Problem Solving And Algorithms

Motivation

Computing technology has changed—and is continuing to change—the world. Essentially every aspect of life has been impacted by computing. Computing related fields in almost all areas of study are emerging.

Various Computational-Related Fields		
Computational Biology	Computational Medicine	Computational Journalism
Computational Chemistry	Computational Pharmacology	Digital Humanities
Computational Physics	Computational Economics	Computational Creativity
Computational Mathematics	Computational Textiles	Computational Music
Computational Materials Science	Computational Architecture	Computational Photography
Computer-Aided Design	Computational Social Science	Computational Advertising
Computer-Aided Manufacturing	Computational Psychology	Computational Intelligence

What is Computer Science?

Computer science is fundamentally about computational problem solving.

Programming and computers are only tools in the field of computing. The field has tremendous breadth and diversity. Areas of study include:

- Programming Language Design
- Systems Programming
- Computer Architecture
- Human–Computer Interaction
- Robotics
- Artificial Intelligence
- Software Engineering
- Database Management / Data Mining
- Computer Networks
- Computer Graphics
- Computer Simulation
- Information Security

Computational Problem Solving

Two things that are needed to perform computational problem solving:

- **a representation** that captures all the relevant aspects of the problem
- **an algorithm** that solves the problem by use of the representation

Thus, computational problem solving finds a solution within a representation that translates into a solution for what is being represented.

The Use of Abstraction in Computational Problem Solving

A representation that leaves out detail of what is being represented is a form of abstraction.

Abstraction is prevalent in the everyday world. For example, **maps are abstract representations**.

Below is the original 1908 map of the London Underground (Subway).

Below is a more abstract, but topologically correct, map of the London Underground subway system. It shows the bends and curves of each track.

This map contains too much information for its purpose—that is, to find out where each subway line leads, and where the connections are between lines.

Below is a more abstract representation of the subway system, developed by Harry Beck in 1931. The track lines are straightened out where the track curves are irrelevant for subway riders. This is a simpler, easier to read, and thus a better representation for its purpose.

This particular abstraction is still in use today.

Washington D.C. Metro Map

Abstraction in Computing

Abstraction is intrinsic to computing and computational problem solving.

- The concept of “1s” and “0s” in digital computing is an abstraction

  digital information is actually represented as high or low voltage levels, magnetic particles oriented one of two ways, pits on an optical disk, etc.

- Programming languages are an abstraction

  the instructions and data of a computer program is an abstract representation of the underlying machine instructions and storage

- Programming design involves the use of abstraction

  programs are conceptualized as various modules that work together

Man, Cabbage, Goat, Wolf Problem

A man lives on the east side of a river. He wishes to bring a cabbage, a goat, and a wolf to a village on the west side of the river to sell. However, his boat is only big enough to hold himself, and either the cabbage, goat, or wolf. In addition, the man cannot leave the goat alone with the cabbage because the goat will eat the cabbage, and he cannot leave the wolf alone with the goat because the wolf will eat the goat. How does the man solve his problem?

There is a simple algorithmic approach for solving this problem by simply trying all possible combinations of items that may be rowed back and forth across the river.

Trying all possible solutions is referred to as a ***brute force approach***.

What would be an appropriate representation for this problem? Whatever representation we use, only the aspects of the problem that are relevant for its solution need to be represented.

- Color of the boat?
- Name of the man?
- Width of the river?

The only information relevant for this problem is where each particular item is at each step in the problem solving. Therefore, by the use of **abstraction**, we define a representation that captures only this needed information.

For example, we could use a sequence to indicate where each of the objects currently are,

man	cabbage	goat	wolf	boat	village
[east,	west,	east,	west,	east,	west]

where it is understood that the first item in the sequence is the location of the man, the second the location of the cabbage, etc.

Note that the village is always on the west side of the river—it doesn't move! Its location is fixed and therefore does not need to be represented.

Also, the boat is always in the same place as the man. So representing the location of both the man and the boat is redundant information. The relevant, **minimal representation** is given below,

man	cabbage	goat	wolf
[E ,	W ,	E ,	E]

The actual problem is to determine how the man can row objects across the river, with certain constraints on which pairs of objects cannot be left alone.

