Introduction to Algorithms

(Chapter 1)

Algorithm origin

The word “algorithm "derived from the name of Persian mathematician Abdallāh Muḥammad ibn Mūsā al-Khwārizmī

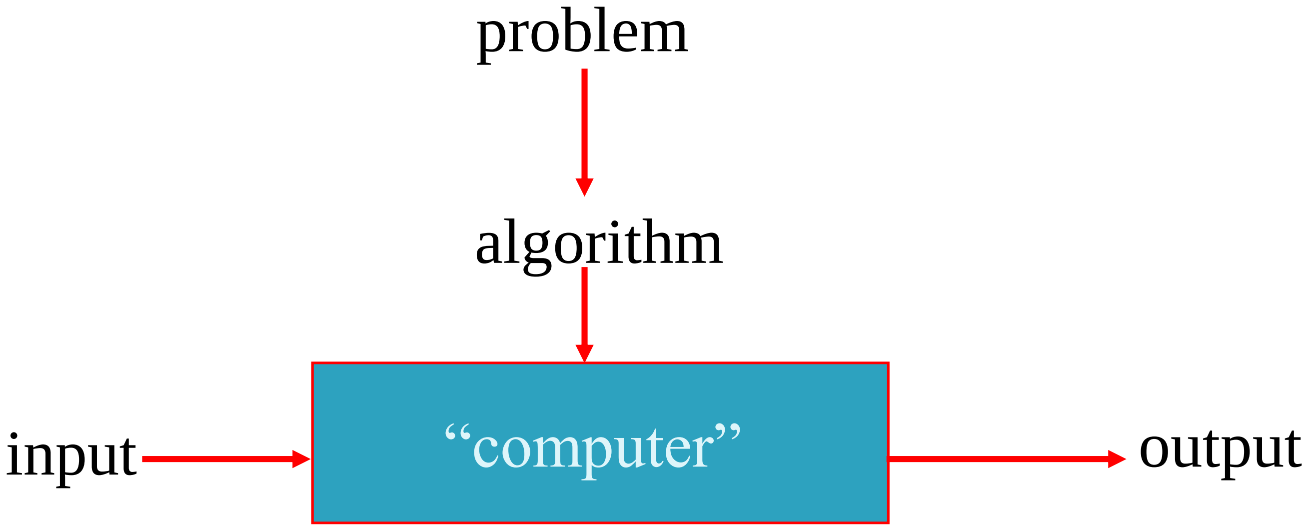
(The word algebra also comes from this name)

What is an Algorithm?

- One definition:

  *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***



Key points

- Each step is precise
- There can be more than one algorithm for the same problem

Example

- Here is a pseudocode algorithm:

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

- What does it do?

It finds the largest element of an array

Time Efficiency

- Is *find* a time-efficient algorithm?

- Seems good
  - To find the largest, you need to check each array element exactly once

```
Algo: find( A[0…n-1] )
    m ← A[0]
    for i ← 1 to n-1 do
         if A[i] > m
             m ← A[i]
    return m
```

Space Efficiency

- Is *find* a space-efficient algorithm? (amount of memory)

- Again… it seems reasonable
  - One temp variable introduced

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

Example

- How about sorted array?
- Is *find* efficient for sorted array?

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
           m ← A[i]
   return m
```

Why do we care?

- Think about computing the n^{th} Fibonacci number:
  - 0, 1, 1, 2, 3, 5, 8, 13, …

First algorithm

```
Algo: fib( n )
   if n ≤ 1
      return n
   else
      return fib( n-1 ) + fib( n-2 )
```

Java implementation

```
public static int fib(int n)  {
   if (n<=1)
      return n;
   else
      return ( fib(n-1) + fib(n-2) );
}
```

Why do we care, Part 2

- Now look at a different algorithm

Second algorithm

```
Algo: fib2( n )
   F[0] ← 0; F[1] ← 1;
   for i ← 2 to n do
      F[i] ← F[i-1] + F[i-2]
   return F[n]
```

```
public static int fib2(int n) {

   int[] f = new int[n+1];

   f[0] = 0;
   f[1] = 1;
   for (int i=2; i<=n; i++)
       f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

Difference

- First approach
  - Recursively calls the Fib function over and over again
- Second approach
  - Stores successive results so we don't have to re-compute them ...
- Second approach is much much faster.
  - For n = 30
  - Running time of first approach = 5957 microsecond
  - Running time of second approach = 7 microsecond

So?

- Fib is a basic example of why we care about algorithm efficiency
- A well thought out algorithm can run much faster
- There can be big variation in efficiency

How to Determine Efficiency

- Could do it experimentally
  - i.e. Write a bunch of implementations, see which one is fastest
- Problem?
  - Time consuming and expensive
  - It is not accurate
- Want to estimate efficiency before writing code

How to Determine Efficiency

- What we know:

1. running time (efficiency) of an algorithm depends on the input size
2. The total execution time for any algorithm depends on number of instructions executed

