CE 209
Computation for Civil Engineers

2019–2020 Fall

Part I

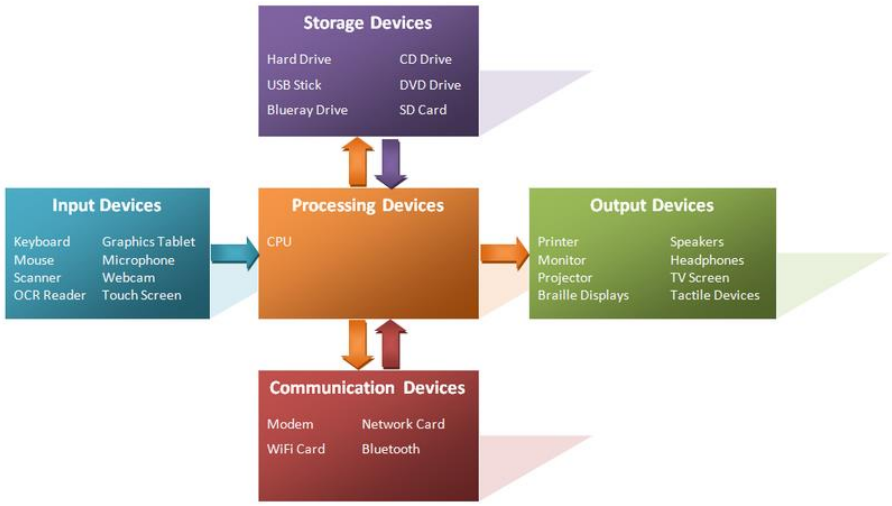
Textbooks

[Nagar 2018] Introduction to Octave for Engineers and Scientists.

[Farrell 2015] Programming Logic and Design - Comprehensive Version 8e.

❶ *COMPUTER SYSTEMS*

- A **computer system** is a combination of multiple pieces of **hardware** and **software** required to process and store data using a computer.
  - **Hardware** is the equipment, or the physical devices, associated with a computer. For example, keyboards, mice, speakers, and printers are all hardware.
  - **Software** is a set of **programs**.
  - **Programs** are **instruction sets** written by programmers that tell the hardware what to do.
  - When you write **software instructions**, you are **programming**.
  - This course focuses on the programming process.

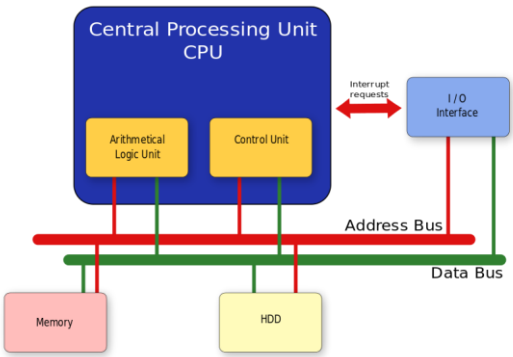


- Software can be classified into two broad types:
  - **Application software** comprises all the programs you apply to a task, such as wordprocessing programs, spreadsheets, payroll and inventory programs, and games. When you hear people say they have "downloaded an **app** onto a mobile device," they are simply using an abbreviation of application.
  - **System software** comprises the programs that you use to manage your computer, including operating systems such as Windows, Linux, or UNIX for larger computers and Google Android and Apple iOS for smartphones..
- Together, computer hardware and software accomplish three major operations in most programs:
  - ❶**Input**: Data items enter the computer system and are placed in memory, where they can be processed. Hardware devices that perform input operations include keyboards and mice. **Data items** include all the text, numbers, and other raw material that are entered into and processed by a computer, such as, facts and figures about entities, images, sounds, and a user's mouse or finger-swiping movements.
  - ❷**Processing**: Processing data items may involve organizing or sorting them, checking them for accuracy, or performing calculations with them. The hardware component

that performs these types of tasks is the **central processing unit**, or **CPU**. Some devices, such as tablets and smartphones, usually contain multiple processors. Writing programs that efficiently use several CPUs requires special techniques.