The computational problem is to find a way to convert the representation of the **start state** of the problem, when all the object are on the east side of the river,

man	cabbage	goat	wolf
[E ,	E ,	E ,	E]

to the **goal state** with all objects on the west side of the river,

man	cabbage	goat	wolf
[W ,	W ,	W ,	W]

with the constraint that certain **invalid states** should never be used.

Thus, in a computational problem solving approach, a problem is solved within the representation used, in which the solution within the representation must translate into a solution of the actual problem.

For example, from the start state, there are three possible moves that can be made, only one of which results in a valid state.

man cabbage goat wolf

[E , E , E , E] **START STATE**

[W , W , E , E]

Man rows cabbage across

INVALID STATE

Goat left alone with wolf

[W , E , W , E]

Man rows goat across

VALID STATE

Cabbage left alone with wolf

[W , E , E , W]

Man rows wolf across

INVALID STATE

Cabbage left alone with goat

We check if the new problem state is the goal state. If true, then we solved the problem in one step! (We know that cannot be so, but the algorithmic approach that we are using does not.)

man	cabbage	goat	wolf	
[E ,	E ,	E ,	E]	**START STATE**

⇩

[W , E , W , E]

Man rows goat across ✓

Is goal state [W , W , W , W] ? **No**

Therefore we continue searching from the current state.

Since the man can only row across objects on the same side of the river, there are only two possible moves from here,

man	cabbage	goat	wolf	
[W ,	E ,	W ,	E]	**INTERMEDIATE STATE**

[E , W , E , E]

Man rows back alone

✓

VALID STATE

Cabbage left alone with wolf

[E , E , E , E]

Man rows goat across

✓

VALID STATE

BUT, previously in this state. It is the start state. No progress made!

This would continue until the goal state is reached,

man cabbage goat wolf

[E , W , E , E]

⇩

.

.

⇩

[W , W , W , W] **GOAL STATE**

Thus, **the computational problem of generating the goal state from the start state translates into a solution of the actual problem since each transition between states has a corresponding action in the actual problem—**of the man rowing across the river with (or without) a particular object.

The Importance of Algorithms

MAY 2012

Sun	Mon	Tues	Wed	Thur	Fri	Sat
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30	31		

As another example computational problem, suppose that you needed to write a program that displays a calendar month for any given month and year.

The representation of this problem is rather straightforward. Only a few values are needed:

- **the month and year**
- **number of days in each month**
- **names of the days of the week**
- **day of the week that the first day of the month falls on**

The month and year, number of days in a month, names of the days of the week can be easily handled. **The less obvious part is how to determine the day of the week that a particular date falls on**.

How would you do that?

> *Start with a known day of the week for a given year in the past and calculate forward from there?*

That would not be a very efficient way of solving the problem.

Since calendars are based on cycles, there must be a more direct method for doing this. Thus, no matter how good a programmer you may be, without knowledge of the needed algorithm, you could not write a program that solves the problem.

Following is an example algorithm for determining the day of the week for any date between January 1, 1800 and December 31, 2099

To determine the day of the week for a given **month**, **day**, and **year**:

1. Let **century_digits** be equal to the first two digits of the year.
2. Let **year_digits** be equal to the last two digits of the year.
3. Let **value** be equal to **year_digits** + floor(**year_digits** / 4)
4. If **century_digits** equals 18, then add 2 to **value**, else
   if **century_digits** equals 20, then add 6 to **value**
5. If the **month** is equal to January and **year** is not a leap year,
   then add 1 to **value**, else,

   if the **month** is equal to February and the **year** is a leap year, then
   add 3 to **value**; if not a leap year, then add 4 to **value**, else,

   if the **month** is equal to March or November, then add 4 to **value**, else,

   if the **month** is equal to April or July, then add 0 to **value**, else,

   if the **month** is equal to May, then add 2 to **value**, else,

   if the **month** is equal to June, then add 5 to **value**, else,

   if the **month** is equal to August, then add 3 to **value**, else,

   if the **month** is equal to October, then add 1 to **value**, else,

   if the **month** is equal to September or December, then add 6 to **value**,
6. Set **value** equal to (**value** + **day**) mod 7.
7. If **value** is equal to 1, then the day of the week is Sunday; else

