Algorithmic Thinking and Terminology

- Problem Formulation
- Algorithm Design Strategy: Implementing the Definition
- Algorithms Runtime Analysis
- Basic Numbers and Vectors Arithmetic

Imdad ullah Khan

Parity Test: Odd/Even integer

Input: An integer A
Output: True if A is even, else False

if $A \bmod 2 = 0$ **then**
  return true

Pseudocode

- A plain English description of "steps" of algorithm
- Use structural conventions like C/JAVA
- Focus on solution rather than technicalities of programming language

Parity Test: Odd/Even integer

Input: An integer A
Output: True if A is even, else False

if $A \bmod 2 = 0$ **then**
  return true

Issues:

- The above algorithm only works if A is given in an **int**
- What if A doesn't fit an **int** and A's digits are given in an array?
- What if A is given in binary/unary/...?

▷ These issues are in addition to usual checks of valid input

Parity Test: Odd/Even integer

Input: An integer A
Output: True if A is even, else False

If 'digits' of A digits are given in an array

$A =$

6	5	4	3	2	1	0
4	6	9	2	7	5	8

if $A[0] \bmod 2 = 0$ **then**
  return true

Questions computer scientists would (must) ask?

What is the problem?

- What is input/output?, what is the "*format*"?
- What are the "*boundary cases*", "*easy cases*", "*bruteforce solution*"?
- What are the available "*tools*"?

Do not jump to solution, spend time on problem formulation

Formulating the problem with precise definitions often yield a solution

▷ e.g. both the above algorithms just use definitions of even numbers

This is *implementing the definition* algorithm design paradigm

▷ The bruteforce solution

What is the dumbest/obvious/laziest way to solve the problem? What is the easiest cases? what are the hardest cases? where is the hardness?

Questions computer scientists would (must) ask?

Input: An integer A
Output: True if A is even, else False

If digits of A are given in an array

$A =$

6	5	4	3	2	1	0
4	6	9	2	7	5	8

if $A[0] \bmod 2 = 0$ **then**
  return true

What if mod is not available?

▷ What are the tools available?

Just check if $A[0] \in \{0, 2, 4, 6, 8\}$

if $A[0] = 0$ **then**
  return true
else if $A[0] = 2$ **then**
  return true
  ⋮
else
  return false

Questions computer scientists would (must) ask?

Is the algorithm "correct"?

- Does it do what it is "*supposed*" to do? ▷ requirement specification
- Does it always "*produce*" the "*correct output*"?
- Does it work for all "*legal inputs*"?

An extremely important step! Without a convincing argument for correction, we cannot call it an algorithm or solution

▷ Relies heavily on the problem formulation

Parity Test: Odd/Even integer

Input: An integer A
Output: True if A is even, else False

if $A \bmod 2 = 0$ **then**
  return true

if $A[0] \bmod 2 = 0$ **then**
  return true

if $A[0] = 0$ **then**
  return true
else if $A[0] = 2$ **then**
  return true
  ⋮
else
  return false

Correctness of these 3 algorithms follows from definition of even/odd and/or mod, depending on how we formulate the problem

Questions computer scientists would (must) ask?

How much "resources" does the algorithm consume?

Analysis of Algorithms: the theoretical study of performance and resource utilization of algorithms

How to measure the "goodness" of an algorithms?

- Time consumption
- Space and memory consumption
- Bandwidth consumption or number of messages passed
- Energy consumption
- ⋮

How to measure runtime?

Clock-time of algorithm execution is not a suitable measure

- Depends on machine/hardware, operating systems, other concurrent programs, implementation language and style etc.
- We want platform and implementation language independent

Number of operations is the right framework

- Measure runtime in terms of number of elementary operations
- Assuming each elementary operation takes fixed computation time
- Important to decide which operations are counted as elementary

if $A \bmod 2 = 0$ **then**
  return true

Number of operations: 1 mod and 1 comparison

Runtime as a function of input size

We want a consistent mechanism to measure efficiency that is platform and implementation language independent

Number of elementary operations depends on the actual input

Measure runtime by number of operations as a function of size of input

$\triangleright$ Has predictive value with respect to increasing input sizes

Size of input: usually number of bits needed to encode the input instance, can be length of an array, number of nodes in a graph etc.

