Algorithms

Lecture : Algorithmic Thinking and Arithmetic Problems

Imdad ullah Khan

Contents

1 Parity check of an integer 1
1.1 Parity Check: Formulation for large input 1
1.2 Parity Check: A Different Formulation 2

2 Algorithmic thinking and Terminology 2

3 Algorithms Design Questions 3
3.1 What is the problem? 3
3.2 Is the algorithm correct? 4
3.3 How much time does it take? 4
3.3.1 Runtime: 5
3.4 Can we improve the algorithm? 5

4 Analysis of Algorithms 5
4.1 Running Time 6
4.2 Elementary operations 6
4.3 Runtime as a function of input size 6
4.4 Best/Worst/Average Case 7
4.5 Growth of runtime as size of input 7

5 Arithmetic problems 7
5.1 Addition of two long integers 7
Correctness of Algorithm for adding two integers given as arrays . . . 8
Runtimeof Algorithm for adding two integers given as arrays 8
5.2 Multiplication of two long integers 9
Runtime of grade-school multiplication algorithm 10
5.2.1 A reformulation of the multiplication problem 11
5.3 Exponentiation of an integer to a power 11
5.3.1 Exponentiation by iterative multiplication 11
5.3.2 Exponentiation by recursive multiplication 12
Runtime of recursive exponentiation 12
5.3.3 Exponentiation by repeated squaring 13
Runtime of repeated squaring 13
5.4 Dot Product 14

Runtime of DOT-PROD 15
5.5 Matrix-Vector Multiplication 16
5.6 Matrix Multiplication via dot product 17
5.7 Matrix-Matrix Multiplication via Matrix-Vector Product 18

1 Parity check of an integer

We are going to discuss a very easy problem that should be trivial for everyone to solve. This will help us develop a thinking style of a computer scientist and establish a lot of terminology.

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

Solution: Here is a simple algorithm to solve this problem.

Algorithm 1 Parity-Test-with-mod

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

The above solution is written in pseudocode (more on it later). This algorithm solves the problem, however following are some issues with it.

- It 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/...?

Note that these issues are in addition to usual checks to see if it is a valid input, i.e. to check that A is not a real number, a string, or an image, etc.

1.1 Parity Check: Formulation for large input

Suppose the integer is very large, (n-digits long for some $n >$ the computer word length) so it does not fit any variable. Say the input integer is given as an array A, where each element is a digit of A (least significant digit at location 1 of the array). For example

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

In this case, the following algorithm solves the problem.

Solution:

By definition of even integers, we need to check if $A[0] \mod 2 = 0$. If so then output A is even.

Algorithm 2 Parity-Test-with-mod

```
Input: $A$ - digits array
Output: true if $A$ is even, otherwise false
  if $A[0] \bmod 2 = 0$ then
      return true
  else
      return false
```

1.2 Parity Check: A Different Formulation

What if the mod operator is not available in the programming language, then we can manually check if the last digit is an even digit, i.e. check if $A[0] \in \{0, 2, 4, 6, 8\}$.

Algorithm 3 Parity-Test-with-no-mod

```
Input: $A$ - digits array
Output: true if $A$ is even, otherwise false
  if $A[0] = 0$ then return true
  else if $A[0] = 2$ then return true
  else if $A[0] = 4$ then return true
  else if $A[0] = 6$ then return true
  else if $A[0] = 8$ then return true
  else
      return false
```

2 Algorithmic thinking and Terminology

We saw an algorithm for a very basic problem, next we discuss how a computer scientist ought to think about algorithms and what question one needs to ask about algorithms. We will then discuss some basic arithmetic problems and algorithms for them without elaborating explicitly about these questions, but they will be answered. Every students should be well versed with these algorithms, we discuss them to develop relevant vocabulary and terminology, such as elementary operations, runtime, correctness. We will also establish the pseudocode notation that we will use throughout this course.

