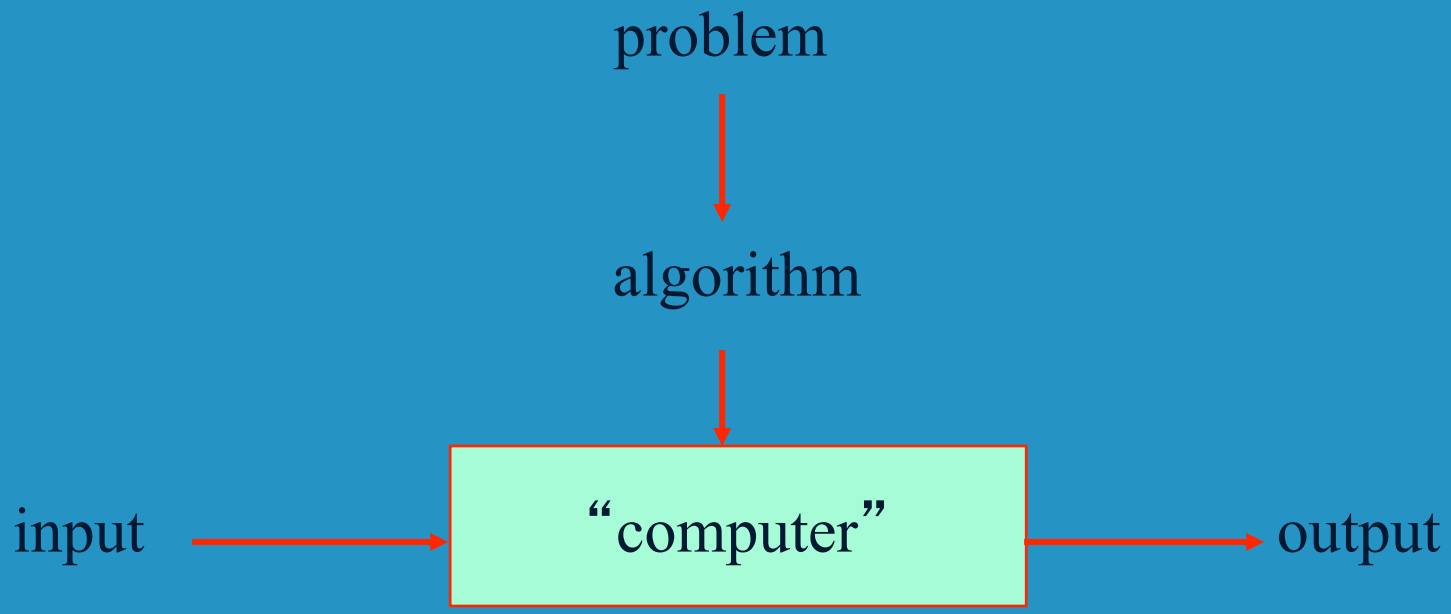
Chapter 1

Introduction

What is an 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.



Ingredients – the given info...

- **Pastry**
  - 2 cups Gold Medal™ all-purpose flour
  - 1 teaspoon salt
  - 2/3 cup plus 2 tablespoons shortening
  - 4 to 6 tablespoons cold water
- **Filling**
  - 1/3 to 1/2 cup sugar
  - 1/4 cup Gold Medal™ all-purpose flour
  - 1/2 teaspoon ground cinnamon
  - 1/2 teaspoon ground nutmeg
  - 1/8 teaspoon salt
  - 8 cups thinly sliced peeled tart apples (8 medium)
  - 2 tablespoons butter or margarine

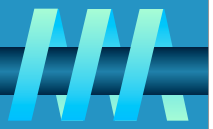
Directions – the algorithm...

- In medium bowl, mix 2 cups flour and 1 teaspoon salt. Cut in shortening, using pastry blender (or pulling 2 table knives through ingredients in opposite directions), until particles are size of small peas. Sprinkle with cold water, 1 tablespoon at a time, tossing with fork until all flour is moistened and pastry almost cleans side of bowl (1 to 2 teaspoons more water can be added if necessary).
- Gather pastry into a ball. Divide in half; shape into 2 flattened rounds on lightly floured surface. Wrap in plastic wrap; refrigerate about 45 minutes or until dough is firm and cold, yet pliable. This allows the shortening to become slightly firm, which helps make the baked pastry more flaky. If refrigerated longer, let pastry soften slightly before rolling.
- Heat oven to 425°F. With floured rolling pin, roll one pastry round into round 2 inches larger than upside-down 9-inch glass pie plate. Fold pastry into fourths; place in pie plate. Unfold and ease into plate, pressing firmly against bottom and side.
- In large bowl, mix sugar, 1/4 cup flour, the cinnamon, nutmeg and 1/8 teaspoon salt. Stir in apples until well mixed. Spoon into pastry-lined pie plate. Cut butter into small pieces; sprinkle over filling. Trim overhanging edge of pastry 1/2 inch from rim of plate.
- Roll other round of pastry into 10-inch round. Fold into fourths and cut slits so steam can escape. Unfold top pastry over filling; trim overhanging edge 1 inch from rim of plate. Fold and roll top edge under lower edge, pressing on rim to seal; flute as desired. Cover edge with 2- to 3-inch strip of foil to prevent excessive browning.
- Bake 40 to 50 minutes or until crust is brown and juice begins to bubble through slits in crust, removing foil for last 15 minutes of baking. Serve warm if desired.