   if **value** is equal to 2, day of the week is Monday; else

   if **value** is equal to 3, day of the week is Tuesday; else

   if **value** is equal to 4, day of the week is Wednesday; else

   if **value** is equal to 5, day of the week is Thursday; else

   if **value** is equal to 6, day of the week is Friday; else

   if **value** is equal to 0, day of the week is Saturday

The Limits of Computational Problem Solving

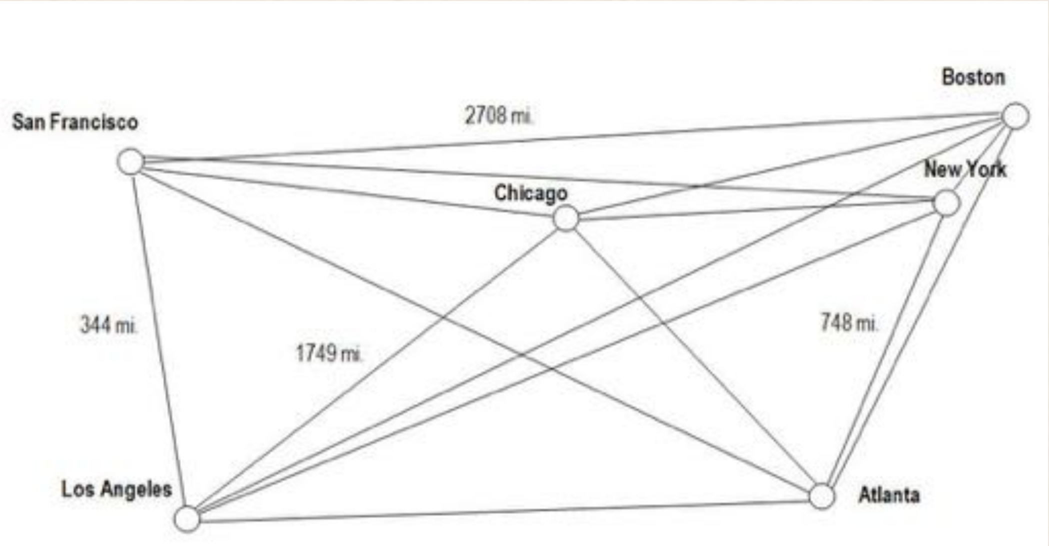
Once an algorithm for a given problem is developed or found, an important question is "**Can a solution to the problem be found in a reasonable amount of time**?"

"But aren't computers very fast, and getting faster all the time?"

Yes, but some problems require an amount of time to compute a solution that is astronomical compared to the capabilities of current computing devices.

A classic problem in computer science that demonstrates this is the **Traveling Salesman problem**.

The Traveling Salesman Problem



A salesman needs to visit a set of cities. He wants to find the shortest route of travel, starting and ending at any city for a given set of cities, what route should he take?

The algorithm for solving this problem is a simple one. Determine the lengths of all possible routes that can be taken, and find the shortest one. That is, by using a **brute force approach**. **The computational issue, therefore, is for a given set of cities, how many possible routes are there?**

If we consider a route to be a specific sequence of names of cities, then **how many permutations of that list are there?**

New York, Boston, Chicago, San Francisco, Los Angeles, Atlanta

New York, Boston, Chicago, San Francisco, Atlanta, Loa Angeles

New York, Boston, Chicago, Los Angeles, San Francisco, Atlanta

etc.

Mathematically, **the number of permutations for n entities is n!** (n factorial).

How big a number is that for various number of cities?

Below are the number of permutations (and this number of routes) there are for varies numbers of cities:

Ten Cities	10!	3,628, 800 (over three million)
Twenty Cities	20!	2,432,902,008,176,640,000
Fifty Cities	50!	Over 10^{64}

If we assume that a computer could compute one million routes per second:

- for **twenty cities**, it would take **77,000 years**
- for **fifty cities**, it would take **longer than the age of the universe!**

The Game of Chess

When a computer plays chess against a human opponent, both have to "think ahead" to the possible outcomes of each move it may make.

A brute force approach can also be used for a computer playing a game of chess. A program can consider all the possible moves that can be made, each ending in a win, loss, or draw. It can then select the move that leads to the most number of ways of winning. (Chess masters, on the other hand, only think ahead a few moves, and "instinctively" know the value of each outcome.)

