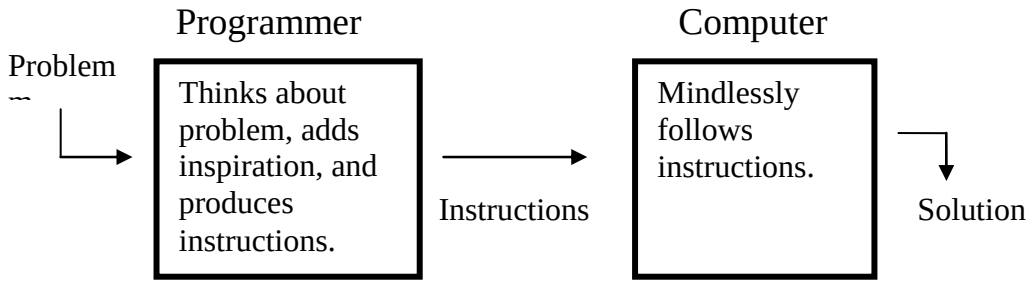
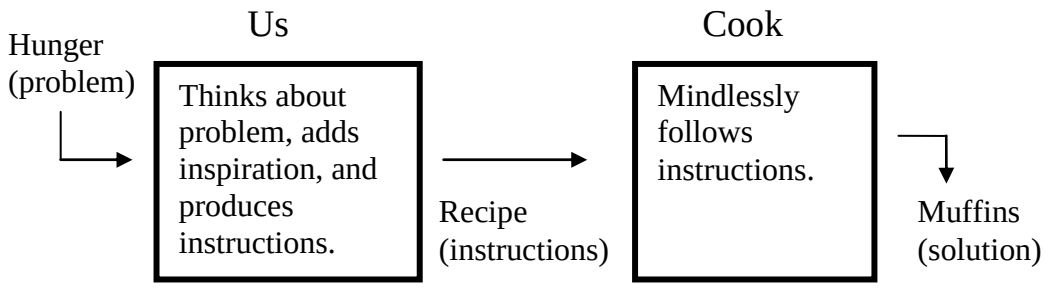
# Chapter 1 - Basic Programming Concepts

The diagram below illustrates the essence of computer programming. The computer is a mindless robot which simply follows the instructions given to it. It follows these instructions very quickly, and very accurately, but that's all it does. It doesn't think. All the thinking is done by the programmer. He (or she) considers the problem to be solved, devises a suitable set of instructions (an *algorithm*), and gives these instructions into the computer. The computer then follows (executes) the instructions, and, if they are correct, the desired solution is achieved.




It is important to keep in mind that computers are dumb (dumber than even your average professor). While they may sometimes look very intelligent indeed (e.g. "Deep Blue" – IBM's famous chess playing computer), this is just because they are following a very clever set of instructions. Computers simply do what they are told to do. If they are given silly instructions, they will produce silly results. Hence the saying: "garbage in – garbage out".

The basics concepts can be illustrated with examples from everyday life. Imagine, for example, that we are hungry (a definite problem). We think about the issue, decide that we would like some muffins, and come up with a suitable recipe (a set of instructions). The recipe is given to a cook who mindlessly follows it and (presuming that the recipe is any good) we wind up with a delicious batch of muffins (a solution to our problem). As can readily be seen by comparing the diagram below to the diagram above, the entire process is analogous to computer programming. The only key difference is that we have "programmed" a cook rather than a computer.