Issue: For inputs of fixed size (n) there could be different runtimes depending on different instances

Parity Test: Odd/Even integer

Input: An integer A
Output: True if A is even, else False

If digits of A are given in an array

$$A = \begin{array}{|c|c|c|c|c|c|c|} \multicolumn{1}{c}{\scriptstyle 6} & \multicolumn{1}{c}{\scriptstyle 5} & \multicolumn{1}{c}{\scriptstyle 4} & \multicolumn{1}{c}{\scriptstyle 3} & \multicolumn{1}{c}{\scriptstyle 2} & \multicolumn{1}{c}{\scriptstyle 1} & \multicolumn{1}{c}{\scriptstyle 0} \\ \hline 4 & 6 & 9 & 2 & 7 & 5 & 8 \\ \hline \end{array}$$

If mod is not available

Just check if $A[0] \in \{0, 2, 4, 6, 8\}$

if $A[0] = 0$ **then**
  return true
else if $A[0] = 2$ **then**
  return true
  ⋮
else
  return false

What is the number of comparisons when $A[0] = 0$ and when $A[0] = 8$?

Best/Worst/Average Case

Issue: For inputs of fixed size (n) there could be different runtimes depending on different instances

Let $T(I)$ be the time, algorithm takes on instance I

Best case runtime: $$t_{best}(n) = \text{MIN}_{I:|I|=n} \left\{ T(I) \right\}$$

Worst case runtime: $$t_{worst}(n) = \text{MAX}_{I:|I|=n} \left\{ T(I) \right\}$$

Average case runtime: $$t_{av}(n) = \text{AVERAGE}_{I:|I|=n} \left\{ T(I) \right\}$$

In general, we consider the worst case runtime

Adding two n digits integers

Input: Two n digits numbers A and B
Output: $A + B$

For *"small"* A and B

1: $C \leftarrow A + B$

- The algorithm is correct by definition of $+$ operator
- Runtime is one integer addition
- Can't really do better than that . . .

Adding two n digits integers

Input: Two n digits numbers A and B (n-digits arrays)
Output: $A + B$ ($n + 1$-digit array)

$A =$

6	5	4	3	2	1	0
4	6	9	2	7	5	8

$B =$

6	5	4	3	2	1	0
5	1	7	2	2	6	1

		1		1	1		
	4	6	9	2	7	5	8
+	5	1	7	2	2	6	1
	9	8	6	5	0	1	9

$c \leftarrow 0$
for $i = 0$ to $n - 1$ **do**
    $S[i] \leftarrow (A[i] + B[i] + c) \bmod 10$
    $c \leftarrow (A[i] + B[i] + c)/10$
$S[n] \leftarrow c$

- Correct?
- Runtime?

Adding two n digits integers

Input: Two n digits numbers A and B (n-digits arrays)
Output: $A + B$ ($n + 1$-digit array)

$c \leftarrow 0$ } 1 time

for $i = 0$ to $n - 1$ **do**
  $S[i] \leftarrow (A[i] + B[i] + c) \bmod 10$
  $c \leftarrow (A[i] + B[i] + c)/10$
} n times

$S[n] \leftarrow c$ } 1 time

$6n$ **single digit arithmetic operations**

Multiplying two n digits integers

Input: Two n digits numbers A and B (n-digits arrays)
Output: $A \times B$ ($2n+1$-digit array)

```
for i = 1 to n do
    c ← 0
    for j = 1 to n do
        Z[i][j + i − 1] ← (A[j] ∗ B[i] + c) mod 10
        c ← (A[j] ∗ B[i] + c)/10
    Z[i][i + n] ← c
carry ← 0
for i = 1 to 2n do
    sum ← carry
    for j = 1 to n do
        sum ← sum + Z[j][i]
    C[i] ← sum mod 10
    carry ← sum/10
C[2n + 1] ← carry
```

			7	5	8
		×	6	3	2
		1	5	1	6
	2	2	7	4	
4	5	4	8		
4	7	9	0	5	6

Ops: $8n^2 + 2n$ arithmetic ops.