- ❸**Output**: After data items have been processed, the resulting information usually is sent to a printer, monitor, or some other output device so people can view, interpret, and use the results. Programming professionals often use the term data for input items, but use the term **information** for data that has been processed and output. Sometimes you place output on **storage devices**, such as your hard drive, flash media, or a cloud-based device. (The **cloud** refers to devices at remote locations accessed through the Internet.) When you send output to a storage device, sometimes it is used later as input for another program.

- You write computer instructions in a computer **programming language** such as Visual Basic, C#, C++, or Java.
- The instructions you write using a programming language are called **program code**; when you write instructions, you are **coding the program**.
- Every programming language has rules governing its word usage and punctuation. These rules are called the language's **syntax**. Mistakes in a language's usage are **syntax errors**. Unless the syntax is perfect, the computer cannot interpret the programming language instruction at all.
- When you write a program, you usually type its instructions using a keyboard.
- When you type program instructions, they are stored in **computer memory**, which is a computer's temporary, internal storage. **Random access memory**, or **RAM**, is a form of internal, volatile memory. Programs that are currently running and data items that are currently being used are stored in RAM for quick access.
  - Internal storage is **volatile**—its contents are lost when the computer is turned off or loses power.
  - Usually, you want to be able to retrieve and perhaps modify the stored instructions later, so you also store them on a permanent storage device, such as a disk. Permanent storage devices are **nonvolatile**—that is, their contents are persistent and are retained even when power is lost.



- After a computer program is typed using programming language statements and stored in memory, it must be translated to **machine language.** Machine language is also called **binary language**, and is represented as a series of **0**s and **1**s.
  - Your programming language statements are called **source code**, and the translated machine language statements are **object code**.
  - Each programming language uses a piece of software, called a **compiler** or an **interpreter**, to translate your source code into machine language.
  - Although there are differences in how compilers and interpreters work, their basic function is the same—to translate your programming statements into code the computer can use. When you use a **compiler**, an entire program is translated before it can execute; when you use an interpreter, each instruction is translated just prior to execution. Usually, you do not choose which type of translation to use—it depends on the programming language.
  - After a program's source code is successfully translated to machine language, the computer can carry out the program instructions. When instructions are carried out, a program **runs**, or **executes**. In a typical program, some input will be accepted, some processing will occur, and results will be output.
  - Besides the popular, comprehensive programming languages such as Java and C++, many programmers use **scripting languages** (also called **scripting programming languages** or **script languages**) such as Python, Lua, Perl, and PHP.
    - Scripts written in these languages usually can be typed directly from a keyboard and are stored as text rather than as binary executable files.
    - Scripting language programs are interpreted line by line each time the program executes, instead of being stored in a compiled (binary) form.
  - Still, with all programming languages, each instruction must be translated to machine language before it can execute.

Paradigms of Programming

A **programming paradigm** is a fundamental style of programming based on distinct concepts that shape the way programmers design, organize and write programs.

Each paradigm is not just a *programming* style but usually involves an end-to-end approach for building systems, including analysis, design, implementation, testing, deployment and maintenance.

- ***Procedural (Structured, Imperative) Programming***
  - Control structures govern the order of execution of computational steps called commands.
  - Similar to descriptions of everyday routines, such as food recipes.
  - Typical commands offered by imperative languages are assignment, I/O, procedure calls.
  - Abstracts one or more actions into a procedure, which can be called as a single command (Procedural Programming).
- ***Object-Oriented Programming***
  - Based on models of human interaction with real world phenomena.
  - Data as well as operations are encapsulated in objects.
  - Information hiding is used to protect internal properties of an object.
  - Objects interact by means of message passing which applies an operation on an object.
  - Objects are created by using templates called classes. This allows the programming of the classes, as opposed to programming of the individual objects.
  - Classes are organized in inheritance hierarchies which provides for class extension or specialization.
- ***Declarative Programming***
  - Expresses the logic of a computation without describing its control flow (tells what to do, not how to do it).
  - Common declarative languages include those of database query languages (e.g., SQL, XQuery), regular expressions, logic programming, functional programming, and configuration management systems.
  - Functional Programming
    - Based on mathematical theory of functions.
    - The values produced are *non-mutable.*
      - Impossible to change any constituent of a composite value.
      - However, it is possible to make a revised copy of composite value.
    - All computations are done by applying (calling) functions.
    - Functions are first class values.
    - All computations are done by applying (calling) functions.

