Algorithm Correctness
• A correct algorithm is one in which every valid input instance produces the correct output.
• The correctness must be proved mathematically.

Algorithm Complexity
• Algorithm complexity is a measure of the resources an algorithm uses. The 2 resources we care about are:
  Time and Space
• For a given algorithm, we express these quantities as a function of the input size.

Algorithm Efficiency
*Time efficiency* indicates how fast an algorithm runs.

*Space efficiency* refers to the "extra" space (beyond that needed for the input) that the algorithm requires.

The amount of extra space required is of less concern for most programming applications than is the time required.

Time is major efficiency measure
*Time efficiency* indicates how fast an algorithm runs.

*Observation:* Almost all algorithms run longer on larger inputs.
Therefore, it is logical to investigate an algorithm's efficiency as a function of some parameter $n$ indicating the algorithm's input size.

The size of the input is usually obvious, but the running time may be influenced by other factors than the number of items in an array.

Algorithm efficiency may vary for different instances of equal size
Some algorithms take the same amount of time on all input instances of a given size.

For others, there are best-case, worst-case, and average-case input instances with running times that depend on more than just the input size.

For algorithm $A$ on input of size $n$:

*Worst-case input:* The input(s) for which $A$ executes the most steps for all inputs of size $n$.

*Best-case input:* The input(s) for which $A$ executes the fewest steps for all inputs of size $n$.

Analyzing Algorithms
Analyzing an algorithm in this course involves predicting the number of steps executed by an algorithm without implementing the algorithm.

We can do this in a machine- and language-independent way using:
1. the RAM model of computation (single processor)
2. asymptotic analysis of worst-case complexity
**RAM Model of Computation**

Single-processor RAM, instructions executed sequentially.

To make the notion of a step as machine-independent as possible, assume:

Each execution of the $i^{th}$ line takes time $c_i$, where $c_i$ is a constant.

**Algorithm Efficiency**

*Imprecise metric:* Experimental measurement of running time.

*Better metric:* Count the number of times each of the algorithm’s operations is executed. However, this exact count is often overkill.

*Best metric:* Identify the operation that contributes most to the total running time and count the number of times that operation is executed.

$\Rightarrow$ the basic operation

The basic operation is usually the statement or set of statements inside the most deeply nested loop.

**Counting Steps in Pseudocode**

**Pitfalls:**

- If the line is a subroutine call, then the actual call takes constant time, but the execution of the subroutine being called might not.
- If the line specifies operations other than primitive ones, then it might take more than constant time. Example: “sort the points by x-coordinate.”

**MaxElement**($A[1\ldots n]$)

1. $\text{maxval} = A[1]$
2. $\text{for } i = 2 \text{ to } n$
3. $\text{if } A[i] > \text{maxval}$
4. $\text{maxval} = A[i]$
5. $\text{return } \text{maxval}$

**Pseudocode**

**INPUT:** An array $A[1\ldots n]$ of comparable items

**OUTPUT:** The value of the largest element in $A$

- 1. $\text{maxval} = A[1]$
- 2. $\text{for } i = 2 \text{ to } n$
- 3. $\text{if } A[i] > \text{maxval}$
- 4. $\text{maxval} = A[i]$
- 5. $\text{return } \text{maxval}$

Give the line number(s) of the basic operation(s).

Does this algorithm have different running time on different input arrays of size $n$? If so, give examples of best- and worst-case input instances.

**Expressing Loops as Summations**

Express the worst-case running time for the basic operation on the last slide as a summation using the bounds given in the code.

\[ \sum_{i=2}^{n} 1 \]

Where does the 1 come from?

**Rule 1 for sum manipulation**

In general, the solution to a summation formula as shown on the last slide is ($u$ is for upper and $l$ is for lower):

\[ \sum_{i=l}^{u} 1 = u - l + 1 \]

So the solution to the summation we just saw is:

\[ \sum_{i=2}^{n} 1 = n - 2 + 1 = n - 1 \]
SequentialSearch(A[1…n], k)

Pseudocode

1. \(i = 1\)
2. while \(i < n+1\) and \(A[i] \neq k\)
3. \(i = i + 1\)
4. if \(i < n+1\) return \(i\)
5. else return \(-1\)

Does this algorithm have different running time on different input arrays of size \(n\)?

