + 1] = key  
end for
\end{verbatim}

- For $2 \leq i \leq n$, let $t_i = \# \text{ times line 4 is executed.}$
Running time $T(n)$ of insertion-sort

\[
\text{for } i = 2 \text{ to } n \text{ do} \\
\text{key} = A[i] \\
//Insert A[i] into the sorted subarray A[1, i - 1] \\
j = i - 1 \\
\text{while } j > 0 \text{ and } A[j] > \text{key} \text{ do} \\
j = j - 1 \\
\text{end while} \\
A[j + 1] = \text{key} \\
\text{end for}
\]

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

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

- 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
  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
    $j = j - 1$
  end while
  $A[j + 1] =$ key
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: $\frac{3n^2}{2} + \frac{7n}{2} - 4$
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

1. Overview

2. A first algorithm: insertion sort

3. Analysis of algorithms

4. Efficiency of algorithms
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!

\[ \Rightarrow \text{Brute force solution is not efficient.} \]
Definition 3 (Attempt 1).

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

**Polynomial** running times: on input of size $n$, $T(n)$ is at most $c \cdot n^d$ for $c, d > 0$ constants.

- Polynomial running times scale well!
- The smaller the exponent of the polynomial the better.
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?
Insertion sort is efficient. Are we done with sorting?

1. Can we do better?

2. And what is better?
   - E.g., is $T(n) = n^2$ better than $\frac{3n^2}{2} + \frac{7n}{2} - 4$?
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: high-level language steps will expand by a constant factor that depends on the hardware.
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, which 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.

A faster algorithm for sorting using the divide-and-conquer principle.