An algorithm is defined as a procedure designed to solve a computational problem. When designing an algorithm, the designer needs to keep in mind some important features which are to be used when formulating an algorithm:

- **Input**: This is the data that is passed to the algorithm, and may be in different forms. For instance, it could be a simple number, or a list of numbers. Other examples include matrices, strings, graphs, images, videos, and many more. The format of input is an important factor in designing algorithms, since processing it can involve extensive computations.
- **Size of Input**: The size of input is a major factor in determining the effectiveness of an algorithm. Algorithms which may be efficient for small sizes of inputs may not do the same for large sizes.
- **Output**: This is the final result that the algorithm outputs. It is imperative that the result is in a form that can be interpreted in terms of the problem which is to be solved by the algorithm. For instance, it makes no sense to output 'Yes' or 'No' to a problem which asked to find the maximum number out of an array of numbers.
- **Pseudocode**: This is the language in which an algorithm is described. Note that when we say pseudocode, we mean to write down the steps of the algorithm using almost plain English in a manner that is understandable to a general user. Our focus will be the solution to the problem, without going into implementation details and technicalities of a programming language. Given our background, we will be using structure conventions of $C/C++/Java$. We will see many examples of this later on in the notes.

3 Algorithms Design Questions

With these features in mind, an algorithm designer seeks to answer the following questions regarding any algorithm:

3.1 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*"?

We saw in the parity test problem (when the input was assumed to be an integer that fits the word size), how an ill-formulated problem (ambiguous requirement specification) could cause problem. Similarly, when we had to consider whether or not mod is available in our toolbox.

As an algorithm designer, you should never jump to solution and must spend considerable amount time on formulating the problem, rephrasing it over and over, going over some special cases.

Formulating the problem with precise notation and definitions often yield a good ideas towards a solution. e.g. both the above algorithms just use definitions of even numbers

This is *implementing the definition* algorithm design paradigm, one can also call it a brute-force solution, though this term is often used for search problem.

One way that I find very useful in designing algorithms to not look for a smart solution right away. I instead ask, what is the dumbest/obvious/laziest way to solve the problem? What are the easiest cases? and what are the hardest cases? where the hardness come from when going from easy cases to hard cases?

3.2 Is the algorithm correct?

- Does it solve the problem? Does it do what it is supposed to do?
- Does it work in every case? Does it always "produce" the correct output? (also defined as the correctness of the algorithm)
- Does the algorithm halts on every input (is there a possibility that the algorithm has an infinite loop, in which case technically it shouldn't be called an algorithm though). Must consider "all legal inputs"

The correctness of the three algorithms we discussed for the parity-test problem follows from definition of even/odd and/or mod, depending on how we formulate the problem

3.3 How much time does it take?

- Actual clock-time depends on the actual input, architecture, compiler etc.
- What is the size of input?
- What are elementary operations? How many elementary operations of each type does it perform?
- How the number of operation depends on the "size" of input
- The answer usually is a function mapping the size of input to number of operations, in the worst case.
- We discuss it in more detail in the following section

3.3.1 Runtime:

of the algorithms we discuused so far. The first algorithm performs one mod operation over an integer and one comparison (with 0). The second algorithm performs one mod operation over a single-digit integer (recall size of the input) and one comparison (with 0) The third algorithm performs a certain number of comparisons of single digit integers. The actual number of comparison depends on the input. If A is even and its last digit is 0, then it takes only one comparison, if $A[0] = 2$, then we first compare $A[0]$ with 0 and then with 2, hence there are two comparisons. Similarly if A is odd or $A[0] = 8$, then it performs five comparisons.

This gives rise to the concept of best-case, worst-case analysis. While the best-case analysis gives us an optimistic view of the efficiency of an algorithm, common sense dictates that we should take a pessimistic view of efficiency, since this allows us to ensure that we can do no worse than worst-case, hence our focus will be on worst-case analysis.

Note that the runtimes of these algorithms do not depend on the input A, as it should not, as discussed above we consider the worst case only. But they do not even depend on the size of the input, we call this constant runtime, that is it stays constant if we increase the size of input, say we double the number of digits the algorithm still in the worst case performs 5 comparisons.