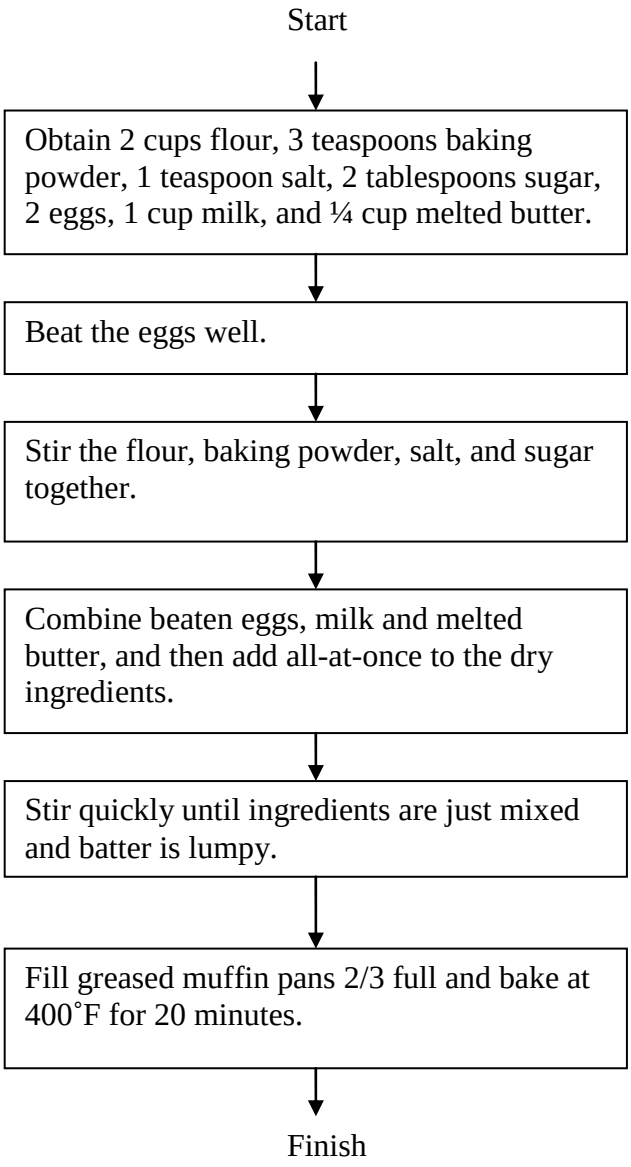


Our recipe might be:

- Obtain 2 cups flour, 3 teaspoons baking powder, 1 teaspoon salt, 2 tablespoons sugar, 2 eggs, 1 cup milk, and ¼ cup melted butter.
- Beat the eggs well.
- Stir the flour, baking powder, salt, and sugar together.
- Combine beaten eggs, milk and melted butter, and then add all-at-once to the dry ingredients.
- Stir quickly until ingredients are just mixed and batter is lumpy.
- Fill greased muffin pans 2/3 full and bake at 400˚F for 20 minutes.

"Garbage in – garbage out" certainly applies. Suppose that, due to a small typographical error, the recipe given to the cook called for the muffins to be baked at 400˚F for 200 minutes. The results would not be what we expected and would certainly not be much of a solution to our problem. Arguably most real cooks would spot such an outrageous mistake but in order to preserve the analogy between this example and computer programming it must be assumed that the cook does not think at all, but instead mindlessly follows the recipe he is given. Never forget that computers do not think at all but just follow the instructions that they are given, regardless of whether these instructions contain obvious errors.

Our instructions for the cook are to be executed sequentially. The cook is to start with the first instruction, move on to the second instruction, and proceed through the instructions one by one. The order in which the steps of the recipe are to be executed can be represented using a "flowchart" (see diagram on next page). The cook is to start at "start" and follows the arrows until "finish" is reached. Flowcharts were once an essential part of computer programming, and every programmer had a plastic template for creating them. With the advent of "structured programming" they have become much less important, but they still can be useful tool, especially for beginners. In this particular case, of course, the "flow" is very straightforward and as a result the flowchart doesn't tell us much that wasn't obvious to begin with. The value of flowcharts is much more evident when dealing with instructions that involve selection (choosing between alternatives) and iteration (doing something over and over again).

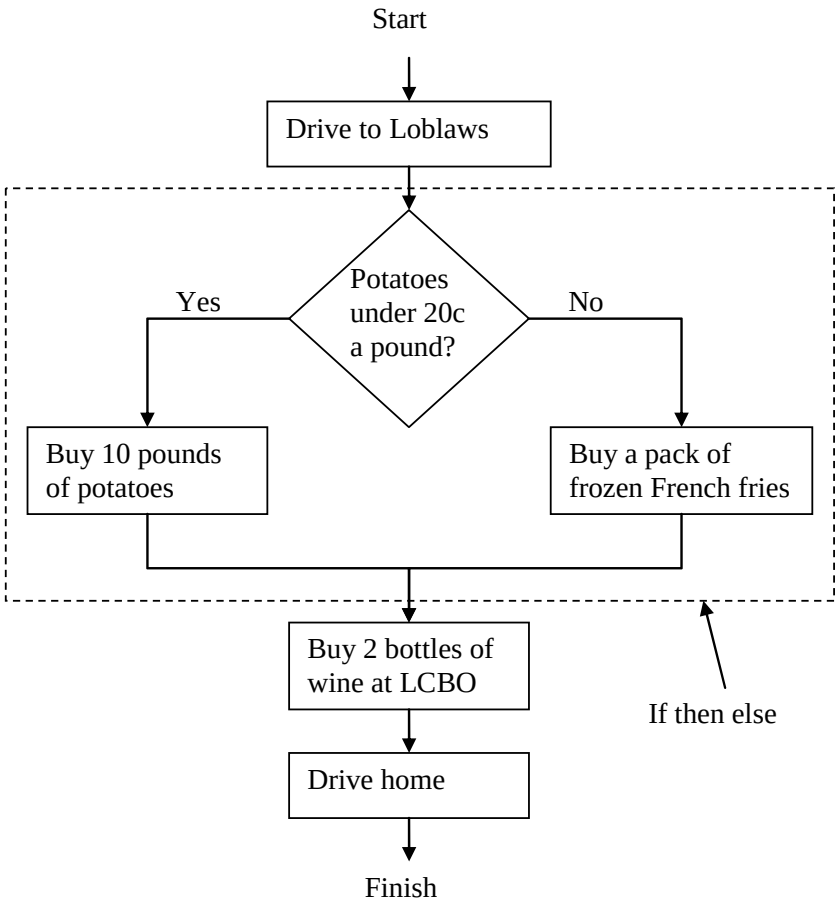



The idea of sequential execution is central to computer programming. In general computers execute the instructions they are given sequentially, just as a cook follows the steps in a recipe. Programs that consist only of a sequence of basic operations are, however, both unusual and not very interesting. Most useful programs involve control structures that provide for selection and iteration.