The resulting output...

- from a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output (apple pie) for any legitimate input (ingredients) in a finite amount of time (40-50min of bake time + preparation time)

Greatest Common Divisor (GCD)



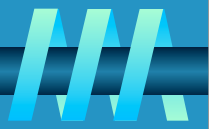
GCD function is used to get the greatest common divisor of two or more integers.

The greatest common divisor is the largest positive integer that divides the numbers without a remainder.

For example, The formula =GCD(60,36) returns the number 12, since 12 is the largest factor that goes into both numbers evenly.

{ 60 / 12 = 5 and 36 / 12 = 3 }

Euclid's Algorithm



Problem: Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

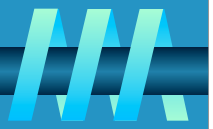
Euclid's algorithm is based on repeated application of equality

$$\gcd(m,n) = \gcd(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

gcd(0,0) defined



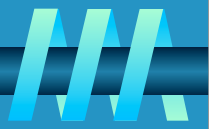
The word "greatest" in "Greatest Common Divisor" does not refer to being largest in the usual ordering of the natural numbers, but to being largest in the partial order of divisibility on the natural numbers, where we consider a to be larger than b only when b divides evenly into a. Most of the time, these two orderings agree whenever the second is defined. However, while, under the usual order, 0 is the smallest natural number, under the divisibility order, 0 is the greatest natural number, because every number divides 0.

Therefore, since every natural number is a common divisor of 0 and 0, and 0 is the greatest (in divisibility) of the natural numbers, gcd(0,0)=0.

how: what is gcd of 0 and 0

looking at: math.stackexchange.com/questions/495119/what-is-gcd0-0

Two descriptions of Euclid's algorithm



Problem: Find gcd(*m*,*n*)

Step 1 If $n = 0$, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

while $n \neq 0$ **do**

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

Other methods for computing gcd(*m*,*n*)

Consecutive integer checking algorithm

Step 1 Assign the value of min{*m*,*n*} to *t*

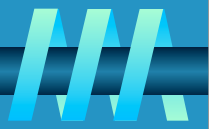
Step 2 Divide *m* by *t*. If the remainder is 0, go to Step 3; otherwise, go to Step 4

Step 3 Divide *n* by *t*. If the remainder is 0, return *t* and stop; otherwise, go to Step 4

Step 4 Decrease *t* by 1 and go to Step 2

Note: will not work correctly when one of the input numbers is zero (0)

Other methods for gcd(*m*,*n*) [cont.]



Middle-school procedure

Step 1 Find the prime factorization of *m*

Step 2 Find the prime factorization of *n*

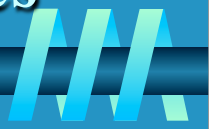
Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd*(m,n)*

Not a legitimate algorithm in this form...

why? prime factorization steps not defined unambiguously – as in how to find the primes list... (aka: exercise left to the reader)

Sieve of Eratosthenes: generate consecutive primes



Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ **to** n **do** $A[p] \leftarrow p$

for $p \leftarrow 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**

    if $A[p] \neq 0$ //p hasn't been previously eliminated from the list

        $j \leftarrow p * p$

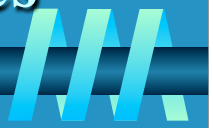
        while $j \leq n$ **do**

            $A[j] \leftarrow 0$ //mark element as eliminated

            $j \leftarrow j + p$

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Sieve of Eratosthenes: generate consecutive primes



Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ **to** n **do** $A[p] \leftarrow p$ end_for

for $p \leftarrow 2$ **to** $\lfloor \sqrt{n} \rfloor$ **do**

    if $A[p] \neq 0$ //p hasn't been previously eliminated from the list

        $j \leftarrow p * p$

        while $j \leq n$ **do**

            $A[j] \leftarrow 0$ //mark element as eliminated

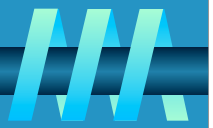
            $j \leftarrow j + p$

        end_while

end_for

			0		0		0	0	0		0		0	0	0		0	
Example:	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

An algorithm may be implemented (coded) in multiple ways...



sieve1

/* one array to store all the integers, and after I perform the sieve I move the remaining prime numbers into their own array. The program below will store and print the first 100,000 primes (you can adapt it easily for a larger list if you want).

www.programminglogic.com/the-sieve-of-eratosthenes-implemented-in-c/

note that their code directly does not work - they mixed lower and upper case variables and print issues