3.4 Can we improve the algorithm?

- Can we tweak the given algorithm to save some operations?
- Can we come up with another algorithm to solve the same problem?
- Computer scientists should never be satisfied unless ......
- May be we can't do better than something called the lower bound on the problem.

Since all three algorithms perform only a constant number of operation and we are usually concerned how can we **improve** the runtime as the input size grows, in this case we do not really worry about improving it.

4 Analysis of Algorithms

Analysis of algorithms is the theoretical study of performance and resource utilization of algorithms. We typically consider the efficiency/performance of an algorithm in terms of time it takes to produce output. We could also consider utilization of other resources such as memory, communication bandwidth etc. One could also consider various other factors such as user-friendliness, maintainability, stability, modularity, and security etc. In this course we will mainly be concerned with time efficiency of algorithm.

4.1 Running Time

This is the total time that the algorithm takes to provide the solution. The running time of an algorithm is a major factor in deciding the most efficient algorithm to solve a problem. There are many ways we could measure the running time, such as clock cycles taken to output, time taken (in seconds), or the number of lines executed in the pseudo-code. But these measures suffer from the fact that they are not constant over time, as in the case of clock cycles, which varies heavily across computing systems, or time taken(in seconds), which depends on machine/hardware, operating systems, other concurrent programs, implementation language, and programming style etc.

We need a **consistent mechanism** to measure running time of an algorithm. This runtime should be **independent of the platform** of actual implementation of the algorithm (such as actual computer architecture, operating system etc.) We need running time also to be **independent to actual programing language** used for implementing the algorithm.

Over the last few decades, **Moore's Law** has predicted the rise in computing power available in orders of magnitude, so processing that might have been unfeasible 20 years ago is trivial with today's computers. Hence, a more stable measure is required, which is the number of elementary operations that are executed, based on the size of input. We will define what constitutes an elementary operation below.

4.2 Elementary operations

These are the operations in the algorithm that are used in determining the running time of the algorithm. These are defined as the operations which constitute the bulk of processing in the algorithm. For instance, an algorithm that finds the maximum in a list of numbers by comparing each number with every other number in the list, will designate the actual comparison between two numbers (is $a < b$?) as the elementary operation. Generally, an elementary operation is any operation that involves processing of a significant magnitude.

4.3 Runtime as a function of input size

We want to measure runtime (number of elementary operations) as a function of size of input. Note that this does not depend on machine , operating system or language. **Size of input** is usually number of bits needed to encode the input instance, can be length of an array, number of nodes in a graph etc. It is important to decide which operations are counted as elementary, so keep this in mind while computing complexity of any algorithm. There is an underlying assumption that all elementary operation takes a constant amount of time.

4.4 Best/Worst/Average Case

For a fixed input size there could be different runtime depending on the actual instance. As we saw in the parity test of an integer. We are generally interested in the worst case behavior of an algorithm

Let $T(I)$ be time, algorithm takes on an instance I. The best case running time is defined to be the minimum value of $T(I)$ over all instances of the same size n.

Best case runtime:

$$t_{best}(n) = \text{MIN}_{I:|I|=n}\{T(I)\}$$

Worst case runtime:

$$t_{worst}(n) = \text{MAX}_{I:|I|=n}\{T(I)\}$$

Average case:

$$t_{av}(n) = \text{AVERAGE}_{I:|I|=n}\{T(I)\}$$

4.5 Growth of runtime as size of input

Apart from (generally) considering only the worst case runtime function of an algorithm. We more importantly, are interesting in

1. Runtime of an algorithm on large input sizes
2. Knowing how the growth of runtime with increasing input sizes. For example we want to know how the runtime changes when input size is doubled?

5 Arithmetic problems

Now that we have established the terminology and thinking styles, we look at some representative problems and see whether we can satisfy the questions with regards to the algorithm employed.

5.1 Addition of two long integers

Input Two long integers, given as arrays A and B each of length n as above.
Output An array S of length $n+1$ such that $S = A + B$.

This problem is very simple if A, B, and their sum are within word size of the given computer. But for larger n as given in the problem specification, we perform the grade-school digit by digit addition, taking care of carry etc.