Suppose that a working woman needs to do some shopping and that she decides to solve this problem by leaving some instructions on the fridge for her stay at home husband. Note that this is once again an exercise in “programming”, though in this case the wife is programming her husband rather than a computer or a cook. The instructions are as follows:

drive to Loblaws
**if** (potatoes are less than 20 cents a pound) **then**
  buy 10 pounds of potatoes
**else**
  buy a pack of frozen French fries
**endif**
buy 2 bottles of wine at the LCBO next door
drive home

The instructions involve selection. Depending upon the price of potatoes, the husband is to buy either 10 pounds of potatoes or some frozen French fries. Upon reaching the "if then else" control structure, the husband must decide whether the controlling condition (potatoes are less than 20 cents a pound) is true or false. If the condition is true, he must follow the instructions in the "then part" of the structure (buy 10 pounds of potatoes). Otherwise he must follow the instructions in the "else part" of the structure (buy a pack of frozen French fries). In either case he must then go on to the instructions following the control structure (buy 2 bottles of wine, drive home). The complete flowchart for the instructions is shown below.



The portion of the flowchart that relates to the “if then else” is identified with a dashed line (note: this is not a regular part of flowcharts). The diamond shaped flowchart element represents a decision. Upon reaching such an element the flow proceeds in one of two possible directions, depending upon whether the condition inside the diamond is true or false. If the condition is true the instructions belonging to the “then part” of the “if then else” construct are executed and if the condition is false the instructions belong to the “else part” are. The two flow paths come back together at the end of the “if then else”. Regardless of which path was taken, execution of the instructions continues with the flowing instruction.

The general rule for the execution of an “if then else” is:

1. Evaluate the true or false condition.
2. Execute either the instructions in the true part (condition true) or the else part (condition false).
3. Resume execution at the instruction after the “if then else” (i.e. at the instruction after the “endif”)

The “then part”“ and “else part” of an “if then else” need not involve only a single instruction. Instead both parts may involve any number of instructions. The husband could, for example, be given the instructions below.

```
drive to Loblaws
if (potatoes are less than 20 cents a pound) then
   buy 10 pounds of potatoes
   buy some steak
else
    buy a pack of frozen French fries
    buy a fillet of fish
endif
buy 2 bottles of wine at the LCBO next door
drive home
```

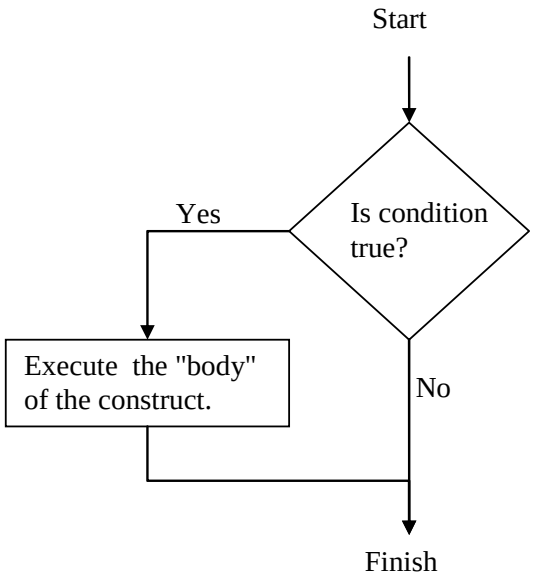
Sometimes the “else” portion of an “if then else” is not required and is omitted. Imagine, for example, that the wife would like the husband to buy some bananas, but only if they are on sale. If the wife were forced to use only “if then else”'s in writing her instructions she would have to add the instruction below. This works, but is clearly rather clumsy.

```
if (bananas are on sale) then
    buy 2 pounds of bananas
else
    do nothing
endif
```

The situation really calls for a “simple if” as shown below.