- Functions are first class values.

- ***Logic Programming***
  - Answers a question via search for a solution.
  - Based on axioms, inference rules, and queries (i.e., automatic proofs within artificial intelligence).
  - Program execution becomes a systematic search in a set of facts, making use of a set of inference rules.

- Many other programming paradigms exist (constraint programming, symbolic programming, generic programming, + about 30 more at the time of writing).
- For this course, we use a **scientific procedural language**.
- Scientific procedural languages: To simplify small proof-of-principle computations, specialized scientific languages such as **MATLAB®**, and **Octave®** and symbolic manipulation languages such as **MAPLE®** or **Mathematica®** provide an easily learned high-level user interface to a unified built-in array of easily called and highly optimized numerical, scientific and graphical libraries.
  - MATLAB code can be transformed into C++ through an add-on product while C++ and FORTRAN routines can be called by a MATLAB program with some effort.
  - MATLAB is unavailable at many sites because of its substantial cost, although this can, however, increasingly be circumvented, through free software packages that imitate MATLAB commands. This course accordingly employs the most widely employed alternative, **GNU Octave**.

❷ *UNDERSTANDING SIMPLE PROGRAM LOGIC*

- A programmer's job involves writing instructions (such as those in the doubling program in the preceding section), but a professional programmer usually does not just sit down at a computer keyboard and start typing. The **program development cycle** can be broken down into at least seven steps:

  1. Understand the problem.
  2. Plan the logic.
  3. Code the program.
  4. Use a compiler or interpreter to translate the program into machine language.
  5. Test the program.
  6. Put the program into production.
  7. Maintain the program.

- ❶ ***Understanding the Problem***: Professional computer programmers write programs to satisfy the needs of others, called **users** or **end users**.
  - Examples of end users include a Human Resources department that needs a printed list of all employees, a Billing department that wants a list of clients who are 30 or more days overdue on their payments, and an Order department that needs a Web site to provide buyers with an online shopping cart.
  - Because programmers are providing a service to these users, programmers must first understand what the users want. When a program runs, you usually think of the logic as a cycle of input-processing-output operations, but when you plan a program, you think of the output first.
  - After you understand what the desired result is, you can plan the input and processing steps to achieve it.
    - Suppose the director of Human Resources says to a programmer, "*Our department needs a list of all employees who have been here over five years, because we want to invite them to a special thank-you dinner*."
    - On the surface, this seems like a simple request. An experienced programmer, however, will know that the request is incomplete. For example, you might not know the answers to the following questions about which employees to include:
      - Does the director want a list of full-time employees only, or a list of full- and part-time employees together?
      - Does she want to include people who have worked for the company on a month-tomonth contractual basis over the past five years, or only regular, permanent employees?
      - Do the listed employees need to have worked for the organization for five years as of today, as of the date of the dinner, or as of some other cutoff date?
      - What about an employee who worked three years, took a two-year leave of absence, and has been back for three years?

- The programmer cannot make any of these decisions; the user (in this case, the Human Resources director) must address these questions.
- More decisions still might be required. For example:
  - What data should be included for each listed employee?
  - Should the list contain both first and last names? Social Security numbers? Phone numbers? Addresses?
  - Should the list be in alphabetical order? Employee ID number order? Length-of-service order? Some other order?
  - Should the employees be grouped by any criteria, such as department number or years of service?

- Several pieces of documentation are often provided to help the programmer understand the problem. **Documentation** consists of all the supporting paperwork for a program; it might include items such as original requests for the program from users, sample output, and descriptions of the data items available for input.
- Understanding the problem might be even more difficult if you are writing an app that you hope to market for mobile devices. Business developers are usually approached by a user with a need, but successful developers of mobile apps often try to identify needs that users aren't even aware of yet. For example, no one knew they wanted to play Angry Birds or leave messages on Facebook before those applications were developed.
- Mobile app developers also must consider a wider variety of user skills than programmers who develop applications that are used internally in a corporation. Mobile app developers must make sure their programs work with a range of screen sizes and hardware specifications because software competition is intense and the hardware changes quickly.
- Fully understanding the problem may be one of the most difficult aspects of programming. On any job, the description of what the user needs may be vague—worse yet, users may not really know what they want, and users who think they know frequently change their minds after seeing sample output.