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

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

$$\begin{array}{cccccccccc} & & & & \scriptstyle 1 & & \scriptstyle 1 & \scriptstyle 1 & & \\ & & 5 & 4 & 6 & 9 & 2 & 7 & 5 & 8 \\ + & & 8 & 5 & 1 & 7 & 2 & 2 & 6 & 1 \\ \hline & 1 & 3 & 9 & 8 & 6 & 5 & 0 & 1 & 9 \end{array}$$

In order to determine the tens digit and unit digit of a 2-digit number, we can employ the mod operator and the divide operator. To determine the units digit, we simply mod by 10. As for the tens digit, we divide by 10, truncating the decimal. We need this to determine the carry as we do manually. Can there be a better way to determine the carry in this case, think of what is the (largest) value of a carry when we add two 1-digit integer. We can determine if there should be a carry if a number is greater than 9, in that case we only have to extract the unit digit. We discussed in class that if the mod operator and type-casting is not available, then we can still separate digits in an integer by using the definition of positional number system.

The algorithm, with mod operator is as follows.

Algorithm 4 Adding two long integers

```
Input: $A, B$ - $n$-digits arrays of integers $A$ and $B$
Output: $S = A + B$
1: $c \leftarrow 0$
2: for $i = 0$ to $n-1$ do
3:     $S[i] \leftarrow (A[i] + B[i] + c) \bmod 10$
4:     $c \leftarrow (A[i] + B[i] + c)/10$
5: $S[n] \leftarrow c$
```

Correctness of Algorithm for adding two integers given as arrays The correction of this algorithm again follows from the definition of addition.

Runtimeof Algorithm for adding two integers given as arrays The running time (number of single digits) arithmetic operations performed by this algorithm is determined in details as follows, later on we will not go into so much detail. We count how many times each step is executed.

Algorithm 5 Adding two long integers

```
Input: $A, B$ - $n$-digits arrays of integers $A$ and $B$
Output: $S = A + B$
1: $c \leftarrow 0$                                  } 1 time
2: for $i = 0$ to $n-1$ do
3:     $S[i] \leftarrow (A[i] + B[i] + c) \bmod 10$  } $n$ times
4:     $c \leftarrow (A[i] + B[i] + c)/10$
5: $S[n] \leftarrow c$                              } 1 time
```

We do not count the memory operations, the algorithm clearly performs $4n$ 1-digit additions, n divisions and n mod operations. If we consider all arithmetic operations to be of the same complexity, then the runtime of this algorithm is $6n$. You should be able to modify the algorithm to be able to add to integers that do not have the same number of digits.

Can we improve this algorithm? Since any algorithm for adding n digits integers must perform some arithmetic on every digit, we cannot really improve upon this algorithm.

5.2 Multiplication of two long integers

Input Two long integers, given as array A and B each of length n as above.
Output An array C length $2n + 1$ such that $C = A \cdot B$.

We apply the grade-school multiplication algorithms, multiply A with the first digit of B, then with the second digit of B and so on, and adding all of these arrays. We use a $n \times n$ array Z (a 2-dimensional array or a matrix) to store the intermediate arrays. Of course, now we know how to add these arrays.

$$\begin{array}{cccccccc}
 & & & & 2 & 7 & 5 & 8 \\
 & & & \times & 9 & 6 & 3 & 2 \\
\cline{4-8}
 & & & & 5 & 5 & 1 & 6 \\
 & & & 8 & 2 & 7 & 4 & \\
 & 1 & 6 & 5 & 4 & 8 & & \\
2 & 4 & 8 & 2 & 2 & & & \\
\hline
2 & 6 & 5 & 6 & 5 & 0 & 5 & 6
\end{array}$$

Here again we will use the technique to determine the carry and the unit digits etc. Can we be sure that when we multiply two 1 digit integers, the result will only have at most 2 digits.