**if** (bananas are on sale) **then**
    buy 2 pounds of bananas
**endif**

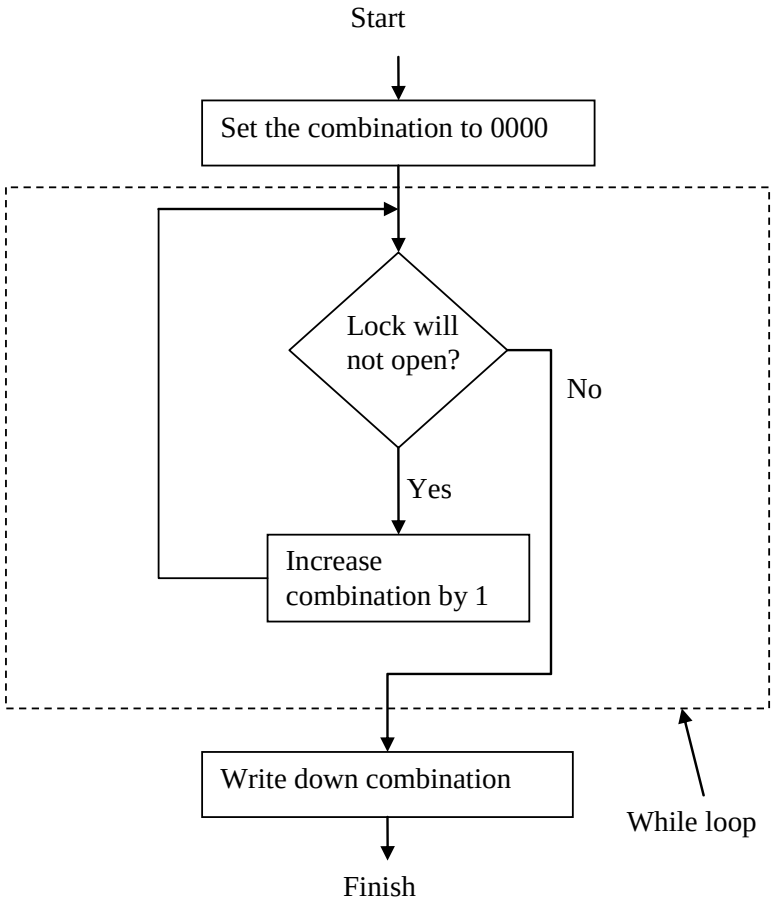
The “simple if” is logically equivalent to an “if then else” with a “do nothing” else part. The construct is not absolutely essential (any “simple if”'s in a program can always be replaced with “if then else”s with “do nothing” else parts) but is certainly useful. The general flowchart for a “simple if” is given below.




Many programs require that something be done over and over again (iteration). The basic control structure used is the “while loop”. Imagine that somebody has forgotten the combination to their bicycle lock and wants our help in solving this problem. The lock is one of those with four rings, each of which can be set to any value between 0 and 9 (there are 10,000 possible combinations, ranging from 0000 to 9999). After considering the situation, we “program” the person by supplying them with the following instructions.

set the combination to 0000
**while** (the lock will not open) **do**
    increase the combination by 1
**endwhile**
write down the combination for future reference

These instructions involve iteration. The person may (and probably will) have to increase the combination by 1 many times before the lock ultimately opens.The complete flowchart for the instructions is shown below.




The portion of the flowchart that relates to the “while loop” is identified with a dashed line (note: again this is not a regular part of flowcharts). It involves a loop (hence the “loop” in “while loop”). If the answer to “lock will not open?” is yes the combination is increased by one and the question is asked again, and so on and so on until the answer to the question becomes no.

The general rule for the execution of a “while loop” is:

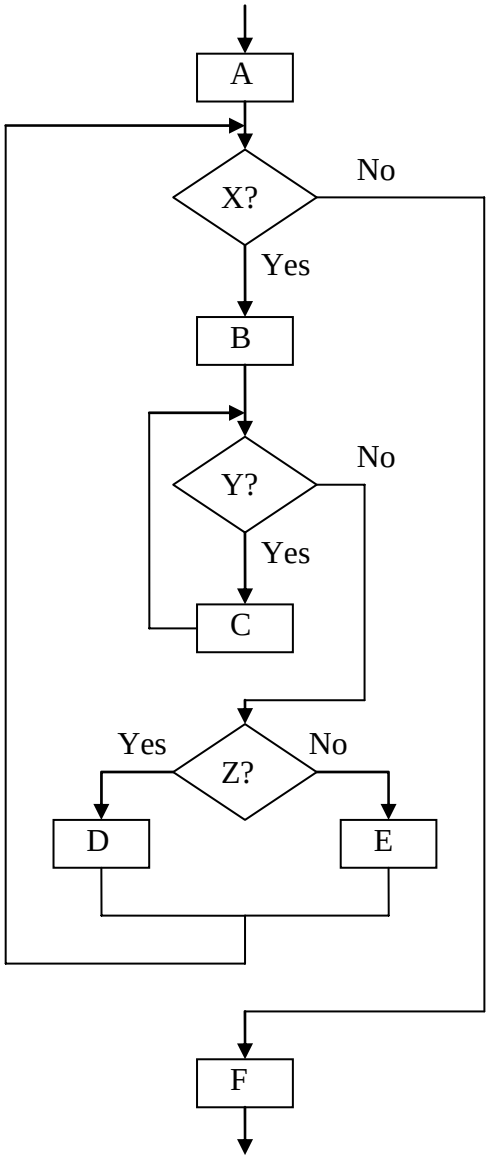
1. Evaluate the true or false condition.
2. If the condition is false skip the rest of the construct and proceed directly to the following instruction (i.e. the instruction after the “endwhile”). Otherwise begin executing the instructions contained in the while loop.
3. When all of the instructions in the while loop have been executed (i.e. the “endwhile” is reached) go back to step 1.