- ❷ ***Planning the Logic***: The heart of the programming process lies in planning the program's logic. During this phase of the process, the programmer plans the steps of the program, deciding what steps to include and how to order them.
  - You can plan the solution to a problem in many ways. The two most common planning tools are **flowcharts** and **pseudocode**. Both tools involve writing the steps of the program in English.
  - Programmers refer to planning a program as "developing an algorithm." An **algorithm** (see the end of the section for the meaning) is the sequence of steps or rules you follow to solve a problem.
  - In addition to flowcharts and pseudocode, programmers use a variety of other tools to help in program development.
    - One such tool is an **IPO chart**, which delineates input, processing, and output tasks.

- Some object-oriented programmers also use **TOE charts**, which list **t**asks, **o**bjects, and **e**vents.
- **Storyboards** and **UML** are frequently used in interactive, object-oriented applications.

- The programmer shouldn't worry about the syntax of any particular language during the planning stage, but should focus on figuring out what sequence of events will lead from the available input to the desired output. Planning the logic includes thinking carefully about all the possible data values a program might encounter and how you want the program to handle each scenario.

- ❸ ***Coding the Program***: After the logic is developed, only then can the programmer write the source code for a program. Hundreds of programming languages are available. Programmers choose particular languages because some have built-in capabilities that make them more efficient than others at handling certain types of operations. Despite their differences, programming languages are quite alike in their basic capabilities—each can handle input operations, arithmetic processing, output operations, and other standard functions. The logic developed to solve a programming problem can be executed using any number of languages. Only after choosing a language must the programmer be concerned with proper punctuation and the correct spelling of commands—in other words, using the correct syntax.

- ❹ ***Use a Compiler or Interpreter to Translate the Program into Machine Language***: Even though there are many programming languages, each computer knows only one language—its machine language, which consists of 1s and 0s.
  - Computers understand machine language because they are made up of thousands of tiny electrical switches, each of which can be set in either the on or off state, which is represented by a 1 or 0, respectively.
  - Languages like Java or Visual Basic are available for programmers because someone has written a translator program (a compiler or interpreter) that changes the programmer's English-like **high-level programming language** into the **low-level machine language** that the computer understands.
  - When you learn the syntax of a programming language, the commands work on any machine on which the language software has been installed. However, your commands then are translated to machine language, which differs in various computer makes and models.
  - If you write a programming statement incorrectly, the translator program doesn't know how to proceed and issues an error message identifying a syntax error. Although making errors is never desirable, syntax errors are not a programmer's deepest concern, because the compiler or interpreter catches every syntax error and displays a message that notifies you of the problem. The computer will not execute a program that contains even one syntax error.
  - Typically, a programmer develops logic, writes the code, and compiles the program, receiving a list of syntax errors. The programmer then corrects the syntax errors and compiles the program again.

- Correcting the first set of errors frequently reveals new errors that originally were not apparent to the compiler.

- ❺ ***Testing the Program***: A program that is free of syntax errors is not necessarily free of logical errors.
  - A logical error results when you use syntactically correct statements but the program doe not produce the intended results.
  - Once a program is free of syntax errors, the programmer can test it—that is, execute it with some sample data to see whether the results are logically correct.
  - The process of finding and correcting program errors is called **debugging**. You debug a program by testing it using many sets of data.
  - Selecting test data is somewhat of an art in itself, and it should be done carefully
- ❻ ***Putting the Program into Production***: Once the program is thoroughly tested and debugged, it is ready for the organization to use. Putting the program into production might mean simply running the program once, if it was written to satisfy a user's request for a special list. However, the process might take months if the program will be run on a regular basis, or if it is one of a large system of programs being developed. Perhaps data-entry people must be trained to prepare the input for the new program, users must be trained to understand the output, or existing data in the company must be changed to an entirely new format to accommodate this program. **Conversion**, the entire set of actions an organization must take to switch over to using a new program or set of programs, can sometimes take months or years to accomplish.
- ❼ ***Maintaining the Program***: After programs are put into production, making necessary changes is called **maintenance**.
  - Maintenance can be required for many reasons: for example, because new tax rates are legislated, the format of an input file is altered, or the end user requires additional information not included in the original output specifications.
  - Frequently, your first programming job will require maintaining previously written programs. When you maintain the programs others have written, you will appreciate the effort the original programmer put into writing clear code, using reasonable variable names, and documenting his or her work.
  - When you make changes to existing programs, you repeat the development cycle. That is, you must understand the changes, then plan, code, translate, and test them before putting them into production. If a substantial number of program changes are required, the original program might be retired, and the program development cycle might be started for a new program.