Algorithm 6 Multiplying two long integers

```
Input: $A, B$ - $n$-digits arrays of integers $A$ and $B$
Output: $C = A * B$
 1: for $i = 1$ to $n$ do
 2:     $c \leftarrow 0$
 3:     for $j = 1$ to $n$ do
 4:         $Z[i][j + i - 1] \leftarrow (A[j] * B[i] + c) \bmod 10$
 5:         $c \leftarrow (A[j] * B[i] + c)/10$
 6:     $Z[i][i + n] \leftarrow c$
 7: $carry \leftarrow 0$
 8: for $i = 1$ to $2n$ do
 9:     $sum \leftarrow carry$
10:     for $j = 1$ to $n$ do
11:         $sum \leftarrow sum + Z[j][i]$
12:     $C[i] \leftarrow sum \bmod 10$
13:     $carry \leftarrow sum/10$
14: $C[2n + 1] \leftarrow carry$
```

Runtime of grade-school multiplication algorithm

- The algorithm has two phases, in the first phase it computes the matrix, where we do all multiplications, and in the second phase it adds all elements of the matrix column-wise.
- In the first phases two for loops are nested each running for n iterations. You should know that by the product rule the body of these two nested loops is executed n^2 times. As in addition, the loop body has 6 arithmetic operations. So in total the first phase performs $6n^2$ operations.
- In the second phase, the outer loop iterates for $2n$ iterations which for each value of i the inner loop iterates n times (different value of j. While in the body of the nested loop there is one addition performed, so in total $2n^2$ additions. Furthermore, the outer loop (outside the nested loop) performs one *mod* and one divisions, so a total of $2n$ arithmetic.
- The grand total number of arithmetic operations performed is $6n^2+2n^2+2n = 8n^2+2n$ arithmetic operations.
- Question, when we double the size of input, (that is make n double of the previous), what happens to the number of operations, how do they grow. Draw a table of the number of operations for $n = 2, 4, 8, 16, 32$, etc.

5.2.1 A reformulation of the multiplication problem

In this section, we demonstrate how the multiplication problem can be reformulated and how it helps us solve the problem much simply.

Think of the integers A and B in positional number system and apply distributive and associative laws we get the following very simple algorithm.

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

The algorithm with this view of the problem is as follows

Algorithm 7 Multiplying two long integers using Distributive law of multiplication over addition

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

The correctness of this algorithm also follows from definition of multiplication and it performs n^2 single-digit multiplications and n^2 shifting (multiplication with powers of 10).

Can we improve these algorithms? When we study divide and conquer paradigm of algorithm design we will **improve** upon this algorithm.

5.3 Exponentiation of an integer to a power

Input Two integers a and $n \geq 0$.
Output An integer x such that $x = a^n$.

Again we apply the grade-school repeated multiplication algorithm, i.e. just execute the definition of a^n and multiply a n times. More precisely

$$x \;=\; a^n \;=\; \overbrace{a * a * \ldots * a * a}^{n \text{ times}}$$

5.3.1 Exponentiation by iterative multiplication

This way of looking at a^n leads us to the following algorithm.

Algorithm 8 Exponentiation

```
Input: $a, n$ - Integers $a$ and $n \geq 0$
Output: $a^n$
1: $x \leftarrow 1$
2: for $i = 1$ to $n$ do
3:     $x \leftarrow x * a$
4: return $x$
```

This algorithm clearly is **correct** and takes n multiplications. This time integer multiplications not 1-digit multiplications. We can tweak it to save one multiplication by initializing x to a, but be careful what if $n = 0$.

5.3.2 Exponentiation by recursive multiplication

Exponentiation can also be performed by a recursive algorithm, in case you recursively love recursion. We use this way of looking at exponentiating.

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

The algorithm implementing the above view of exponentiation is as follows.

Algorithm 9 Recursive Exponentiation

Input: a, n - Integers a and $n \geq 0$

Output: a^n

```
1: function REC-EXP($a$,$n$)
2:     if $n = 0$ then return 1
3:     else if $n = 1$ then return $a$
4:     else
5:         return $a * $ REC-EXP$(a, n-1)$
```

It's correctness can be proved with simple inductive reasoning.

