1 Overview

Definition 1 *An algorithm is a well-defined computational procedure that transforms the input into the output.*

The desired input/output relationship is specified by the statement of the computational problem for which the algorithm is designed.

Definition 2 *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, i.e., how much time and space are used and how they **scale** as the input size grows.

We will primarily focus on efficiency in running time: we want algorithms that run quickly.

Definition 3 *The running time of an algorithm is the number of primitive computational steps performed.*

A primitive computational step is an operation that usually belongs to one of the following categories.

- **Arithmetic operations**: add, subtract, multiply, divide
- **Data movement operations:** load, store, copy
- **Control operations:** branching, subroutine call and return

For example, assigning a value to a variable, looking up an entry in an array, or adding two *fixed-size* integers are primitive computational steps.

We will use pseudocode to describe our algorithms. In general one line of pseudocode could correspond to one computational step.

2 A first algorithm: Insertion-Sort

Consider the problem of **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 such that $x'_1 \leq x'_2 \leq \ldots \leq x'_n$.

Intuition for Insertion-Sort: Start with a sorted subarray of size 1 consisting of the leftmost element of A. Then increase the size of the sorted subarray by one at a time, by inserting into the correct position the next element of A that does not belong to the sorted subarray so far. The correct position is found as follows: compare the new element with every item in the sorted subarray starting from its rightmost item and insert the new element after the first item that is smaller or equal to the new element.

2.1 Pseudocode

Insertion-Sort(A, n)
  for $j = 2$ to n **do**
    key $= A[j]$
    //Insert $A[j]$ into the sorted sequence $A[1 \ldots j-1]$

$i = j - 1$
while $i > 0$ and $A[i] >$ key **do**
$A[i+1] = A[i]$
$i = i - 1$
end while
$A[i+1] =$key
end for

Insertion-sort sorts *incrementally* and *in place*: besides the space required to store the input, the extra space required by the algorithm is a constant number of memory slots.

In the following two subsections we will analyze Insertion-Sort. The analysis of an algorithm consists of two separate yet equally important parts: proof of correctness and analysis of running time.

2.2 Correctness of Insertion-Sort

We will use **induction** on the size of the array $n \geq 1$. To refresh your memory on mathematical induction, we start with the following example.

Claim 1 *Show that for all* $n \geq 1$, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$.

Proof.

- **Base case:** For $n = 1$, we have $\sum_{i=1}^{1} i = 1 = \frac{1 \cdot 2}{2}$ hence the induction base holds.
- **Inductive Hypothesis:** Assume that the statement $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ is true for n.
- **Inductive Step:** We will show that it is true for $n+1$, i.e., that $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$.
  We have $\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) = \frac{n(n+1)}{2} + (n+1) = \frac{(n+1)(n+2)}{2}$.
- **Conclusion:** The statement is true for all n since it is true for $n = 1$ and we can repeatedly apply the inductive step to show it true for $n = 2, 3, \ldots$.

□

In general, to show that a statement $P(n)$ is true for all $n = 1, 2, \ldots$, we may use induction as follows. First we show that $P(1)$, the base case, is true. Next we assume that the statement is true for some $n \geq 1$ (induction hypothesis). We then show that $P(n+1)$ is true, using the truth of $P(n)$. Since $P(1)$ is true, applying the inductive step for all $n = 2, 3, \ldots$ shows that $P(n)$ is true for all $n \geq 1$.

Theorem 1 *Let* $n \geq 1$ *be a positive integer. Let* $A = (x_1, \ldots, x_n)$ *be the input to Insertion Sort. Then after the* i*-th loop, the subarray* $A[1, \ldots, i]$ *contains a nondecreasing permutation of* $x_1, \ldots, x_i$ *(the first* i *integers of* A*) for all* $1 \leq i \leq n$.

Proof. We will prove this theorem by induction on i.