The while loop is a “pre-test” loop. The condition is tested before anything is done and, if the condition is initially false, the instructions in the loop are not executed at all. In our example the combination will never get increased by 1 if the actual combination is 0000 (try following the flow chart from start to finish for this case).

In our example the while loop contains just one basic instruction (increase combination by 1). These need not be the case. While loops may contain any number of instructions and these instructions may include while loops and if constructs (“if then elses”s and “simple if”'s). For example, the following is quite permissible:

```
Do action A
while condition X do
   Do action B
   while condition Y do
      Do action C
   endwhile
   if condition Z then
      Do action D
   else
      Do action E
   endif
endwhile
Do action F
```

The flow chart for these instructions is shown on the next page. To complete the picture, the “then” and “else” parts of an “if then else” may also include while loops and if constructs. One can have if's inside while's inside if's inside while's (and so on and so on). Lest this seems very scary, it must be said that in practice it is rarely necessary or desirable to take things beyond a few levels of “nesting”. When things get too complex or are in danger of becoming so, functions (covered later in the course) can be used to simplify matters by, in effect, combining groups of instructions into single instructions.




The overall picture can be summarized as follows:

- A “statement” is either a basic instruction (do something or other), an “if then else”, a “simple if”, or a “while loop”.

- A “statement list” is a collection of statements. Execution rule: The statements are executed sequentially, starting from the first and running through the last. The execution of each statement is completed before the execution of the next is begun.

- A “program” is a statement list.

- An “if then else” involves a true or false condition and two statement lists (the “then part” and the “else part”). Execution rule: If the condition is true the “then part” is executed. Otherwise the “else part” is.

- A “simple if” involves a true or false condition and a statement list. Execution rule: If the condition is true the statement list is executed. Otherwise nothing is done.

- A “while” involves a true or false condition and a statement list. Execution rule: the statement list is repeatedly executed as long as the condition is true.

Sequential execution, selection and iteration are the bread and butter of computer programming. The “if then else” and the “while loop” (and their variants, such as the “simple if”) represent the basic vocabulary used in putting programs together. A thorough understanding of these constructs is absolutely essential to be being able to program. While understanding the concepts does not guarantee success, not understanding them certainly guarantees failure. Just as it is impossible to, say, compose French poems without being able to speak French, it is absolutely impossible to program without being able to speak “if then else” and “while loop”.

## Case Study #1

Imagine that our priceless collection of hockey cards has been dropped and is lying all over the floor. Each card has a number on it, and eventually we’d like to get all of the cards back in order. To begin with, however, we’ll consider a simpler problem – that of just finding the card with the highest number. This promises to be a tedious job and, as we’d like to go out instead, we hit upon the idea of having our kid brother do the work. He’s not too bright (which just as well – who else would do a job like this?) but follows instructions well, and is willing to do the job for a very reasonable price. Our challenge is to come up with an appropriate set of instructions. Or, to put it another way, we need to suitably “program” our kid brother.

After thinking for a while, we give our brother a piece of paper containing the instructions below and then head out the door.

**Instructions for Finding Highest Card**

Find three cardboard boxes. Label them A, B, and C.
Gather up all of the cards and pile them in box A.
Move the top card in box A to box B
**while** box A is not empty **do**
    **if** the top card in A has a higher number than the top card in B **then**
        Move the top card from box A to box B
    **else**
        Move the top card from box A to box C
    **endif**

**endwhile**
Stop and relax (the highest numbered card is the top card in box B).

**Exercise:** Make six or so numbered cards, shuffle them, and follow the instructions. Keep the cards – they'll come in useful in future exercises.

When we come home, our kid brother is relaxing and the highest numbered card is sitting on top of box B. Our instructions have worked, and our problem has been solved. While this example doesn't actually involve a computer, it does illustrate all of the key aspects of computer programming. We have a problem (to find the highest numbered card), we have a "robot" (in this case our kid brother), and solving the problem requires producing a set of instructions for the "robot" to follow. The only fundamental difference between what we've done and computer programming is that, instead of programming a computer, we have programmed our kid brother.

Just as when programming computers, "garbage in – garbage out" applies. Imagine, for example, that, in our haste to leave, we wrote line #8 as "Move the top card from box A to box B" instead of "Move the top card from box A to box C". In this case, we would come home to find all of the cards in box B, and the highest numbered card could be anywhere in the pile. The fault would be entirely ours. Just as there is never any point in getting upset with a computer when a computer program doesn't work as intended (why couldn't the silly machine see what I wanted!), there would be no point in getting upset at our brother. He did his job correctly, following his instructions exactly as they were written. Computers (and, in this example, our brother), aren't meant to be anything other than silly. Their job is not to judge the instructions they are given, but simply to do exactly as they are told. The problem was that our instructions were incorrect. In computer programming jargon, they contained a "bug" (a programming error).