There are approximately 10^{120} possible chess games that can be played. This is related to the average number of look-ahead steps needed for deciding each move.

There are approximately,

10^{80} atoms in the observable universe

and an estimated

3×10^{90} grains of sand to fill the universe solid

Thus, there are *more possible chess games that can be played than grains of sand to fill the universe solid!*

Therefore, for problems such as this and the Traveling Salesman problem in which a brute-force approach is impractical to use, clever and more efficient problem-solving methods must be discovered that find either an exact or an approximate solution to the problem.

Computer Algorithms

An **algorithm** is a finite number of clearly described, unambiguous "doable" steps that can be systematically followed to produce a desired result for given input in a finite amount of time (that is, it eventually terminates).

The word "algorithm" is derived from the ninth-century Arab mathematician, **Al-Khwarizmi** who worked on "written processes to achieve some goal."

Algorithms and Computers: A Perfect Match

Computer algorithms are central to computer science. They provide step-by-step methods of computation that a machine can carry out.

Having high-speed machines (computers) that can consistently follow a given set of instructions provides a reliable and effective means of realizing computation. However, **the computation that a given computer performs is only as good as the underlying algorithm used.**

Because **computers can execute a large number of instructions very quickly and reliably without error**, algorithms and computers are a perfect match!

Euclid's Algorithm
One of the Oldest Known Algorithms

An algorithm for computing the greatest common denominator (GCD) of two given integers. It is one of the oldest numerical algorithms still in common use.

1. Assign M the larger of the two values and N the smaller.

2. Divide M by N, call the remainder R.

3. If R is not 0, then assign M the value of N, assign N the value of R, and go to step 2.

4. The greatest common divisor is N.

Example Use

Finding the GCD of 18 and 20

1. Assign M the value of the larger of the two values, and N the smaller.
   M $\leftarrow$ 20   N $\leftarrow$ 18

Example Use

Finding the GCD of 18 and 20

1. Assign M the value of the larger of the two values, and N the smaller.
   $M \leftarrow 20 \quad N \leftarrow 18$

2. Divide M by N, call the remainder R.
   $M/N = 20 / 18 = 1$, with $R \leftarrow 2$

Example Use

Finding the GCD of 18 and 20 (**first time** through, **second time** through)

1. Assign M the value of the larger of the two values, and N the smaller.
   M ← 20   N ← 18

2. Divide M by N, call the remainder R.
   M/N = 20 / 18 = 1, with R ← 2
   M/N = 18 / 2 = 9, with R ← 0

3. If R is not 0, assign M the value of N, assign N the value of R, and go to step 2.
   R = 2. Therefore, M ← 18, N ← 2. Go to step 2.

Example Use

Finding the GCD of 18 and 20 (**first time** through, **second time** through)

1. Assign M the value of the larger of the two values, and N the smaller.
   M ← 20   N ← 18

2. Divide M by N, call the remainder R.

   M/N = 20 / 18 = 1, with R ← 2

   M/N = 18 / 2 = 9, with R ← 0

3. If R is not 0, assign M the value of N, assign N the value of R, and go to step 2.

   R = 2. Therefore, M ← 18, N ← 2. Go to step 2.

   R is 0. Therefore, proceed to step 4.

Example Use

Finding the GCD of 18 and 20 (**first time** through, **second time** through)

1. Assign M the value of the larger of the two values, and N the smaller.

   $M \leftarrow 20$ $N \leftarrow 18$

2. Divide M by N, call the remainder R.

   $M/N = 20 / 18 = 1$, with $R \leftarrow 2$

   $M/N = 18 / 2 = 9$, with $R \leftarrow 0$

3. If R is not 0, assign M the value of N, assign N the value of R, and go to step 2.

   $R = 2$. Therefore, $M \leftarrow 18$, $N \leftarrow 2$. Go to step 2.

   R is 0. Therefore, proceed to step 4.