- **Base case:** $i = 1$: If the array consists of a single element, then the algorithm does nothing since an array with a single element is trivially sorted.
- **Induction hypothesis:** Assume that after the i-th loop, the subarray $A[1, \ldots, i]$ contains a nondecreasing permutation of $x_1, \ldots, x_i$ (the first i integers of A) for some $1 \leq i < n$.
- **Inductive step:** We will show that the statement is true for $i+1$. In loop $i+1$, element $A[i+1] = x_{i+1}$ is inserted into $A[1, \ldots, i]$. By the induction hypothesis, $A[1, \ldots, i]$ is a nondecreasing permutation of $x_1, \ldots, x_i$. $A[i+1]$ is inserted after the first element $A[l]$ for $1 \leq l \leq i$ such that $x_{i+1} \geq A[l]$. All elements in $A[1, \ldots, i]$ greater than x_{i+1} are pushed one position to the right with their order preserved. Therefore, at the end of loop $i+1$, $A[1, \ldots, i+1]$ contains a nondecreasing permutation of $x_1, \ldots, x_{i+1}$.

□

2.3 Analysis of running time

The pseudocode for Insertion-Sort consists of 8 lines (ignore the EndWhile and EndFor lines that exist just for expositional purposes). Each line ℓ for $1 \leq \ell \leq 8$ and $\ell \neq 3$ corresponds to a primitive computational operation requiring time $c_\ell > 0$ to be executed (line 3 is a comment and therefore requires time $c_3 = 0$). Assume that line ℓ_5 is executed r_j times for element j. Then the running time $T_{IS}(n)$ of Insertion-Sort on input an array of n integers is given by the following equation:

$$T_{IS}(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} r_j + c_6 \sum_{j=2}^{n} (r_j - 1) + c_7 \sum_{j=2}^{n} (r_j - 1) + c_8(n-1) \quad (1)$$

- **Best case running time:** assume that A is already sorted in non-decreasing order. Then $r_j = 1$ for all $2 \leq j \leq n$; substituting this into equation (1) we easily conclude that

  $$T_{IS}(n) = \alpha n + \beta$$

  for real constants $\alpha > 0$, β.

- **Worst case running time:** assume that the input is sorted in non-increasing order. Then $r_j = j$ for all $2 \leq j \leq n$: element $A[j]$ will be compared with all elements to the left of it in A, hence $j - 1$ executions of line 5; one last execution of line 5 is required where $i = 0$ and thus the body of the while loop is not executed. Using Claim 1, it is easy to conclude from equation (1) that

  $$T_{IS}(n) = \alpha n^2 + \beta n + \gamma,$$

  for real constants $a > 0$, β, γ.

3 Discussion of efficiency

Back to the question of efficiency: is insertion-sort efficient?
To answer this question, let's consider for a moment the brute force way to solve the sorting problem:

1. At each step, generate a new permutation of the n integers.
2. If the permutation is in non-decreasing order, stop and output the permutation.

In the worst case, this algorithm will generate $n!$ permutations.

Definition 4 *The worst-case running time of an algorithm is the largest possible running time of the algorithm over all inputs of a given size n.*

From now on, unless explicitly stated differently, we will be analyzing the worst-case running time of an algorithm because

- this is a well defined method to derive efficiency bounds;
- average-case analysis can be tricky: how do we generate a "random" instance? The algorithm could perform well on a certain kind of random instances and not so well on others so our analysis might reveal more about the way the random instances are generated rather than the algorithm itself.

Back to the brute force solution to the sorting problem: to understand how this behavior scales as n grows, we recall Stirling's approximation formula for the factorial

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n.$$

So if we want to sort 1000 numbers the algorithm will generate on the order of 367^{1000} permutations in the worst case. This is enormous so clearly this approach will fail to give us an answer on inputs of length 1000 in the worst case. Since efficiency relates to the performance of the algorithm as n grows, we cannot call the brute force algorithm efficient.

Definition 5 *(Efficiency: attempt 1) An algorithm is efficient if it achieves better worst-case performance than brute-force search.*

Scaling properties: This definition captures the notion that an algorithm must improve upon the brute force solution to the problem. However we would still like to capture more concretely (in a quantifiable manner) the notion that the running time of an efficient algorithm should have good scaling properties.

In particular, if the input size n grows by a constant factor, i.e., becomes $\beta \cdot n$ for some constant $\beta > 0$, we would like the running time of the algorithm to increase by a constant factor as well, i.e., from $T(n)$ to $\gamma \cdot T(n)$, for some constant $\gamma > 0$ *independent* of n.

For example, if $T(n) = 2^n$, then doubling the input size from n to $2n$ (i.e., increasing the input by the constant factor of 2), results in a running time of 2^{2n}. Hence the running time increases by a factor of 2^n, an exponential in n amount!

