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

- asymptotic analysis examples:
  - two algorithms for prefix averages
  - The $i$-th prefix average of an array $X$ is average of the first $(i + 1)$ elements of $X$:
    
    $$A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{i + 1}$$

  - 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 ← 0 to n − 1 do
3.  s ← X[0]
4.  for j ← 1 to i do
5.   s ← s + X[j]
6.  A[i] ← s / (i + 1)
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 ← 0 to n − 1 do
4.   s ← s + X[i]
5.   A[i] ← s / (i + 1)
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, n)
Input array X of n ≥ 1 integer.
Empty array A; A is same size as X
Output array A[0],...,A[n-1] changed to hold prefix averages of X
1. if n = 1 then
2.   A[0] ← X[0]
3. return A[0]
4. tot ← recPrefixSumAndAverage(X, A, n-1)
5. tot ← tot + X[n-1]
6. A[n-1] ← tot / n
7. return tot;
```
### Selection sort

**Algorithm:** SelectionSort(A[0..n-1])

// The algorithm sorts an array by selection sort
// Input: A array of length n
// Output: A[0..n-1] sorted in ascending order

for i := 1 to n-2 do
  max := i;

  for j := i+1 to n do
      max := j;

  swap A[i] and A[max];

\[\text{Selection sort continues...}\]

### Insertion sort

**Algorithm:** InsertionSort(A[0..n-1])

// The algorithm sorts an array by insertion sort
// Input: A array of length n
// Output: A[0..n-1] sorted in ascending order

for i := 1 to n-1 do
  j := i;

    swap A[j] and A[j-1];

  j := j - 1;

\[\text{Insertion sort continues...}\]

### Mystery algorithm

for i := 1 to n-1 do
  max := i;

  for j := i+1 to n do
      max := j;

  swap A[i] and A[max];

\[\text{Mystery algorithm continues...}\]

### What is an algorithm?

- Recipe, process, method, technique, procedure, routine,... with following requirements:
  1. **Finiteness**
     - 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**

### Pseudocode

**Algorithm arrayMax(A, n)**

*Input:* array A of n integers
*Output:* maximum element of A

1. currentMax := A[0]
2. for i := 1 to n do
   a. if A[i] > currentMax then
      currentMax := A[i]
3. return currentMax

**Very High-level pseudocode**

**Algorithm arrayMax(A, n)**

*Input:* array A of n integers
*Output:* maximum element of A

1. currentMax := A[0]
2. Step through each element in A, updating currentMax when a bigger element is found
3. return currentMax

**Detailed pseudocode**

**Algorithm arrayMax(A, n)**

*Input:* array A of n integers
*Output:* maximum element of A

1. currentMax := A[0]
2. for i := 1 to n do
   a. if A[i] > currentMax then
      currentMax := A[i]
3. return currentMax

**Pseudocode**

- Mixture of English, math expressions, and computer code
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues
- Can write at different levels of detail.
Pseudocode Details

- Control flow
  - `if` then ... (else ...)
  - `while` do ...
  - `repeat` until ...
  - `for` do ...
  - Indentation replaces braces
- Method declaration
  - `Algorithm method (arg, arg ...)`
- Method call
  - `var.method (arg, arg ...)`
- Return value
  - `return expression`
- 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

\[
T(n) \approx c_n C(n)
\]

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-jacs</td>
<td>Number of items 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>(n)</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>

Counting Primitive Operations (§1.1)

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

\[
\text{Algorithm:} \quad \text{arrayMax}(A, n)
\]

\[
\begin{align*}
\text{currentMax} & \leftarrow A[0] \\
\text{for } i & \leftarrow 1 \text{ to } n - 1 \text{ do} \\
\text{if } A[i] & > \text{currentMax} \text{ then} \\
\text{currentMax} & \leftarrow A[i] \\
\{ \text{increment counter } i \} & \text{2}(n - 1) \\
\text{return } \text{currentMax} & \text{1} \\
\end{align*}
\]

Total \(7n - 2\)

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 primitive ops by alg on input \(i\).
- \(W(n) = \text{maximum } t(i) \text{ taken over all } i \in I_n\)
- \(B(n) = \text{minimum } t(i) \text{ taken over all } i \in I_n\)
- \(A(n) = \sum_{i \in I_n} p(i) t(i) \text{ prob. of } i \text{ occurring.}\)

- **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
Arithmetic Progression

- The running time of \textit{prefixAverages1} is \(O(1 + 2 + 3 + \ldots + n)\).
- The sum of the first \(n\) integers is \(\frac{n(n + 1)}{2}\).
- There is a simple visual proof of this fact.
- Thus, algorithm \textit{prefixAverages1} runs in \(O(n^2)\) 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

Best-case, average-case, worst-case

For some algorithm, 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

Types of formulas for basic operation count

- Exact formula
  e.g., \(C(n) = \frac{n(n-1)}{2}\)
- Formula indicating order of growth with specific multiplicative constant
  e.g., \(C(n) \sim 0.5 \cdot n^2\)
- 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

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
#operations
1. A ← new array of n integers \( n \)
2. for \( i ← 0 \) to \( n - 1 \) do \( n \)
3. \( x ← X[0] \)
4. for \( j ← 1 \) to \( i \) do \( 1 + 2 + \ldots + (n - 1) \)
5. \( s ← s + X[j] \)
6. \( A[i] ← s / (i + 1) \)
7. return A
```

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)