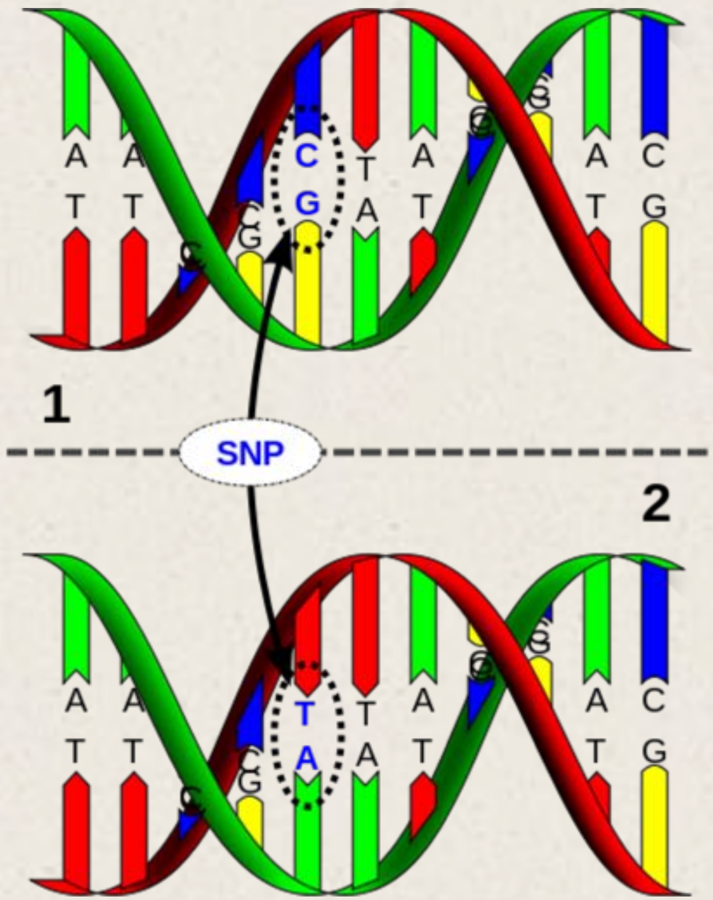
4. The greatest common divisor is N.

   GCD = N = 2

Notable Contemporary Algorithms

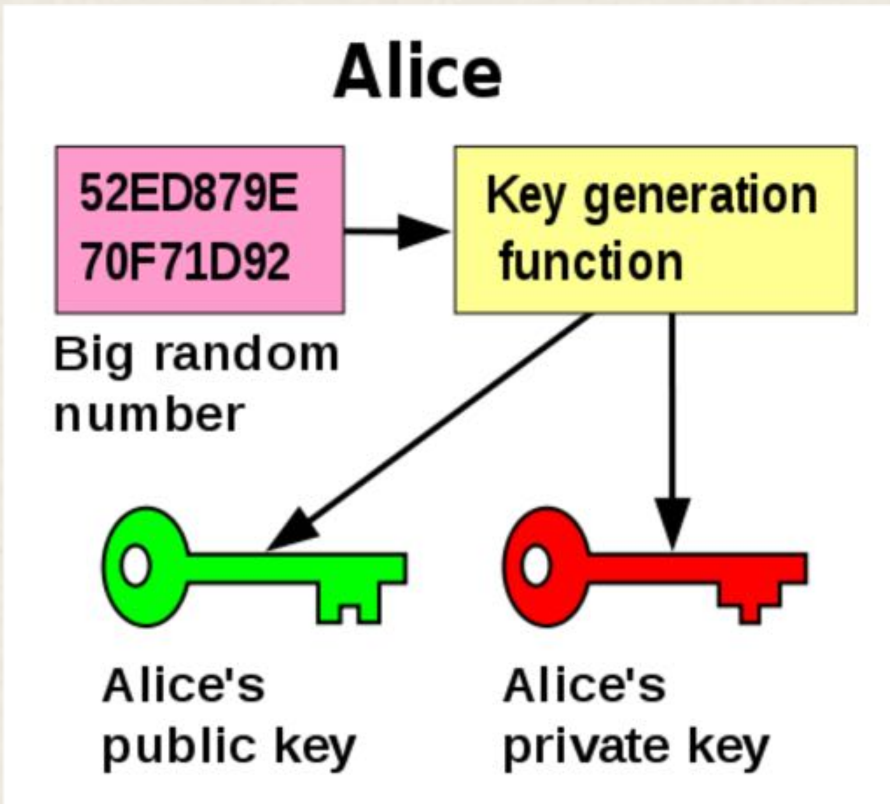
BLAST Algorithm

The sequencing of the human genome was dependent on the development of fast, efficient comparing and matching of DNA sequences.