However if we require the running time to be a polynomial in n, i.e., $T(n) \leq c \cdot n^k$ for constants $c > 0, k > 0$, then increasing n by a constant factor yields an increase of $T(n)$ by a constant factor.

For example, increasing the input size from n to βn for constant $\beta > 1$ yields a running time $T(\beta n) \leq cn^k \cdot \beta^k$. Since β^k is a constant independent of n, for $\gamma = \beta^k$, we obtain $T(\beta n) \leq \gamma n^k$, thus the running time has increased only by the constant factor γ. (Note that the smaller the exponent k of the polynomial the better.)

This leads to a new definition.

Definition 6 *An algorithm is efficient if it has a polynomial running time.*

Note that this definition might sound too restrictive: what if c and/or d are very large? However

- The algorithms we will consider tend to have **small degree polynomial** running times: $n, n \log n, n^2, n^3$.
- This definition allows us to distinguish between **easy** and **hard** problems: those that admit polynomial-time (efficient) algorithms are easy while those for which no polynomial-time algorithm is known are considered hard.

4 Asymptotic Order of Growth

Given Definition 6, Insertion-Sort is efficient. So are we done with sorting?
To answer this question, we should first answer a different question: Can we do better than Insertion-Sort? In particular, if there were an algorithm with a better running time than Insertion-Sort then that algorithm would be preferred for the problem of sorting.
But what exactly do we mean by "better" running time? For example, is $T_1(n) = \alpha' n^2 + \beta' n + \gamma'$ for $\alpha' = \alpha$ and $\beta' < \beta$ what we are aiming for or should we aim for $\alpha' < \alpha$ or should we aim for something different yet?
To address this issue, we note that only a coarse classification of running times for algorithms would be useful. Exact (or too detailed, such as $T_{IS}(n)$ and $T_1(n)$) characterizations are useless because they

- don't reveal similarities between running times in an immediate and useful way as n grows large.
- are often meaningless: pseudocode steps will *expand* by a *constant* factor depending on the architecture where they will be implemented. This means that lower order terms and even the constant of the highest-order term are irrelevant.

In what follows we present definitions that will allow us to compare the **rate of growth** of different functions (to the end of comparing the rate of growth of the running times of different algorithms) by ignoring constant factors and low-order terms and focusing entirely on the highest-order term.

4.1 Tight Asymptotic Upper and Lower Bounds

Definition 7 *We say that* $T(n) = O(f(n))$ *if there exist constants* $c > 0$ *and* $n_0 \geq 0$ *such that for all* $n \geq n_0$*, we have* $T(n) \leq c \cdot f(n)$*.*

E.g., $T(n) = an^2 + b$. Then $T(n) = O(n^2)$ (choose $c = a + b$). A weaker upper bound is $T(n) = O(n^{2.5})$.

Definition 8 *We say that* $T(n) = \Omega(f(n))$ *if there exist constants* $c > 0$ *and* $n_0 \geq 0$ *such that for all* $n \geq n_0$*, we have* $T(n) \geq c \cdot f(n)$*.*

E.g., $T(n) = an^2 + b$. Then $T(n) = \Omega(n^2)$ (choose $c = a$). A weaker lower bound is $T(n) = \Omega(n^{1.5})$.

Definition 9 *We say that* $T(n) = \Theta(f(n))$ *if there exist constants* $c_1, c_2 > 0$ *and* $n_0 \geq 0$ *such that for all* $n \geq n_0$*, we have* $c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$*.*

E.g., $T(n) = an^2 + b$, $a, b > 0$. Then $T(n) = \Theta(n^2)$ (choose $c_1 = a, c_2 = a + b$). Alternatively, $T(n) = \Theta(f(n))$ if $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$.

4.2 Asymptotic Upper and Lower Bounds that are *not* tight

Definition 10 *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, this notation says that $T(n)$ becomes insignificant relative to $f(n)$ as n approaches infinity. We can usually prove that $T(n) = o(f(n))$ by showing that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$ (if the limit exists).

Definition 11 *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, this notation says that $T(n)$ becomes arbitrarily large relative to $f(n)$ as n approaches infinity. Note that $T(n) = \omega(f(n))$ implies that $\lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$ (if the limit exists), and therefore $f(n) = o(T(n))$.

4.3 Properties of Asymptotic Growth Rates

Using the definitions above, we can show the following transitivity properties.

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 $f = \Theta(h)$.