Runtime of recursive exponentiation Its runtime is something, ok let me tell you this. Say its runtime is $T(n)$ when I input n, (we will discuss recurrences and their solution in more detail later). We have

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ T(n-1) & \text{if } n \geq 2 \end{cases}$$

5.3.3 Exponentiation by repeated squaring

We can improve by a lot by considering the following magic of power (or power of magic).

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

Note that when n is even $n/2$ is an integer and when n is odd, $(n-1)/2$ is an integer, so in both cases we get the same problem (exponentiating one integer to the power of another integer) but of smaller size. And smaller powers are supposed to be easy, really! well at least $n = 0$ or 1, or sometime even 2. So we exploit this formula and use recursion.

Algorithm 10 Exponentiation by repeated squaring

```
Input: $a, ns$ - Integers $a$ and $n \geq 0$
Output: $a^n$
1: function REP-SQ-EXP($a$,$n$)
2:     if $n = 0$ then return 1
3:     else if $n > 0$ AND $n$ is even then
4:         $z \leftarrow$ REP-SQ-EXPP($a, n/2$)
5:         return $z * z$
6:     else
7:         $z \leftarrow$ REP-SQ-EXP($a, (n-1)/2$)
8:         return $a * z * z$
```

Again correctness of this algorithm follows form the above formula. But you need to prove it using induction, i.e. prove that this algorithm returns a^n for all positive integers n, prove the base case using the stopping condition of this function etc.

Runtime of repeated squaring It is not very straight-forward to find the runtime of a recursive procedure always, in this case it is quite easy. Usually the runtime of a recursive algorithm is expressed as a recurrence relation, (that is the time algorithm takes for an input of size n is defined in terms of the time it takes on inputs of smaller size). In this case we know that the runtime doesn't really depends on the size of a (well not directly, we may assume that a is restricted to be an int. Multiplying y a few times with itself is just like multiplying z a few times with itself, the only thing that matters is how many times you do the multiplication).

In this case denote the runtime of REP-SQ-EXP on input a, n as $R(n)$ (as discussed above a doesn't have to be involved in determining runtime of REC-EXP). So we have to determine $R(n)$ as a function of n. What we know just by inspecting the algorithm is summarized in the following recurrence relation.

$$R(n) = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ R(n/2) + 2 & \text{if } n > 1 \text{ and } n \text{ is even} \\ R(n-1/2) + 3 & \text{if } n > 1 \text{ and } n \text{ is odd} \end{cases}$$

We will discuss recurrence relation later, but for now if you think about it as follows: Assume n is a power of 2 so it stays even when halve it.

$$R(n) = R(n/2) + 2 = R(n/4) + 2 + 2 = R(n/8) + 2 + 2 + 2 = \dots$$

In general we have

$$R(n) = R(n/2^j) + \overbrace{2 + 2 + \ldots + 2}^{j \text{ times}}$$

when $j = \log n$, then we get

$$\begin{aligned} R(n) &= R\left(n/2^{\log n}\right) + \overbrace{2 + 2 + \ldots + 2}^{\log n \text{ times}} \\ &= R\left(n/n\right) + \overbrace{2 + 2 + \ldots + 2}^{\log n \text{ times}} \\ &= R\left(1\right) + \overbrace{2 + 2 + \ldots + 2}^{\log n \text{ times}} = 1 + 2\log n \end{aligned}$$

It is very easy to argue that in general that is for n not necessarily power of 2 or even, the runtime is something like $1 + 3\log n$.

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

5.4 Dot Product

Dot Product is an operation defined for two n dimensional vectors. Dot product of two vectors $A = (a_1, a_2, \ldots, a_n)$, $B = (b_1, b_2, \ldots, b_n)$ is defined as

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

Input: Two n-dimensional vectors as arrays A and B
Output: $A \cdot B := \langle A, B \rangle := A[1]B[1] + \ldots + A[n]B[n] := \sum\limits_{i=1}^{n} A[i]B[i]$

Dot product is also commonly called an inner product or a scalar product. The a geometric interpretation for dot product is the following. If u is a unit vector and v is any vector then $v \cdot u$ is the projection of v onto u. The projection of any point p on v onto u is the point on u closest to p. If v and u are both unit vectors then $v \cdot u$ is the cos of the angle between the two vectors. So in a way the dot product between two vectors measures their similarity. It tells us how much of v is in the direction of u.