What is the worst-case input instance?

Give the line number(s) of the basic operation(s).

InsertionSort(A)

INPUT: An array \(A[1…n]\) of comparable items \(\{a_1, a_2, \ldots, a_n\}\)
OUTPUT: A permutation of the input array \(\{a_1, a_2, \ldots, a_n\}\) such that \(a_1 \leq a_2 \leq \cdots \leq a_n\).

1. for \(j = 2\) to length[A]
2. key = A[j]
3. \(i = j - 1\)
4. while \(i > 0\) and \(A[i] > key\)
5. \(A[i + 1] = A[i]\)
6. \(i = i - 1\)
7. \(A[i + 1] = key\)

Are best-case and worst-case different?


**Analysis of InsertionSort**

<table>
<thead>
<tr>
<th>InsertionSort(A)</th>
<th>times repeated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. for j = 2 to length(A)</td>
<td>n</td>
</tr>
<tr>
<td>2. key = A[j]</td>
<td>n-1</td>
</tr>
<tr>
<td>3. i = j - 1</td>
<td>n-1</td>
</tr>
<tr>
<td>4. while i &gt; 0 and A[i] &gt; key</td>
<td>( \sum t_i )</td>
</tr>
<tr>
<td>5. A[i + 1] = A[i]</td>
<td>( \sum (j - 1) )</td>
</tr>
<tr>
<td>6. i = i - 1</td>
<td>( \sum (j - 1) )</td>
</tr>
<tr>
<td>7. A[i + 1] = key</td>
<td>n-1</td>
</tr>
</tbody>
</table>

**Analysis of InsertionSort**

\[
\text{InsertionSort}(A) = \begin{cases} 
1. & \text{for } j = 2 \text{ to } \text{length}(A) \\
2. & \text{key} = A[j] \\
3. & i = j - 1 \\
4. & \text{while } i > 0 \text{ and } A[i] > \text{key} \\
6. & i = i - 1 \\
7. & A[i + 1] = \text{key} 
\end{cases}
\]

- For insertion sort, the running time varies for different input instances.

What is the "exact" running time for the best case?
\( T(n) = n^2 \) for worst-case input?
\( T(n) = 2n \) for best-case input?

What is the "exact" running time for the worst case?
\( T(n) = \frac{n^2}{2} + \frac{n}{2} + \frac{1}{4} + \frac{1}{4} \) for worst-case input?
\( T(n) = \frac{n^2}{2} + \frac{n}{2} + \frac{1}{4} + \frac{1}{4} \) for best-case input?

**Rule 2 for sum manipulation**

\[
\sum_{j=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \ldots + n = \frac{n(n+1)}{2}
\]

For insertion sort, the inner while loop is run the most when it is always the case that \( A[i] > \text{key} \), so the upper limit is \( t_j = j \).

\[
\sum_{j=2}^{n} t_j = \left( \sum_{j=1}^{n} j \right) - 1 = \frac{n(n+1)}{2} - 1
\]

\[
\sum_{j=2}^{n} \sum_{i=1}^{j-1} 1 = \left( \sum_{i=1}^{n} (j - 1) - 1 \right) + \left( \sum_{j=2}^{n} (j - 1) \right) = \frac{n(n-1)}{2} + \frac{n(n-1)}{2}
\]

**Why analyze running time on worst-case inputs?**

- The worst-case time gives a guaranteed upper bound for any input.
- For some algorithms, the worst case occurs often. For example, when searching for a given key in an array, the worst case often occurs because the key is not in the array.
- The average case frequently differs from the worst case by a factor of \( \frac{1}{2} \).

**UniqueElements**

INPUT: An array \( A[1..n] \) of comparable items

OUTPUT: Returns "true" if all items are unique and "false" otherwise

UniqueElements(A) =
1. for i = 1 to length[A]-1
2. for j = i + 1 to length[A]
4. return true

**Homework 1 (due next class)**

1. Is there a difference in \( T(n) \) for best- and worst-case input? Give examples of a best- and a worst-case solution.
2. Give the line number(s) of the basic operation.
3. Set up a sum representing the number of times the basic operation is executed in the worst-case (don't solve it).
4. What is the running time in the worst-case? (solve summation here)

**Proving Correctness Using Loop Invariants**

Using loop invariants is like mathematical induction:

- To prove that a property holds, you prove a base case and an inductive step.
  - Showing that the invariant holds before the first iteration is like the base case.
  - Showing that the invariant holds from iteration to iteration is like the inductive step.
- The termination part differs from the usual use of mathematical induction, in which the inductive step is used infinitely. We stop the "induction" when the loop terminates.
Correctness of InsertionSort

InsertionSort(A)
1. for j = 2 to length[A]
2. key = A[j]
3. i = j - 1
4. while i > 0 and A[i] > key
6. i = i - 1
7. A[i+1] = key

In order to show that InsertionSort actually sorts the items in A in non-decreasing order, what do we need to prove?
1. The algorithm terminates.
3. The elements of A’ form a permutation of A.

Proving Correctness--Insertion Sort

Loop invariant:
Let j be the position of the key in the array A.
At the start of each iteration of the for loop, the sub-array A[1...j-1] consists of the elements originally in A[1...j-1], but in sorted order.

Basis (initialization): When j = 2, A[1...j-1] has a single element and is therefore trivially sorted.

Maintenance Step or Inductive Hypothesis (IHOP): Assume the invariant holds through the beginning of the iteration in which the position of j=k.

When j = k, key = A[k]. By the IHOP, we know that the sub-array A[1...k-1] is in sorted order. During this iteration, items A[k-1], A[k-2], ..., A[1] and so on are each moved one position to the right until either a value less than key is found or until k-1 values have been shifted right, when the value of key is inserted.

Due to the total ordering on integers, key will be inserted in the correct position in the values A[1...k-1], so at the end of iteration k, the sub-array A[1...k] will contain only the elements that were originally in A[1...k], but in sorted order.

Therefore, the loop invariant holds at the start of the iteration when the position of j = k+1.

Termination: The for loop ends when j = n+1. By the IHOP, we have that the subarray A[1...n] is in sorted order. Therefore, the entire array is sorted and the algorithm ends correctly.

Proving Correctness--Insertion Sort

InsertionSort(A)
1. for j = 2 to length[A]
2. key = A[j]
3. i = j - 1
4. while i > 0 and A[i] > key
6. i = i - 1
7. A[i+1] = key

We need to show that the loop invariant is true:
1. ...prior to the first iteration (basis or initialization).
2. ...before each iteration (inductive hypothesis), so it remains true for the next iteration (inductive step or maintenance).
3. ...when the loop terminates the invariant allows us to argue that the algorithm is correct (termination).
BubbleSort

1. for $i = 1$ to $A.length - 1$
2. for $j = A.length$ downto $i + 1$

How do we express the data size?
Does the algorithm have the same or different running time for all inputs of size $n$?
Give a summation for the running time of BubbleSort and solve it.
In order to show that BubbleSort actually sorts the items in $A$ in non-decreasing order, what do we need to prove?
State a loop invariant for the outer for loop and prove that the loop invariant holds.

Order of Growth

An abstraction to ease analysis. Usually expressed as $\Theta$

Asymptotic analysis calculates algorithm running time in terms of its rate of growth with increasing problem size. To make this task easier, we can
- Drop lower order terms
- Ignore the constant coefficient in the leading term.
For insertion sort the order of growth is $n^2$, meaning that when the input size doubles, the running time quadruples.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Asymptotic Analysis

Main idea: We are interested in the running time in the limit as the input size grows to infinity.

InsertionSort is an example of an incremental algorithm because it processes each element in sequence.

In the worst-case, $T(n)$ for insertion sort grows like $n^2$ (but it is a mistake to say $T(n)$ equals $n^2$).

In the best-case, $T(n)$ for insertion sort grows like $n$.

We usually consider one algorithm to be more efficient than another if its worst-case running time has a smaller order of growth.

Analysis of Divide-and-Conquer Algorithms

The divide-and-conquer paradigm (Ch.2, Sect.3)
- divide the problem into a number of subproblems
- conquer the subproblems (solve them)
  - Base case: If the subproblems are small enough, solve them by brute force
  - combine the subproblem solutions to get the solution to the original problem

Example: Merge Sort
- divide the n-element sequence to be sorted into two n/2-element sequences.
- conquer the subproblems recursively. (Base case occurs when the sequences are of size 1).
- combine the resulting two sorted n/2-element sequences by merging them together.