I also changed final output somewhat... */

sieve2

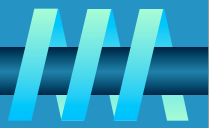
/* use a single array, fill it with 1s, and then put 0s on all the numbers that are not primes. The program below prints the first 650,000 or so primes using this method:

www.programminglogic.com/the-sieve-of-eratosthenes-implemented-in-c/

note that their code directly does not work - they mixed lower and upper case variables and print issues

I also changed final output somewhat... */

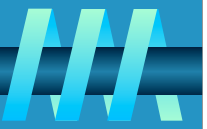
An algorithm may be implemented (coded) in multiple ways...



```c
#include <stdio.h>

#define LIMIT 1500000 /*size of integers array*/
#define PRIMES 100000 /*size of primes array*/

int main(){
   int i,j,numbers[LIMIT];
   int primes[PRIMES];

   /*fill the array with natural numbers*/
   for (i=0;i<LIMIT;i++){
      numbers[i]=i+2;
   }

   /*sieve the non-primes*/
   for (i=0;i<LIMIT;i++){
      if (numbers[i]!=-1){
         for (j=2*numbers[i]-2;j<LIMIT;j+=numbers[i])
            numbers[j]=-1;
      }
   }

   /*transfer the primes to their own array*/
   j = 0;
   for (i=0;i<LIMIT&&j<PRIMES;i++)
      if (numbers[i]!=-1)
         primes[j++] = numbers[i];

   /*print*/
   for (i=0;i<PRIMES;i++)
      printf("sieve1: %dth prime = %d\n",(i+1), primes[i]);

return 0;
}
```

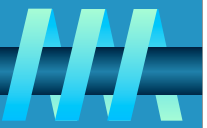
```c
#include <stdio.h>
#include <stdlib.h>

#define LIMIT 10000000 /*size of integers array*/

int main(){
    unsigned long long int i,j;
    int *primes;
    int z = 1;

    primes = malloc(sizeof(int)*LIMIT);

    for (i=2;i<LIMIT;i++)
        primes[i]=1;

    for (i=2;i<LIMIT;i++)
        if (primes[i])
            for (j=i;i*j<LIMIT;j++)
                primes[i*j]=0;

    for (i=2;i<LIMIT;i++)
        if (primes[i])
            printf("sieve2: %dth prime = %llu¥n",z++,i);

    return 0;
}
```

Why study algorithms?



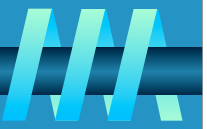
- Theoretical importance
  - the core of computer science
- Practical importance
  - A practitioner's toolkit of known algorithms
  - Framework for designing and analyzing algorithms for new problems
- Personal importance (to me...)
  - Algorithms – the puzzle in computer science

Two main issues related to algorithms



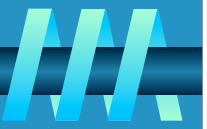
- **How to design algorithms**
  - how to solve the “puzzle”
  - problem solving strategies
  - number of paradigms
- **How to analyze algorithm efficiency**
  - what “efficiency”?

Analysis of algorithms



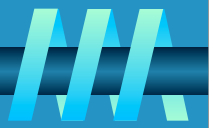
- How good is the algorithm?
  - time efficiency
  - space efficiency
  - energy efficiency
  - human efficiency
  - ? what exactly is the metric you are targeting here?
- Does there exist a better algorithm?
  - lower bounds
  - optimality
  - ? again, what exactly is the metric you are targeting here?

And yet another issue...



- How to describe or represent an algorithm
  - natural language (english, other)
  - code (which one)
  - pseudo code (what is that?)

Fundamentals of Algorithmic Problem Solving



- Understand the problem
  - Understand the desired result
- Know capabilities of computing device
  - Select best device given constraints
- Exact or Approximate problem solving
  - Again, desired results may dictate this
- Algorithm design techniques
  - A good programmer "reuses" prior art
- Design of algorithm and data structures
  - Other choices may drive this – i.e. architecture, device, etc.
- Methods of specifying an algorithm
  - English language is wonderfully ambiguous
- Proving an algorithm correctness
  - A real mathematical proof or proof by example / testing / etc.
- Analyzing an algorithm
  - For what?
- Code the algorithm
  - This should be the easy part by now...

Algorithm design techniques/strategies

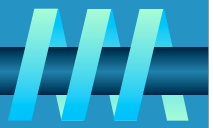
More General Techniques

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs

Less General Techniques

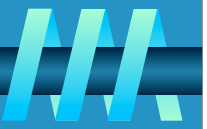
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Important problem types



- sorting
  - rearrange the items of a given list in a nondecreasing order.
- searching
  - find a given value (key)
- string processing (is a searching problem)
  - find a give string, of characters from alphabet, bit strings (0/1), gene sequence
- graph problems
  - connected points
- combinatorial problems
  - find combinatorial object – permutation, combination, subset – that satisfies certain constraints
- geometric problems
  - deal with geometric objects – points, lines, polygons
- numerical problems
  - mathematical objects of continuous nature such as solving equations and systems of equations, evaluating functions, etc.

Fundamental data structures



- list
  - array
  - linked list
  - string
- stack
- queue
- priority queue
- graph
- tree
- set and dictionary