What we're interested in is how to compute the dot product. We need to multiply all the corresponding elements and then sum them up. So we can run a for loop, keep a running sum and at the i_{th} turn add the product of the i_{th} terms to it. The exact code is given below.

Algorithm 11 Dot product of two vectors

Input: A, B - n dimensional vectors as arrays of length n
Output: $s = A \cdot B$

```
1: function DOT-PROD($A$, $B$)
2:     $s \leftarrow 0$
3:     for $i = i$ to $n$ do
4:         $s \leftarrow s + A[i] * B[i]$
5:     return $s$
```

Runtime of dot-prod How much time does this take? Well that depends on a lot of factors. What machine you're running the program on. How many other programs are being run at the same time. What operating system you're using. And many other things. So just asking how much time a program takes to terminate doesn't really tell us much. So instead we'll ask a different question. How many elementary operations does this program performs. Elementary operations are things such as adding or multiplying two 32 bit numbers, comparing two 32 bit numbers, swapping elements of an array etc. In particular we're interested in how the number of elementary operations performed grows with the size of the input.This captures the efficiency of the algorithm much better.

So with that in mind, let's analyze this algorithm. What's the size of the input? A dot product is always between two vectors. What can change however is the size of these vectors. So that's our input size. So suppose the vectors are both n dimensional. Then this code performs n additions and n multiplications. So a total of $2n$ elementary operations are performed. Can we do any better? Maybe for particular cases we can, when a vector contains a lot of zeros. But in the general case probably this is as efficient as we can get.

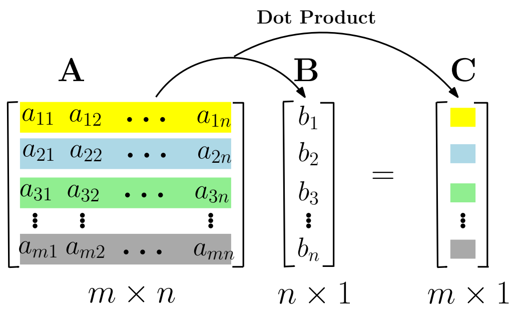
5.5 Matrix-Vector Multiplication

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 only for the case when the number of columns in matrix A equals the number of rows in vector x. So, if A is an $m \times n$ matrix (i.e., with n columns), then the product Ax is defined for $n \times 1$ column vectors x. If we let $Ax = b$, then b is an $m \times 1$ column vector. Matrix vector multiplication should be known to everyone, it is explained in the following diagram


Dot Product
A
B
C
a_{11} a_{12} $\cdots$ a_{1n}
a_{21} a_{22} $\cdots$ a_{2n}
a_{31} a_{32} $\cdots$ a_{3n}
a_{m1} a_{m2} $\cdots$ a_{mn}
b_1
b_2
b_3
b_n
=
$m \times n$
$n \times 1$
$m \times 1$

Algorithm for Matrix vector multiplication is given as follows.

Algorithm 12 Matrix Vector Multiplication

Input: A, B - $m \times n$ matrix and $n \times 1$ vector respectively given as arrays of appropriate lengths