RSA Algorithm

The RSA algorithm is the basis of public key encryption. It requires the factorization of large prime numbers to break, which for large enough primes, is considered impossible. It is the method used for secure web communication.

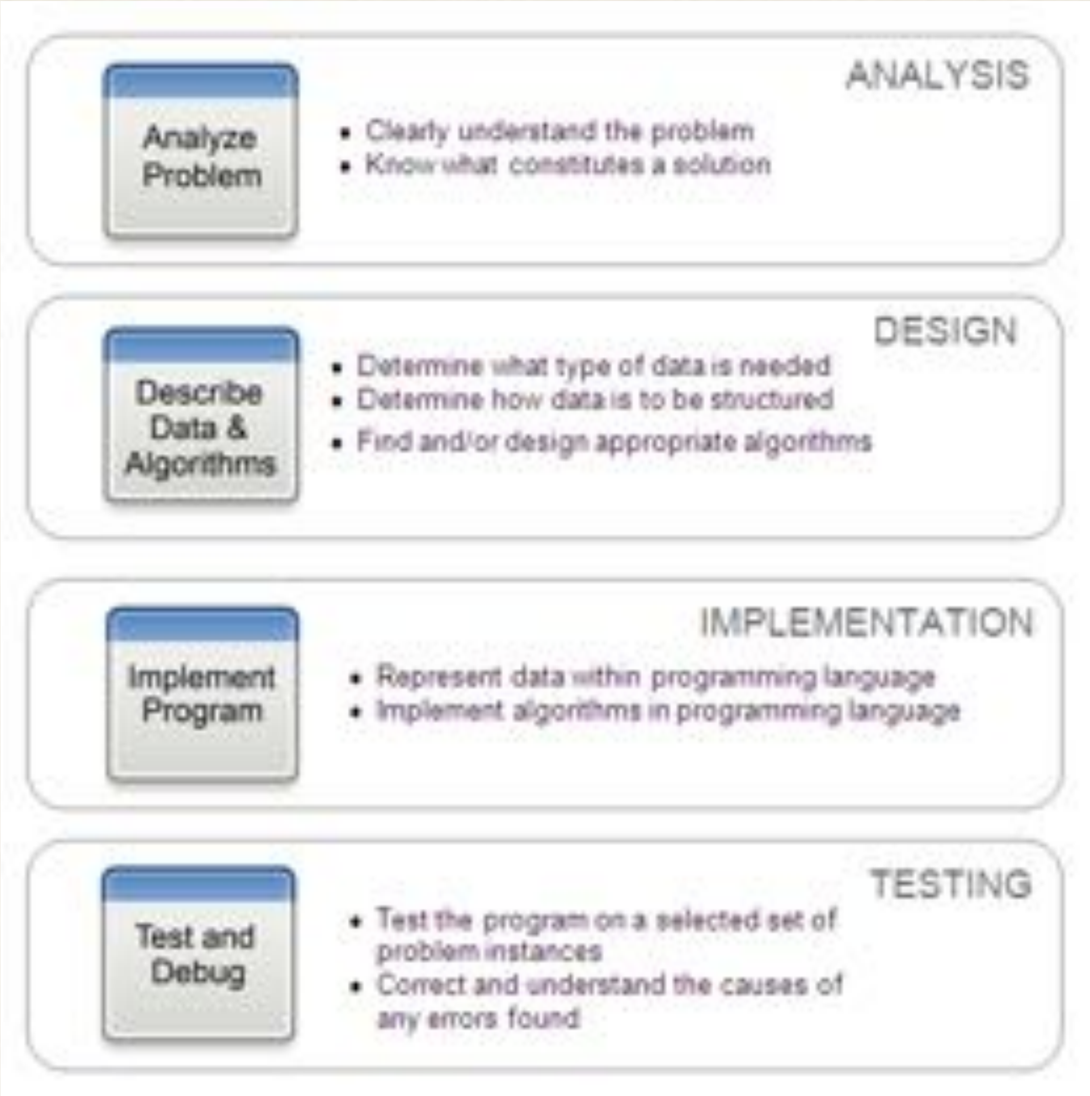


The Process of Computational Problem Solving

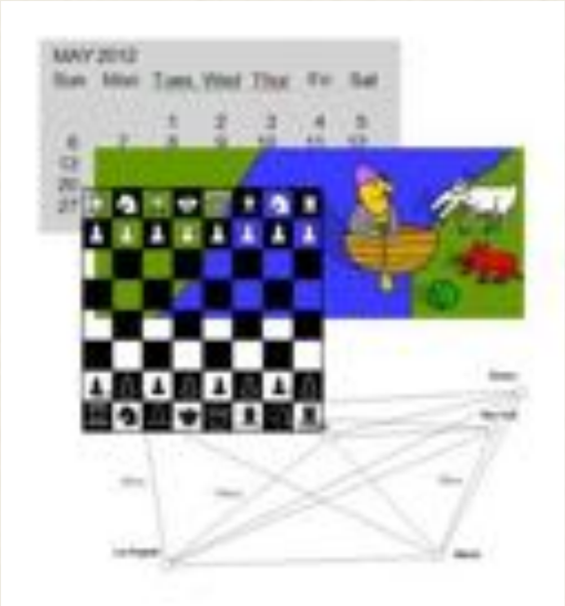
Computational problem solving does not simply involve the act of computer programming. It is a *process*, with programming being only one of the steps.

Before a program is written, a design for the program must be developed. And before a design can be developed, the problem to be solved must be well understood. Once written, the program must be thoroughly tested.

Computational Problem Solving Steps



Problem Analysis



Must understand the fundamental computational issues involved

- For **calendar month problem**, can use direct calculation for determining the day of the week for a given date
- For **MCGW problem**, can use brute-force approach of trying all of the possible rowing actions that may be taken
- For the **Traveling Salesman** and **Chess playing problems**, a brute-force approach is intractable. Therefore, other more clever approaches need to be tried

Knowing what constitutes a solution.

For some problems, there is only one solution. For others, there may be a number (or infinite number) of solutions. Thus, a problem may be stated as finding,

- **A solution** (calendar month, chess playing)
- **An approximate solution**
- **A best solution** (MCGW, Traveling Salesman Problem)
- **All solutions**

Describe Data and Algorithms

- For **calendar month problem**, need to store the month and year, the number of days in each month, and the names of the days of the week