Alıntı: Cesur Baransel, *Dünyada Önde Gelen Bilgisayarcılar*, *Türkiye Bilişim Ansiklopedisi*, pp. 333-343, Papatya Yayıcılık, İstanbul, 2006.

Harezmli Muhammed (780(?) - 850)

M.S. 830 yıllarında Abbasi halifesi Memun, Harezmli Muhammed'e aşağıdaki buyruğu verir:

"*Yalnızca en kolay ve en gerekli olan aritmetiği kullanarak, cebir ve mukabele kuralları ile hesaplama üzerine kısa bir kitap yaz. Öyle ki bu kitap, miras paylaşımında, zekât hesaplamada, ticari anlaşmalarda, kanalların kazılmasında, arazilerin ölçülmesinde ve benzer diğer işlemlerde kolaylıkla kullanılabilsin*".

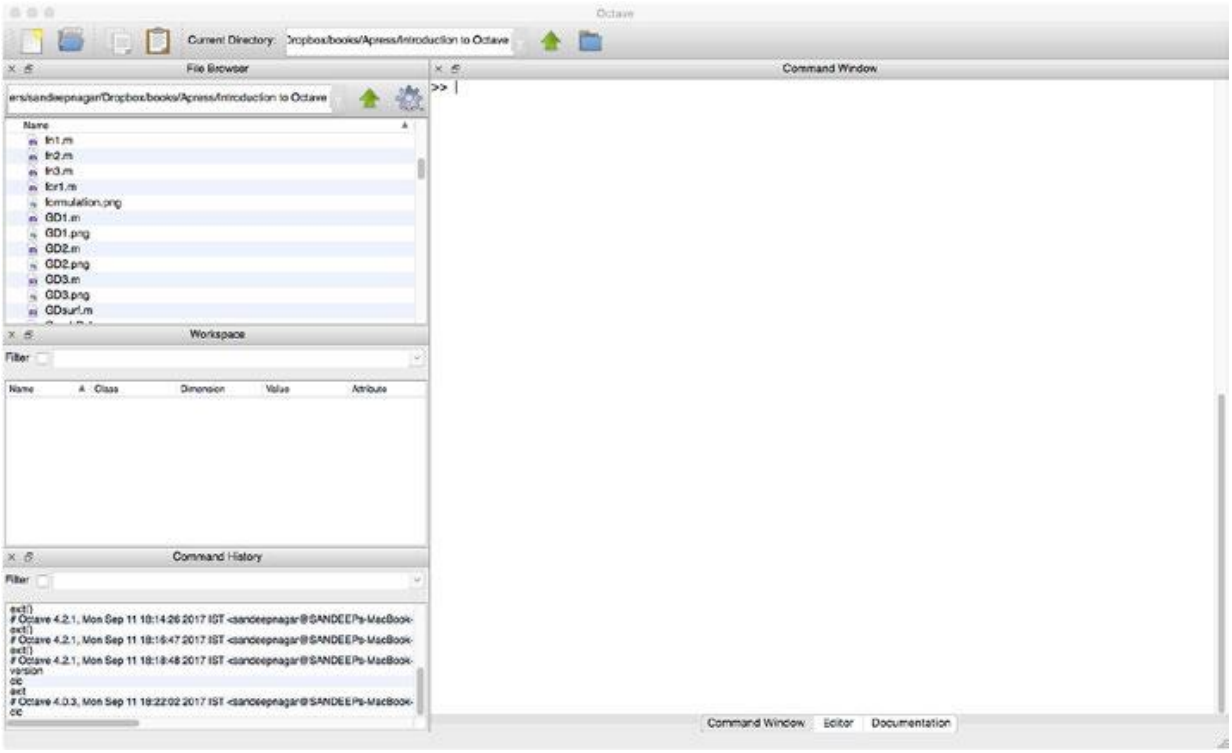
Harezmli bunun üzerine *Al Kitab-ül Muhtasar Fi Hesab Al-Cebr Ve'l Mukabele* adlı, onlu sayı sistemi ve denklemler yoluyla sorun çözme yöntemlerinin ilk kez anlatıldığı ünlü kitabını yazar. Tarihteki ilk cebir kitabı olarak kabul edilen bu kitap matematik tarihindeki en önemli yapıtlardan birisi olup İngilizce'deki "algebra" sözcüğü kitabın adındaki "al-cebr" sözcüğünden gelmektedir.