Multiplying two n digits integers

Input: Two n digits numbers A and B (n-digits arrays)
Output: (integer) $C = A \times B$

Reformulate and apply distributive and associative laws

$$\Big(A[0]*10^0 + A[1]*10^1 + A[2]*10^2 + \dots\Big) \times \Big(B[0]*10^0 + B[1]*10^1 + B[2]*10^2 + \dots\Big)$$

$C \leftarrow 0$
for $i = 1$ to n **do**
  for $j = 1$ to n **do**
    $C \leftarrow C + 10^{i+j} \times A[i] * B[j]$

		×	7	5	8
			6	3	2
		–	–	–	–
	–	–	–	–	
–	–	–	–		
4	7	9	0	5	6

Ops: n^2 single digit multiplications + shifting (multiplying by 10^x)

Exponentiation

Input: Two integers, a and $n \geq 0$

Output: a^n

Problem Formulation

$$a^n = \underbrace{a \times a \times \ldots \times a}_{n \text{ times}}$$

$x \leftarrow 1$
for $i = 1$ to n **do**
  $x \leftarrow x * a$
return x

- Correct by definition
- Takes n multiplications

  ▷ integer multiplications

- Initializing x to a, saves one multiplication

  ▷ Careful! what if $n = 0$

Can we do better?

Exponentiation

Input: Two integers, a and $n \geq 0$

Output: a^n

Problem Formulation

$$a^n = \begin{cases} a * a^{n-1} & \text{if } n > 1 \\ a & \text{if } n = 1 \\ 1 & \text{if } n = 0 \end{cases}$$

function REC-EXP(a,n)
  if $n = 0$ **then return** 1
  else if $n = 1$ **then return** a
  else
    return $a *$ REC-EXP($a, n-1$)

- Correct by the above definition
- Number of operations?

  ▷ Number of recursive calls $\times$ Number of operations per call

Exponentiation

Input: Two integers, a and $n \geq 0$

Output: a^n

Problem Formulation

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n > 1 \text{ even} \\ a \cdot a^{n-1/2} \cdot a^{n-1/2} & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0 \end{cases}$$

```
function REP-SQ-EXP(a,n)
    if n = 0 then return 1
    else if n > 0 AND n is even then
        z ← REP-SQ-EXP(a, n/2)
        return z * z
    else
        z ← REP-SQ-EXP(a, n−1/2)
        return a * z * z
```

- Correctness
- Number of calls?
- operations per call?

Give a non-recursive implementation of repeated squaring based exponentiation. You can also use the binary expansion of n

Dot Product of two vectors

Input: Two n-dimensional vectors as arrays A and B

Output: $A \cdot B := \langle A, B \rangle := A[1]B[1] + \ldots + A[n]B[n] := \sum_{i=1}^{n} A[i]B[i]$

$$\overset{\mathbf{A}}{\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}} \cdot \overset{\mathbf{B}}{\begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}} = \sum_{i=1}^{n} a_i b_i$$

```
function DOT-PROD(A, B)
    s ← 0
    for i = i to n do
        s ← s + A[i] * B[i]
    return s
```

- **Correctness** follows from definition
- **Runtime** is n multiplications and $n - 1$ additions

▷ integer/real additions and multiplications

- At least n "operations" are required for reading the input

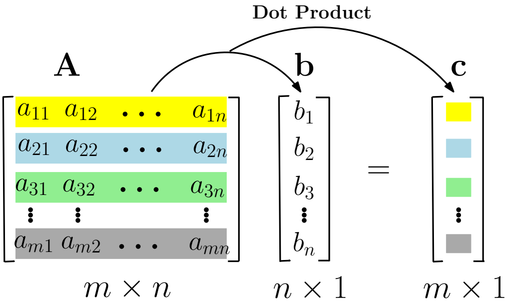
▷ Lower Bound

Matrix-Vector Multiplication

Input: Matrix A and vector b **Output:** $c = A * b$

- **Condition:** num columns of A = num rows of b

$$A_{m \times n} \times b_{n \times 1} = c_{m \times 1}$$



Matrix-Vector Multiplication

Input: Matrix A and vector b **Output:** $c = A * b$

function MAT-VECTPROD(A, b)
  $c[\,][\,] \leftarrow$ ZEROS($m \times 1$)
  for $i = 1$ to m **do**
    $c[i] \leftarrow$ DOT-PROD($A[i][:], b$)
    return c