Perhaps, at this point, you are wondering just how we came up with the instructions for finding the highest numbered card. If so, you have gotten to the heart of the matter. Getting from a problem to a suitable set of instructions is what programming is all about. There is no simple answer to the question "how does one program". Programming is not a mechanical process. There is no set of rules that will always take one from a problem to suitable instructions, and the process always involves an element of creativity. As one becomes experienced, however, one develops a kind of toolkit of ideas that can be applied. Our particular set of instructions, or "algorithm", is an example of a technique that is often employed to find the largest (or smallest) of a set of values.

Box "B" is used to contain our "best card so far". The top card in this box is always the highest numbered card we've seen. At the very beginning, when all of the cards are in box "A", the top card in this box is the highest numbered card that we've seen. We therefore move this card to box "B". For each of the remaining cards in box "A", we decide whether or not the card is better than our "best card so far". If it is, it goes into box "B", and becomes our new "best card so far". Otherwise it goes into box "C" – the reject box.

**Exercise:** Suppose we wanted to find the lowest numbered card instead of the highest numbered card. How would we have to modify the instructions?

Recall that our ultimate goal was to get all of the cards in order. Creating instructions for finding the highest numbered card was just a first step. It was, however, a very useful step. Imagine that, after the highest numbered card is found, it is put into yet another cardboard box. Applying the same instructions to the remaining cards will then produce the second highest card. If it is put into the same box, this box will now contain the highest two cards in their proper order. And so on, until no cards remain to be processed, at which point the box will contain all of the cards, properly sorted. If we try and convert this idea into a set of instructions we might, as a first step, come up with something like the following:

> Find the highest numbered card.
> Put in into an empty box.
> Find the highest numbered of the remaining cards (the second highest card).
> Put it into the same box.
> Find the highest numbered of the remaining cards (the third highest card).
> Put it into the same box.
> and so on…

These instructions suffer from being somewhat imprecise. "And so on…" is too abstract a concept for our kid brother. He requires something a bit more definite (as do computers). Does the fact that something has to be done over and over again suggest a while loop? If so, you're on the right track. Whenever your thought process leads you to an "and so on…", chances are that you need a loop. The set of instructions below show how the idea that we've been working on can be precisely expressed using a while loop.

**Instructions for Sorting Cards**

> Find a cardboard box. Label it D.
> **while** all cards are not yet in box D **do**
> *find the highest numbered of the remaining cards*
> Move this card to box D.
> **endwhile**
> Stop and relax (box D contains all of the cards in their proper order).

As a final step in producing a complete card sorting, we must replace the abstract "find the highest numbered of the remaining cards" with the actual instructions needed to do this. This is an example of a design process that we will see over and over again – "stepwise refinement". We begin by assuming relatively abstract concepts (like finding the highest card), and then flesh out the details afterwards. In this case, of course, we are ahead of the game in that we've already developed instructions for finding the highest numbered card. Inserting these instructions in place of "find the highest numbered card" will produce a rather more complex algorithm. But if the sorting algorithm makes sense to you when expressed in terms of "finding the highest numbered card", it should

still make sense (as scary as it may look) when the details are filled in. The result is shown below. Note, because the instructions for finding the highest numbered card involve a while loop, we end up with a loop within a loop. This is common in programming. As noted earlier, the body of a while loop may involve while loops, which in turn may involve further while loops, and so on. Similarly, the "if" and "else" parts of an "if then else" may involve further "if then else" constructs, which in turn can have further "if then else" constructs in their "if" and "else" parts, and so on. The term "nested" is applied to such situations. In our case, we have nested loops. An "inner loop" is nested within an "outer loop".

**Algorithm for Sorting Cards**

```
Find four cardboard boxes.  Label them A, B, C, and D.
while all cards are not yet in box D do
        Take all cards that are not yet in box D and put them in box A.
        Move the top card in box A to box B
        while box A is not empty do
                if the top card in A has a higher number than the top card in B then
                        Move the top card from box A to box B
                else
                        Move the top card from box A to box C
                endif
        endwhile
        Move the top card in box B card to box D.
endwhile
Stop and relax (box D contains all of the cards in their proper order).
```

**Exercise:** Draw the flow chart for this algorithm.

**Exercise:** Take the cards you made earlier and sort them by applying the algorithm. Act dumb (as a computer would) and just and follow the rules. Whenever you get to a "while", decide whether or not the condition is true. If it is, start executing the body of the while loop. Otherwise skip to the instruction after the matching "endwhile". Whenever you get to an "endwhile", loop back to the matching "while" and re-evaluate the condition. Note the "matching" - don't get the outer and inner loops mixed up!

