Algorithms, Design and Analysis

Introduction.

Algorithm

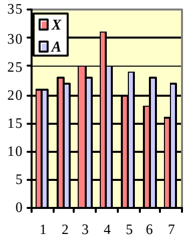
- An _algorithm_ is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

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:

$$A[i] = (X[1] + X[2] + \ldots + X[i])/i$$

- 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
    A ← new array of n integers
    for i ← 1 to n do
        s ← X[1]
        for j ← 2 to i do
            s ← s + X[j]
        A[i] ← s / i
    return A
```

Prefix Averages (Linear, non-recursive)

- The following algorithm computes prefix averages in linear time by keeping a running sum

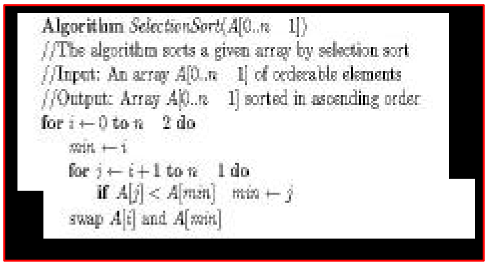
```
Algorithm prefixAverages2(X, n)
    Input array X of n integers
    Output array A of prefix averages of X
    A ← new array of n integers
    s ← 0
    for i ← 1 to n do
        s ← s + X[i]
        A[i] ← s / i
    return A
```

Prefix Averages (Linear)

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

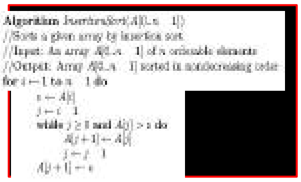
```
Algorithm recPrefixSumAndAverage(X, A, k)
    Input array X[1..n] of integers, integer k,  1 ≤ k ≤ n integer.
          Empty array A of same size as X.
    Output array A[1…k] changed to hold prefix averages of X.
          returns sum of X[1], X[2],…,X[k]
    if k=1
            A[1] ← X[1]
            return A[1]
    tot ← recPrefixSumAndAverage(X,A,k-1)
    tot ← tot + X[k]
    A[k] ← tot / k
    return tot;
```

Selection sort



```
Algorithm SelectionSort(A[0..n − 1])
//The algorithm sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable 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
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

Insertion sort



```
Algorithm InsertionSort(A[0..n − 1])
//Sorts a given array by insertion sort
//Input: An array A[0..n − 1] of n orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 1 to n − 1 do
    v ← A[i]
    j ← i − 1
    while j ≥ 0 and A[j] > v do
        A[j + 1] ← A[j]
        j ← j − 1
    A[j + 1] ← v
```

Mystery algorithm

```
for i := 1 to n - 1 do
    max := i ;
    for j := i + 1 to n do
        if |A[ j, i ]| > |A[ max, i ]| then max := j ;
    for k := i  to n + 1 do
        swap A[ i, k ]  with  A[ max, k ];
    for j := i + 1 to n do
        for k := n + 1 downto i do
            A[ j, k ] := A[ j, k ]  - A[ i, k ] * A[ j, i ] / A[ i, i ] ;
```

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
    - steps are sufficiently simple and basic

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.

Very High-level pseudocode:

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A
  currentMax ← A[0]
  Step through each element in A,
   updating currentMax when a
   bigger element is found
  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.

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 − 1 do
    if A[i] > currentMax then
      currentMax ← A[i]
  return currentMax
```

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm ***method*** (*arg* [, *arg* …])

  Input …

  Output …
- Method call

  var.method (*arg* [, ***arg*** …])
- Return value

  return ***expression***
- Expressions

  ← Assignment (like = in Java)

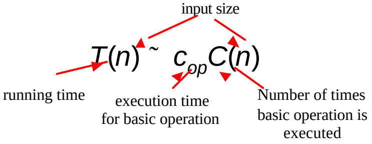
  = Equality testing (like == in Java)

  n^2 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



$$T(n) \approx c_{op}C(n)$$

Input size and basic operation examples

Problem	*Input size measure*	*Basic operation*
Search for key in list of *n* items	**Number of items in list *n***	**Key comparison**
Multiply two matrices of floating point numbers	**Dimensions of matrices**	**Floating point multiplication**
Compute a^n	***n***	**Floating point multiplication**
Graph problem	**#vertices and/or edges**	**Visiting a vertex or traversing an edge**

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
    currentMax ← A[0]                               2
    for i ← 1 to n – 1 do                          1 + n
        if A[i] > currentMax then                 2(n – 1)
                currentMax ← A[i]                 2(n – 1)
    { increment counter i }                       2(n – 1)
    return currentMax                                1

                                                  Total    7n – 2
```

Best-case Count of All Operations

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

```
Algorithm arrayMax(A, n)                          # operations
    currentMax ← A[0]                               2
    for i ← 1 to n – 1 do                          1 + n
        if A[i] > currentMax then                 2(n – 1)
                currentMax ← A[i]                   0
    { increment counter i }                       2(n – 1)
    return currentMax                                1

                                                  Total    5n
```

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                 2(n – 1)
            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) = $\sum_{i \in I_n} p(i)t(i)$, 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

---

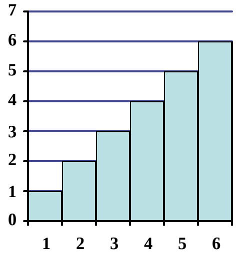
Prefix Averages (Quadratic)

- 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
A ← new array of n integers
for i ← 1 to n do
    s ← X[1]
    for j ← 2 to i do
        s ← s + X[j]
    A[i] ← s / i
return A
```

Analysis of prefixAverages1

- Let Basic Operation = key additions (additions between array elements)
- The running time of ***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 ***prefixAverages1*** runs in $O(n^2)$ time



Analysis of recPrefixSumAndAverage

- Let's count all operations, worst-case. Use recurrence equation.

```
Algorithm recPrefixSumAndAverage(X, A, n)             T(n) operations
    Input array X of  n ³  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.
          returns sum of X[0], X[1],…,X[n-1]          #operations
    if n=1                                             1
        A[0] ← X[0]                                    3
        return A[0]                                    2
    tot ← recPrefixSumAndAverage(X,A,n-1)              3+T(n-1)
    tot ← tot + X[n-1]                                 4
    A[n-1] ← tot / n                                   4
    return tot;                                        1
```

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) is 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) ˜ 0.5 n^2

- Formula indicating order of growth with unknown multiplicative constant

  e.g., C(n) ˜ 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*) of 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)