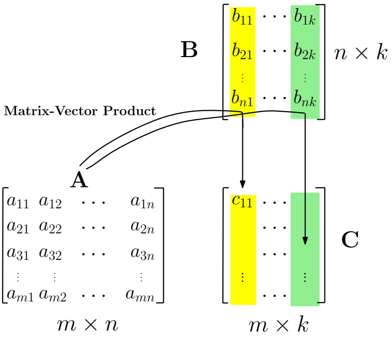
- **Correct** by definition
- **Runtime** is m dot-products of n-dim vectors
- Total runtime $m \times n$ real multiplications and additions

Matrix-Matrix Multiplication

Input: Matrices A and B **Output:** $C = A * B$

- **Condition:** num columns of A = num rows of B

$$A_{m\times n} \times B_{n\times k} = C_{m\times k}$$



Matrix-Matrix Multiplication

Input: Matrices A and B **Output:** $C = A * B$

- **Condition:** num columns of A = num rows of B

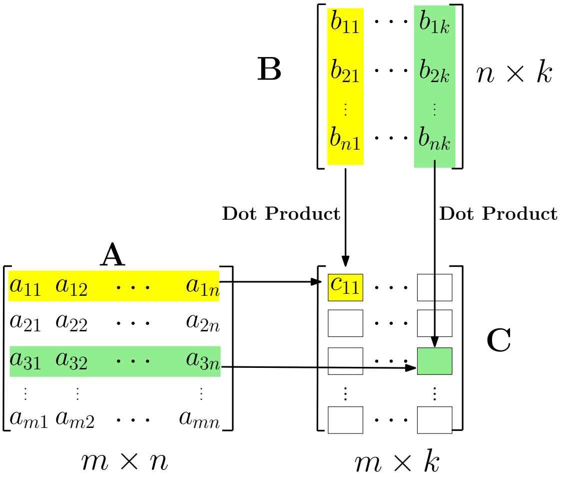
$$A_{m\times n} \times B_{n\times k} = C_{m\times k}$$

function MAT-MATPROD(A, B)
  $C[\,][\,] \leftarrow$ ZEROS($m \times k$)
  for $j = 1$ to k **do**
    $C[:][j] \leftarrow$ MAT-VECTPROD($A, B[:][j]$)
  return C

- k Matrix-Vector products of $m \times n$ and $n \times 1$
- Total $k \times m \times n$ real multiplications and additions

Matrix-Matrix Multiplication: Dot Product

Input: Matrices A and B **Output:** $C = A * B$



Input: Matrices A and B **Output:** $C = A * B$

- **Condition:** num columns of A = num rows of B

$$A_{m \times n} \times B_{n \times k} = C_{m \times k}$$

function MAT-MATPROD(A, B)
  $C[\,][\,] \leftarrow$ ZEROS($m \times k$)
  for $i = 1$ to m **do**
    for $j = 1$ to k **do**
      $C[i][j] \leftarrow$ DOT-PROD($A[i][:], B[:][j]$)
  return C

- Performs $m \times k$ dot-products of n-dim vectors
- Total $m \times k \times n$ real multiplications and additions

Summary

- Problem formulation with precise definitions/notation is important
- Definition-based (and other strategies) critically depend on it
- Pseudocode is a good human-readable way to describe solution
- Correctness of an algorithm is argued in view of problem statement
- Runtime of an algorithm is the most basic measure of its goodness
- Runtime is measured by number of well-chosen elementary operations as a function of size of input
- We usually consider the worst case runtime for a fixed input size
- Discussed how an algorithm can be used as a subroutine in another
- Gave different algorithms (for exponentiation) with different runtime
- Always ask if a solution can be improved (usually in terms of runtime)
- Lower bound means no algorithm has runtime lower than the bound