- For the **MCGW problem**, need to store the current state of the problem (as earlier shown)

- For **Traveling Salesman** need to store the distance between every pair of cities

- For the **chess playing problem**, need to store the configuration of pieces on a chess board

Table Representation of Data for the Traveling Salesman Problem

	Atlanta	Boston	Chicago	Los Angeles	New York City	San Francisco
Atlanta	-	1110	718	2175	888	2473
Boston	1110	-	992	2991	215	3106
Chicago	718	992	-	2015	791	2131
Los Angeles	2175	2991	2015	-	2790	381
New York City	888	215	791	2790	-	2901
San Francisco	2473	3106	2131	381	2901	-

Note that only half of the table need be stored

Representation for Chess Playing Program

R	N	B	Q	K	B	N	R
P	P	P	P	P	P	P	P
P	P	P	P	P	P	P	P
R	N	B	Q	K	B	N	R

4	2	3	4	5	3	2	4
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-4	-2	-3	-4	-5	-3	-2	-4
-1	-1	-1	-1	-1	-1	-1	-1

Although the representation on the left is intuitive, the one on the right is more appropriate for computational problem solving.

Describing the Algorithms Needed

When solving a computational problem, either suitable existing algorithms may be found, or new algorithms must be developed.

For the MCGW problem, there are **standard search algorithms** that can be used.

For the calendar month problem, **a day of the week algorithm already exists**.

For the Traveling Salesman problem, **there are various (nontrivial) algorithms that can be utilized** for solving problems with tens of thousands of cities.

Finally, for the chess playing, since it is infeasible to look ahead at the final outcomes of every possible move, **there are algorithms that make a best guess at which moves to make**. Algorithms that work well in general but are not guaranteed to give the correct result for each specific problem are called ***heuristic algorithms***.