Output: $C = A \times B$

```
1: function MAT-VECTPROD($A$, $B$)
2:     $C[\,][\,] \leftarrow$ ZEROS($m \times 1$)
3:     for $i = 1$ to $m$ do
4:         $C[i] \leftarrow$ 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

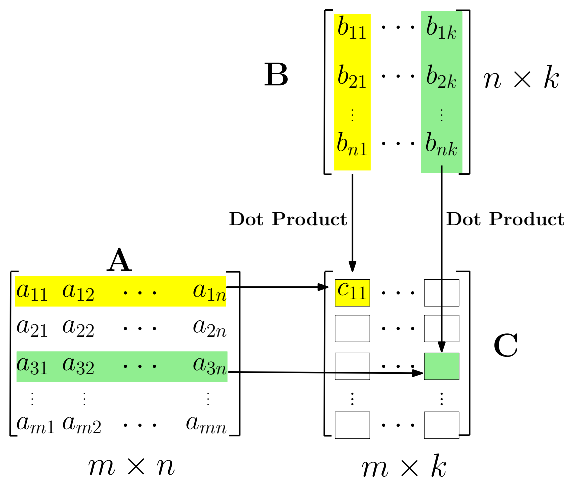
5.6 Matrix Multiplication via dot product

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

If A and B are two matrices of dimensions $m \times n$ and $n \times k$, then their product is another matrix C of dimensions $m \times k$ such that the $(i,j)_{th}$ entry of C is the dot product of the i_{th} row of A with the j_{th} column of B. That is

$$(C)_{ij} = A_i \cdot B_j.$$

This is explained in the following diagram



We know how to compute the dot product of two vectors so we can use that for matrix multiplication. The code is as follows

Algorithm 13 Matrix Matrix Multiplication

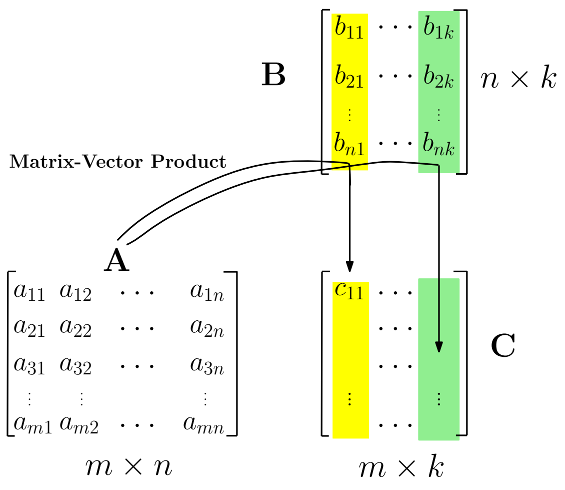
Input: A, B - $m \times n$ and $n \times k$ matrices respectively given as arrays of appropriate lengths
Output: $C = A \times B$

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

Analysis : How many elementary operations are performed in this algorithm? i goes from 1 to m and for each value of i, j goes from 1 to k. So the inner loop runs a total of mk times. Each time the inner loop runs we compute a dot product of two n dimensional vectors. Computing the dot product takes of two n dimensional vectors takes $2n$ operations, so the algorithm uses a total of $mk * 2n = 2mnk$ operations. Can we do any better for this problem? We defined matrix multiplication in terms of dot products and we said $2n$ is the minimum number of operations needed for a dot product, so it would seem we can't. But in fact there is a better algorithm for this. For those interested, you can look up Strassen's Algorithm for matrix multiplication.

5.7 Matrix-Matrix Multiplication via Matrix-Vector Product

Matrix matrix multiplication (of appropriate dimensions) can be achieved equivalently through repeated MATRIX-VECTOR-MULTIPLICATION. The process is explained in the following figure followed by its pseudo code.


B
b_{11} $\cdots$ b_{1k}
b_{21} $\cdots$ b_{2k}
b_{n1} $\cdots$ b_{nk}
$n \times k$
Matrix-Vector Product
A
a_{11} a_{12} $\cdots$ a_{1n}
a_{21} a_{22} $\cdots$ a_{2n}
a_{31} a_{32} $\cdots$ a_{3n}
a_{m1} a_{m2} $\cdots$ a_{mn}
$m \times n$
c_{11}
C
$m \times k$

Algorithm 14 Matrix Matrix Multiplication

Input: A, B - $m \times n$ and $n \times k$ matrices respectively given as arrays of appropriate lengths
Output: $C = A \times B$

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

Runtime of this algorithm is k multiplication of dimension $m \times n$ matrix with vectors of dimension $n \times 1$. Each matrix vector multiplication takes mn times as discussed above, so total runtime of this algorithm is kmn.