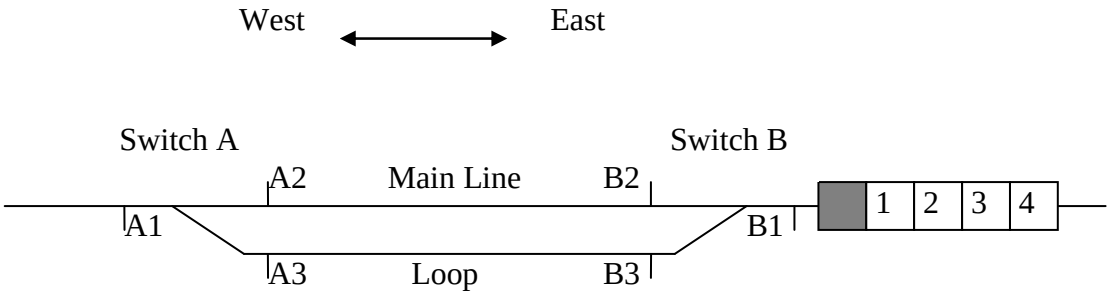
If we leave these instructions with our brother, and go away for long enough, we should come home to find our card collection properly sorted. The algorithm works. Often, and particularly when dealing with small problems, this is all that really matters. In general, however, efficiency is also a consideration. An algorithm which allows our brother to sort the cards in, say, two hours, is obviously better than one which works but keeps him up all night. Our algorithm is in fact not very efficient. Consider the situation just after the largest card has been found and moved to box D. At this point we move all of the remaining cards back to box A and start looking for the second highest card. But do we have to really have to look through all of the remaining cards to find the second largest? The answer is no – of the cards left in box B, only the top card need been considered, as all of the cards below it are guaranteed to be lower. To find the second highest card, we

need only compare the top card in box B (assuming there is one) with all of the cards left in box C. We can also take advantage of the fact that, at all times, the cards in box B will be sorted (in reverse order). Once all of the cards are either in box B or box D, all of the cards in box B can simply be transferred, one by one, to box D. Do not worry if this is a bit hard to follow. All that really matters is that you understand that efficiency is a consideration, and that some algorithms are better than others.

**Exercise (advanced):** Make the algorithm more efficient by modifying it to take advantage of the observations above.

## Case Study #2

The diagram below depicts a railway yard. A train arriving from the east (with the engine at the west end) is to return to the east (with the engine at the east end). The two switches (switch A and switch B) can be set for either the main line or the loop. Six track positions (A1, A2, A3, B1, B2, and B3) are identified.




The train crew are rookies and need our help in turning the train around. After considering the problem, we produce the following instructions.

Move west until the engine arrives at B1.
Ensure switch B set for the main line (B1 to B2).
Move west until engine arrives at A2
Ensure switch A set for the main line (A1 to A2)
Uncouple engine from train.
Move west until the engine is past A1.
Set switch A to for the loop (A1 to A3)
Move east until engine arrives at B3.
Set switch B to for the loop (B1 to B3).
Move east until engine past B1.
Set switch B for the main line (B1 to B2)
Move west until the engine couples onto train.
Depart to the east.

Aside: This procedure is known as “running around the train”.

Now let us consider a rather more interesting program. Suppose the railway regulations require that the order of the cars be preserved when the train returns to the east (that car #1 is still just behind the engine, that car #2 is directly behind it car #1 and so on). Although the diagram shows only four cars, our instructions should work for trains of any length (provided that the cars of the train will fit between points A2 and B2 and between points A3 and B3). This requires rather more thought, but eventually we come up with the instructions shown below. The first step involves doing all but the last three steps of the original instructions. In the interest of brevity the details are not repeated.

```
Pull train to A2 and return engine to east of B1 (details as before)

while cars remain between A2 and B2 do
   Set switch B to for the main line (B1 to B2).
   Move west until engines couples to cars.
   Uncouple car next to engine from any other cars.
   Move east until attached car is past B1.
   Set switch B for the loop (B1 to B3)
   if there are no cars between A3 and B3 then
      Move west until attached car reaches A3.
   else
      Move west until attached car couples to cars.
   endif
   Uncouple engine.
   Move east until engine is past B1.
endwhile
Move west until engine couples to cars.
Depart to the east.
```

These instructions will work, but once the cars of the train are all between A3 and B3 the engine will be rather pointlessly uncoupled from the train, taken to point B1, and then brought back onto the train. As shown below, this difficulty can be overcome by the use of a “simple if”.

```
Pull train to A2 and return engine to east of B1 (details as before)

while cars remain between A2 and B2 do
   Set switch B to for the main line (B1 to B2).
   Move west until engines couples to cars.
   Uncouple car by engine from any other cars.
   Move east until attached car is past B1.
   Set switch B for the loop (B1 to B3)
   if there are no cars between A3 and B3 then
      Move west until attached car reaches A3.
   else
```

```
      Move west until attached car couples to cars.
    endif
    if cars remain between A2 and B2 then
      Uncouple engine.
      Move east until engine is past B1.
    endif
  endwhile
  Depart to the east.
```

**Exercise:** The above sets of instructions assume that the cars of the train will fit between points A2 and B2 and between points A3 and B3. Suppose that this isn't the case and that there is only room between A2 and B2 and between A3 and B3 for a single car. Provide the train crew with instructions to the two cases considered above (cars left as they are, cars reversed). You may assume that there is plenty of track to the west of point A1.