Kitabın matematik tarihindeki önemi özellikle matematik tarihi konulu kitaplarda ayrıntısı ile yer almaktadır. Ancak bu kitap bilgisayar bilimleri açısından da ayrı bir öneme sahiptir. Harezmli, kitabın geniş kitleler tarafından kullanılabilmesini sağlamak üzere günlük yaşamda çözülmesi gereken sorunları sistemli olarak sınıflamış ve her sorun türü için geçerli genel çözüm biçimleri oluşturmuştur. Her çözüm, sonlu sayıda ve iyi tanımlanmış adımlar içeren reçeteler biçiminde okuyucuya sunulmuş ve böylece matematikçi olmayan kişiler tarafından da etkinlikle kullanılabilmesi sağlanmıştır. Genel amaçlı bir çözümün, birbirini izleyen adımlar biçiminde ve her adımın çözüm reçetesini uygulayan kişinin en küçük bir duraksamaya düşmesine olanak bırakmayacak yalınlıkta ve kesinlikte tanımlanması yöntemi Harezmli'nin buluşudur. Bu yöntem Harezmli'nin adıyla anılmakta olup batı dillerinde "algorismus" "algorism", "algorithme", "algorithm" gibi biçimlerde kullanılmaktadır. Harezmli kitabını o zamanın bilim dili olan Arapça yazmıştı. Kitap "Liber Algebre et Almochabolae" başlığı ile 1183 yılında İspanya'da ilk kez Latince'ye çevrilirken, Arapça'da "Ebu Cafer Muhammed bin Musa Al-Harezmî" olarak geçen yazar adı, "Mahmed Moysi Alguarizmi" olarak aktarılmıştır. Yöntemin batı dillerindeki biçiminin kaynağı bu Latince tercümedeki "Alguarizmi" soyadıdır (Bu ad Avrupa'da çeşitli kaynaklarda *Al-Quarizmi*, *Al-Kuarithmi* ve *Algoritmi* olarak da geçer. Arapça sesbirimlerinin çevriyazımında hırıltılı *h*'nın *kh*, *g* ya da *q*'ye, peltek *z*'nin *th*'ye dönüşmesi sık görülür). Osmanlılar, Harezmli'nin yöntemini "Harezmiyet" yani "Harezmli Yolu" olarak adlandırmışlardır. Günümüz Türkçe'sinde ise bu kavrama karşılık olarak "Harezmli Yolu" ya da kısaca "Harezmlice" sözcüğü kullanılmaktadır.

