Algorithms, Design and Analysis
Introduction.

Algorithm
- An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

Computing Prefix Averages
- Input: Array \( X[1..n] \)
- Output: Array \( A[1..n] \) of prefix averages of \( X; i \)-th prefix average = average of the first \( i \) elements of \( X \):
  \[ A[i] = \frac{X[1] + X[2] + \ldots + X[i]}{i} \]
- Computing the array \( A \) of prefix averages of another array \( X \) has applications to financial analysis.

Prefix Averages (Quadratic)
- The following algorithm computes prefix averages in quadratic time by applying the definition

```
Algorithm prefixAverages1(X, n)
Input array X of n integers
Output array A of prefix averages of X
1. A ← new array of n integers
2. for i ← 1 to n do
3.   s ← X[1]
4.   for j ← 2 to i do
5.     s ← s + X[j]
6.   A[i] ← s / i
7. return A
```

Prefix Averages (Linear, non-recursive)
- The following algorithm computes prefix averages in linear time by keeping a running sum.

```
Algorithm prefixAverages2(X, n)
Input array X of n integers
Output array A of prefix averages of X
1. A ← new array of n integers
2. s ← 0
3. for i ← 1 to n do
4.   s ← s + X[i]
5.   A[i] ← s / i
6. return A
```

Prefix Averages (Linear)
- The following algorithm computes prefix averages in linear time by computing prefix sums (and averages).

```
Algorithm recPrefixSumAndAverage(X, A, k)
Input array X[1..n] of integers, integer k, 1 ≤ k ≤ n integer.
Output array A[1..k] changed to hold prefix averages of X
2. if k = 1 then
3.   return A[1]
4.   tot ← recPrefixSumAndAverage(X, A, k-1)
5.   tot ← tot + X[k]
6.   A[k] ← tot / k
7. return tot
```

Prefix Averages (Linear, non-recursive)
- The following algorithm computes prefix averages in linear time by keeping a running sum.
Selection sort

```plaintext
Algorithm SelectionSort(A[0..n-1]):
// The algorithm sorts a given array by selection sort
// Input: An array A[0..n-1] of unordered elements
// Output: Array A[0..n-1] sorted in ascending order
for i = 0 to n-2 do
  min := i
  for j = i+1 to n-1 do
      min := j
  swap A[i] and A[min]
```

Insertion sort

```plaintext
Algorithm InsertionSort(A[0..n-1]):
// The algorithm sorts a given array by insertion sort
// Input: An array A[0..n-1] of unordered elements
// Output: Array A[0..n-1] sorted in ascending order
for i = 1 to n-1 do
  j := i
    swap A[j-1] and A[j]
    j := j-1
```

Mystery algorithm

```plaintext
for i := 1 to n-1 do
  max := i
  for j := i+1 to n do
      max := j
  swap A[i] and A[max]
```

### What is an algorithm?

- Recipe, process, method, technique, procedure, routine,... with following requirements:
  1. **Finiteness**: It terminates after a finite number of steps
  2. **Definiteness**: Rigorously and unambiguously specified
  3. **Input**: Valid inputs are clearly specified
  4. **Output**: Can be proved to produce the correct output given a valid input
  5. **Effectiveness**: Steps are sufficiently simple and basic

### Pseudocode

**Very High-level pseudocode**

```
Algorithm arrayMax(A, n):
  Input array A of n integers
  Output maximum element of A
  currentMax := A[0]
  for i := 1 to n do
    if A[i] > currentMax then
      currentMax := A[i]
  return currentMax
```

**Detailed pseudocode**

```
Algorithm arrayMax(A, n):
  Input array A of n integers
  Output maximum element of A
  currentMax := A[0]
  for i := 1 to n do
    if A[i] > currentMax then
      currentMax := A[i]
  return currentMax
```
Pseudocode Details

- Control flow
  - if...
  - while...
  - repeat...
  - for...
- Method declaration
  - Algorithm method (arg, arg...) Input...
  - Output...
- Method call
  - var.method (arg, arg...) Return value
- Expressions
  - Assignment (like in Java)
  - Equality testing (like in Java)
- Superscripts and other mathematical formatting allowed

Theoretical analysis of time efficiency

Time efficiency is analyzed by determining the number of repetitions of the basic operation as a function of input size.

- Basic operation: the operation that contributes most towards the running time of the algorithm

\[ T(n) \]

Input size and basic operation examples

<table>
<thead>
<tr>
<th>Problem</th>
<th>Input size measure</th>
<th>Basic operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search for key in list of n items</td>
<td>Number of times in list</td>
<td>Key comparison</td>
</tr>
<tr>
<td>Multiply two matrices of floating point numbers</td>
<td>Dimensions of matrices</td>
<td>Floating point multiplication</td>
</tr>
<tr>
<td>Compute ( a^n )</td>
<td>( a )</td>
<td>Floating point multiplication</td>
</tr>
<tr>
<td>Graph problem</td>
<td>Vertices and/or edges</td>
<td>Visiting a vertex or traversing an edge</td>
</tr>
</tbody>
</table>

Best-case, average-case, worst-case

For some algorithms efficiency depends on type of input:

- Worst case: \( W(n) \) – maximum over inputs of size \( n \)
- Best case: \( B(n) \) – minimum over inputs of size \( n \)
- Average case: \( A(n) \) – “average” over inputs of size \( n \)
  - Number of times the basic operation will be executed on typical input
  - NOT the average of worst and best case
  - Expected number of basic operations repetitions considered as a random variable under some assumption about the probability distribution of all possible inputs of size \( n \)

Worst-case count, all operations

- Worst-case operations count, as a function of the input size

Algorithm arrayMax (A, n) # operations
\[
\begin{array}{l}
\text{currentMax} \leftarrow A[0] \\
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do} \\
\quad \text{if } A[i] > \text{currentMax} \text{ then} \\
\quad \quad \text{currentMax} \leftarrow A[i] \\
\quad \quad [\text{increment counter } i ] \\
\text{return currentMax}
\end{array}
\]
Total \( 3n - 2 \)

Best-case Count of All Operations

- Best-case operations count, as a function of the input size

Algorithm arrayMax (A, n) # operations
\[
\begin{array}{l}
\text{currentMax} \leftarrow A[0] \\
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do} \\
\quad \text{if } A[i] > \text{currentMax} \text{ then} \\
\quad \quad \text{currentMax} \leftarrow A[i] \\
\quad \quad [\text{increment counter } i ] \\
\text{return currentMax}
\end{array}
\]
Total \( n \)
Count of Basic Operations

- Let basic operation = key comparison
- Then best-case and worst-case same for this method

Algorithm

```
arrayMax(A, n)

# operations

currentMax ← A[0]
for i ← 1 to n − 1 do
    if A[i] > currentMax then
        currentMax ← A[i]  
        (increment counter i )
return currentMax

Total 2n - 2
```

Defining Worst \( [W(n)] \), Best \([B(N)]\), and Average \([A(n)]\)

- Let \( I_n \) = set of all inputs of size \( n \).
- Let \( t(i) \) = # of ops by alg on input \( i \).
- \( W(n) = \) maximum \( t(i) \) taken over all \( i \) in \( I_n \)
- \( B(n) = \) minimum \( t(i) \) taken over all \( i \) in \( I_n \)
- \( A(n) = \sum_{i \in I_n} p(i) \cdot t(i) \), \( p(i) = \) prob. of \( i \) occurring.

- We focus on the worst case
  - Easier to analyze
  - Usually want to know how bad can algorithm be
  - Average-case requires knowing probability; often difficult to determine

Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm

```
prefixAverages1(X, n)

Input array \( X \) of \( n \) integers
Output array \( A \) of prefix averages of \( X \)

1. \( A \) ← new array of \( n \) integers
2. for i ← 1 to \( n \) do
3. \( s ← X[1] \)
4. for j ← 2 to \( i \) do
5. \( s ← s + X[j] \)
6. \( A[i] ← s/i \)
7. return \( A \)
```

Analysis of prefixAverages1

- Let Basic Operation = key additions (additions between array elements)
- The running time of \( \text{prefixAverages1} \) is \( 1 + 2 + \ldots + n - 1 \)
- The sum of the first \( n - 1 \) integers is \( n(n - 1)/2 \)
  - There is a simple visual proof of this fact
- Thus, algorithm \( \text{prefixAverages1} \) runs in \( \Theta(n^2) \) time

Prefix Averages, Linear

- Recurrence equation
  - \( T(1) = 6 \)
  - \( T(n) = 13 + T(n-1) \) for \( n > 1 \).
- Solution of recurrence is
  - \( T(n) = 13(n-1) + 6 \)
  - \( T(n) = O(n) \).
Empirical analysis of time efficiency

- Select a specific (typical) sample of inputs
- Use physical unit of time (e.g., milliseconds)
  OR
- Count actual number of basic operations
- Analyze the empirical data

Types of formulas for basic operation count

- Exact formula
e.g., \( C(n) = n(n-1)/2 \)
- Formula indicating order of growth with specific multiplicative constant
e.g., \( C(n) \sim 0.5 n^2 \)
- Formula indicating order of growth with unknown multiplicative constant
e.g., \( C(n) \sim cn^2 \)

Time efficiency of nonrecursive algorithms

Steps in mathematical analysis of nonrecursive algorithms:

- Decide on parameter \( n \) indicating input size
- Identify algorithm’s basic operation
- Determine worst, average, and best case for input of size \( n \)
- Set up summation for \( C(n) \) reflecting algorithm’s loop structure
- Simplify summation using standard formulas (see Appendix A)

Example: Sequential search

- Problem: Given a list of \( n \) elements and a search key \( K \), find an element equal to \( K \) if any.
- Algorithm: Scan the list and compare its successive elements with \( K \) until either a matching element is found (successful search) or the list is exhausted (unsuccessful search)
- Worst case
- Best case
- Average case

Time efficiency of recursive algorithms

Steps in mathematical analysis of recursive algorithms:

- Decide on parameter \( n \) indicating input size
- Identify algorithm’s basic operation
- Determine worst, average, and best case for input of size \( n \)
- Set up a recurrence relation and initial condition(s) for \( C(n) \)—the number of times the basic operation will be executed for an input of size \( n \) (alternatively count recursive calls).
- Solve the recurrence to obtain a closed form or estimate the order of magnitude of the solution (see Appendix B)