Divide and conquer algorithms generally involve recursion. To analyze recursive algorithms, we need to take a closer look at logarithms.

Review of Logarithms

A logarithm is an inverse exponential function. Saying $b^x = y$ is equivalent to saying $\log_b y = x$.

- notation convention for logarithms:
  \[
  \log n = \log_{10} n \quad \text{(binary logarithm)}
  \]
  \[
  \ln n = \log_e n \quad \text{(natural logarithm)}
  \]

- properties of logarithms:
  \[
  \log_a(xy) = \log_a x + \log_a y
  \]
  \[
  \log_a \left(\frac{x}{y}\right) = \log_a x - \log_a y
  \]
  \[
  \log_a x^d = d \log_a x
  \]
  \[
  \log_a x = \frac{\log_b x}{\log_b a} \quad \text{(e.g., } \log_2 2^n = 2^{\log_2 n})
  \]

- \textbf{While exponential functions grow very fast, log functions grow very slowly.}

More Notation

- Floor: \[ \lfloor x \rfloor = \text{the largest integer } \leq x \]
- Ceiling: \[ \lceil x \rceil = \text{the smallest integer } \geq x \]
- Geometric series: \[ \sum_{i=0}^{x} \left(\frac{1}{2}\right)^i = \frac{1}{2^{x+1}} - 1 \]
- Harmonic series: \[ H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} = \ln n + c \]
- Telescoping series: \[ \sum_{i=0}^{x} (a_i - a_{i+1}) = a_0 - a_i \]
  \[ \sum_{i=0}^{n} \frac{1}{k(k+1)} = \sum_{i=0}^{n} \left(\frac{1}{k} - \frac{1}{k+1}\right) = 1 - \frac{1}{n} \]
General Plan for Analyzing Time Efficiency of Recursive Algorithms

1. Decide on a parameter indicating input size.
2. Identify the algorithm’s basic operation.
3. Figure out whether the number of times the basic operation is executed varies on different inputs of the same size; if it can, you may need to consider the cases separately. Usually, concentrate on worst-case.
4. Set up a recurrence relation, with the appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or otherwise ascertain the order of growth.

Analyzing Merge-Sort

\[
T(n) = \begin{cases} 
\theta(1) & \text{if } n = 1 \\
2T(n/2) + \Theta(n) & \text{otherwise} 
\end{cases}
\]

- \(a = 2\) (two sub-problems)
- \(n/b = n/2\) (each sub-problem has size approx. \(n/2\))
- \(D(n) = \Theta(1)\) (just compute midpoint of array)
- \(C(n) = \Theta(n)\) (merging can be done by scanning sorted sub-arrays)

Recurrence for worst-case running time for Merge-Sort
Solving the Merge-Sort recurrence

There are several methods to solve a recurrence relation like the one for merge-sort. The easiest for this exact form of relation is to use the Master Theorem, which we will see in Ch. 4.

Another way is to use a recursion tree which shows successive expansions of the recurrence.

Start with an initial cost of $cn$ as the root of the tree. The original problem has 2 subproblems of size $n/2$. Each of those subproblems has 2 subproblems of size $n/4$. Continue until the size of the subproblems is 1.

Recursion Tree for Merge-Sort

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + cn & \text{otherwise} \end{cases}$$

Recurrence for worst-case running time of Merge-Sort

Another example:

**Algorithm** PrefixAverages(X):

*Input:* An array $X[1...n]$ of numbers.
*Output:* An array $A[1...n]$ of numbers such that $A[i]$ is the average of elements $X[1], \ldots, X[i]$.

1. Create an array $A$ such that length[$A$] = $n$
2. $s = 0$
3. for $j = 1$ to $n$
4.   $s = s + X[j]$
5.   $A[j] = s / j$
6. return $A$

What is the line number(s) of the basic operation?

Set up a sum for the number of times the basic operation is run.

Give the worst-case running time.

Proving Correctness—PrefixAverages

**Loop invariant:**

At the start of each iteration of the for loop,

$s = (X[1]+\ldots+X[j-1])$ and $A[j-1] = s/(j-1)$

**PrefixAverages**($X$)

1. Create array $A$ of length $n$
2. $s = 0$
3. for $j = 1$ to $n$
4.   $s = s + X[j]$
5.   $A[j] = s / j$
6. return $A$