## Case Study #3

While they do illustrate many of the key aspects of computer programming, our previous examples have their limitations. Although punched cards have a prominent position in computing history, computers don't manipulate hockey cards, and they also don't move railway cars about. Instead they process data (crunch numbers). In view of this, let us move on to an example involving numbers. Imagine that we've a friend who needs to compute factorials, and that he insists on being given precise instructions.

For those who might not familiar with factorials, 4 factorial (usually represented as 4!) is equal to 4 x 3 x 2 x 1 (or 24). 5! is equal to 5 x 4 x 3 x 2 x 1 (or 120), and so on. In general…

$$n! = n \times (n - 1) \times (n - 2) \times (n - 3) \ldots\ldots \times 1$$

If somebody were to ask you to compute, say, 6!, you might start with 6 and keep multiplying by smaller value (5, 4, and so on) until you reached one. Or you might do things the other way around, starting with 1 and working up to 6. Either way, you would probably just consider the steps involved to be obvious, and do them more or less naturally. In the case of our friend (or a computer), however, we can't just say "it's obvious" or give loose description of a process (as we've done here). Instead we must precisely define what must be done to compute the factorial of a number. We must consider the process, work out exactly what is we're doing, and produce a set of instructions.

Let us assume that our friend has a piece of paper, a pencil, an eraser, and a calculator that allows him to perform basic arithmetic operations such as addition and multiplication. He is able to follow instructions expressed in the usual way and in particular understands while loops and if-then-else constructs. He is not, however, prepared to do any thinking. Instead he will have to be told exactly what to do. Our friend, as we shall see, is a pretty good approximation of a computer.

Now, just what is it we do when we compute the factorial of the number? Keep in mind that our goal here is not to calculate the factorial of a number (something we can all do pretty easily) Instead we want to put together a precise description of the steps required (a bit harder, at least for newcomers to programming). We begin with a number, multiply it the number less one, multiply the result by the number less two, and so on until we're done. The "and so on" suggests that we're going to need a while loop, and this is in fact the case.

**Algorithm for Computing N!**

Draw two boxes on the paper. Label one N and the other RESULT.

Put the value of interest in N.
Put the value 1 in RESULT
**while** the value in N is greater than 1 **do**
    Take the values in N and RESULT, multiply them together, and put the answer in RESULT (after erasing the previous value)
    Take the value in N, subtract one from it, and put the answer in N (after erasing the previous value)
**endwhile**
Stop – the answer is now in RESULT.

**Exercise:** Use the algorithm to compute 5!

There are three levels of understanding in play here. Ask yourself the following questions:

1/. Can you apply the algorithm?
2/. Can you see how it works?
3/. Could you have produced the algorithm by yourself?

Do not be distressed if your answer to the third question is "no". We have, after all, only just started. Also we did rather leap from deciding that a loop would be requiring to presenting the final product. In part this is because the thought process involved is difficult, if not impossible, to put into words. To a large extent, programming requires something of a "creative spark". On the other hand, experience is also involved. Once you've had some practice at putting together algorithms, this particular problem should (and probably will) come to look very easy indeed.

How did we know that the algorithm should involve named "boxes" (N and RESULT), and that the instructions should involve putting numbers in these boxes, with each box only ever containing one number at a time? This much definitely came from experience. Though this particular algorithm is intended for a friend, the ultimate objective is to learn how to program computers, and computer programs always have this general form. First we define "boxes", and then we have the computer move data from box to box, performing computations as it does so. And, as computer "boxes" can only ever contain

one value (putting a new value in a box destroys the previous one), we had our friend operate the same way.

Experience also played a role in producing the details of the algorithm. With experience, you'll develop a kind of "toolkit" of standard solutions to common problems. You'll come to know, for example, that the product of a number of terms can computed using a loop in which a single multiplication is performed.

problem:

to compute answer = term1 * term2 * term3 ….

"standard" solution:

Put the value 1 in ANSWER
**while** …. **do**
Take the value in ANSWER, multiply it by the next term,
and put the result back in ANSWER
…..
**endwhile**

Similarly, you'll come to realize that, if you want to do something a number of times, this can be achieved by using a "countdown" loop.

problem:

to do something N times

"standard" solution:

Put the desired count into N
**while** the value in N is greater than zero **do**
Do whatever is required
Take the value in N, subtract 1 from it, and put the result back in N
**endwhile**

The algorithm presented is essentially the combination of these two patterns, although some creativity was also involved in its creation. The "standard" countdown loop was modified, for example, so as to avoid multiplying by one, which would be a waste of time. Instead of having "while the value in N is greater than zero" the algorithm has "while the value in N is greater than 1".

**Exercise:** Modify the algorithm so that the multiplications are done in the reverse order (so that we start with 2 and work up to the number of interest). You'll find that an extra box is required.

**Exercise:** Write an algorithm (for our friend) which computes 2 to the Nth power. Make sure that your algorithm works for any integral N greater than or equal to zero (recall that 2 to the zero is one).

**Exercise (advanced):** Modify your algorithm for computing 2 to the Nth power so it works for negative N. You will need to make use of the if..then..else construct.