Example

- Remember this algorithm:

```
1. Algo: find( A[0…n-1] )
2.     m ← A[0]
3.     for i ← 1 to n-1 do
4.         if A[i] > m
5.             m ← A[i]
6.     return m
```

for n=3

stmt	#times
1	0
2	1
3	2
4	2
5	2
6	1

- How many instructions are executed if n=3?
  - f(3) =1 + 3*(3−1)+1

Example

- Remember this algorithm:

```
1. Algo: find( A[0…n-1] )
2.     m ← A[0]
3.     for i ← 1 to n-1 do
4.         if A[i] > m
5.             m ← A[i]
6.     return m
```

for n=8

stmt	#times
1	0
2	1
3	7
4	7
5	7
6	1

- What about n=8?
  - f(8) =1 + 3*(8-1)+1

- For input size n, then running time is
  $f(n) = 1 + 3*(n-1) + 1$

Basic Instructions

- Which instruction in *find* gets executed the most?

```
1. Algo: find( A[0…n-1] )
2.      m ← A[0]
3.      for i ← 1 to n-1 do
4.          if A[i] > m
5.              m ← A[i]
6.      return m
```

	(n=3)	(n=10)	(n=100)
stmt	#times	#times	#times
1	0	0	0
2	1	1	1
3	2	9	99
4	2	9	99
5	2	9	99
6	1	1	1

- We define the basic operation of an algorithm as the statement that gets executed most frequently

Basic Operations

This is the fundamental concept we use to analyze algorithmic efficiency:

count the number of basic operation
executed for an input of size n

- Using this idea, we would say for *find*
  - $f(n) = n-1$
- Because we don't count instructions that are not basic operations

Example 1

- Consider this algorithm:

```
1. Mystery1(n)  // n > 0
2.   S ← 0
3.   for i ← 1 to n do
4.       S ← S + i * i
5.   return S
```

1. What does this algorithm do?

   Calculates: $1^2 + 2^2 + 3^2 + \ldots + n^2$

2. What is the basic operation? Multiplication on line 4
   (or addition… doesn't matter)

3. How many times is the basic operation executed for input size n?

How many times?

```
Mystery(n)   // n > 0
  S ← 0
  for i ← 1 to n do
      S ← S + i * i
  return S
```

- Count operations each time in loop
  - 1^{st} time: 1 op,
  - 2^{nd} time: 1 op, …
  - n^{th} time: 1 op
- So you have a sum $\sum_{i=1}^{n} 1$
- What does this equal?
  - 1 + 1 + 1 … + 1 (n times)
  - = n
  - This sum is also in appendix A

Example 2

- Consider this algorithm:

```
1. Mystery2(A[0..n-1][0..n-1])  // n > 0
2.   S ← 0
3.   for i ← 0 to n-1 do
4.       for j← 0 to n-1 do
5.           S ← S + A[i][j];
6.   return S
```

1. What does this algorithm do? Calculates sum of the elements in array A
2. What is the basic operation? Addition on line 5
3. How many times is the basic operation executed for input size n?

Example 2

```
Mystery2(A[0..n-1][0..n-1])  // n > 0
  S ← 0
  for i ← 0 to n-1 do
     for j← 0 to n-1 do
          S ← S + A[i][j];
  return S
```

- The outer loop
  - i goes from 0 to n-1
  - So we have

$$\sum_{i=0}^{n-1} something$$

Example 2

```
Mystery2(A[0..n-1][0..n-1])  // n > 0
  S ← 0
  for i ← 0 to n-1 do
     for j← 0 to n-1 do
          S ← S + A[i][j];
  return S
```

- The inner loop:
  - j goes from 0 to n−1
  - At each iteration, we do one basic operation
  - So we have

$$\sum_{j=0}^{n-1} 1$$

  - We do this for each iteration of the outer loop

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1$$

Simplifying the Sum

- We know:

$$\sum_{j=0}^{n-1} 1 = 1 + 1 + \ ... + 1 = n$$

- Which equals:

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1 = \sum_{i=0}^{n-1} n = n + n + \ ... + n = n^2$$

Example 3

```
Loops(A[0..n-1])
 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
```

- What does this algorithm do?
- What is the basic operation?
- How many times is the operation executed for input n?

What Does it Do?

```
Loops(A[0..n-1])
 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
```

5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

Basic Operation

```
Loops(A[0..n-1])
 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
```

- Two options:
  - There are variable assignments and comparisons
  - Most people would say the basic operation is the key comparison *A[j]>v*
  - Why?
    - It is really the key thing being checked in each loop

The Example 3

```
Loops(A[0..n-1])
 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
```

- Look at outer loop first

- There is a variable i getting incremented from 1 up to $n-1$

- So… we have: $\sum_{i=1}^{n-1}(something)$

The Example 3

```
Loops(A[0..n-1])
  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
```

- The inner loop:
  - j goes from 0 to i–1
  - At each iteration, we do one basic operation
  - Mathematically, the number of steps is:

$$\sum_{j=0}^{i-1} 1$$

- We do this for each iteration of the outer loop
- So the total number of basic operations is:

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

Simplifying the Sum

- We know: $\sum_{j=0}^{i-1} 1 = i$
- So: $\sum_{i=1}^{n-1}\sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i$
- Which equals: $\frac{(n-1)n}{2}$

(we just showed this… and it is in appendix A)

Two Main Issues in this Course

- How to design algorithms

- How to analyze algorithm efficiency
  - Time/space efficiency

Algorithm Design Techniques

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Important Problem Types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Numerical problems

Try it/ Homework

- Chapter 1.1 page 8, question 5
- Chapter 1.2 page 18, question 9
- Chapter 1.3 page 23, question 1