Harezmli, Batı kaynaklarında çoğunlukla Arap ve bazen de Acem kökenli olarak anılmaktadır. Bazı kaynaklarda ise milliyetinden söz edilmeksizin İslam düşünürü ve matematikçisi olarak yer alır. Harezmli'nin yaşadığı dönem Abbasi imparatorluğunun (749-1258) en parlak dönemidir. Dönemin halifesi Memun (813-833) ve babası Harun Reşid doğu Hindistan'dan İspanya'ya dek uzanan ülke sınırları içerisindeki bilim adamlarını Bağdat'ta kurdukları "Dar-Ül Hikme (Bilgeler Evi)" adlı o zamanın araştırma merkezinde bir araya getirerek çalışmalarına destek olmuşlar ve burada çalışanlara maaş bağlamışlardı. Harezmli doğup büyüdüğü yer olan Harezm'den 810 yıllarında, otuz yaşlarında iken, çağrı üzerine Bağdat'a gelerek Dar-Ül Hikme'nin başına geçmiş ve 850 yılında ölümüne değin Bağdat'da yaşamıştır. Dar-Ül Hikme 800-1100 yılları arasında Dünya'nın bilim ve kültür merkezi olma özelliğini korumuştur. Ancak bundan önceki Emevi döneminde, yaklaşık 670-740 yılları arasındaki 70 yıllık dönemde Araplar ve Türkler arasında sürekli savaşlar yaşanmıştır. Buhara'nın 673'de kuşatılması ile başlayan ve Baykent, Talkan ve Curcan katliamlarına sahne olan bu dönemde Aral Gölü'nün altındaki Harezm bölgesi Emevilerin Horasan valisi Kuteybe tarafından işgal edilir. İşgal sırasında yaşanan talan ve yıkım ve yazılı dil bilenler dâhil tüm bilginlerin kılıçtan geçirildiği olaylar Harezmli ünlü Türk bilgini Biruni tarafından ayrıntısıyla yazılmıştır. Bu olaylardan ancak bir yüz yıl kadar sonra kendisine El-Harezmî (Harezmli kişi) dedirten, Türkçe çevirmenlik yapan ve Harezm'de doğup yetişen birisinin Arap kökenli olma olasılığı pek yoktur. Dolayısıyla cebirin babasının ve algoritma kavramının yaratıcısının Türk olma olasılığı çok daha yüksektir.

❸ *OCTAVE GUI*

- Octave's GUI looks quite similar to MATLAB's GUI. The left side has three panels:



  - ***File Browser***: You can browse through the files in a working directory and change the names. You can run an .m file by clicking on the file. The file opens in the Editor window and can be run from there.
  - ***Workspace***: It stores the variables names, values, and their properties like types and sizes. It is useful for developers to visualize the variables and their contents. The meaning of variables and their values, sizes, etc. is illustrated in subsequent chapters.
  - ***Command History****: It* stores the commands used in an Octave session. A command can be run by simply clicking it in command window. It is then executed at Octave command prompt.
- All three panes are optional and can be closed down for a session by clicking the cross sign in the upper-right.
- On the right side, there is a pane named Command Window. The bottom part of the Command window includes three tabs:
  - Command window
  - Editor
  - Documentation
- The ***Command window*** takes input one line at a time.
- The ***Editor window*** is used to write an ".m script file" that can then be executed.

- The **Documentation window** can be used to read documentation and seek help to learn more about commands. Octave has an extensive documentation that enables a beginner to learn Octave with nothing but a command line. It also helps an experienced user who can seek help in using less common commands.
- Sometimes you'll need to obtain a clear screen, which is what the clc command does.
- When you start an Octave session, you can work in an interactive session in the sense that the Octave prompt **>>** waits for you to input a command, which will be executed as soon as you press the Enter key at the end of command.
- The Octave command prompt presents a full-featured interactive command-line commonly called REPL (read-eval-print loop). The interactive shell of the Octave programming language is commonly called REPL because it:
  - ***R***eads what a user types
  - ***E***valuates what it reads
  - ***P***rints out the return value after evaluation
  - ***L***oops back and does it all over again

Working with Files

- Apart from working on Octave REPL, you can write multi-line programs using the built-in text editor in Octave and run the program.
- This can be created by typing **edit helloagain** at the Octave command prompt. A new file called helloagain.m will be created in a folder/directory in which the present session of Octave is running.
- Alternatively, the program can also be created in the editor by clicking on the lowermost part of the Command Window, which has an option named Editor. This opens a blank editor window in which the **helloagain.m** code can be written manually. You can then save the file using Ctrl+S.
- Note that all Octave script files are saved with an **.m** extension. You can open the existing file by navigating to the appropriate folder and choosing the file in the explorer.

```
disp("\nHello World!\n")
disp("Hello again\n")
```

Listing. The helloagain.m File

- The **\n** character in the string input is used to print a newline character, which simply adds a paragraph return and prints the next characters on a new line.
- The **disp()** function prints the string at the command prompt.
- You have many options for running an Octave file:
  - You can simply type the name of file (without the extension) at the Octave command prompt. For example, by typing **helloagain**.
  - From the **Editor** menu, you can click on **Run** and choose Save File and Run.

- You can also choose to click the given key combination. It prompts you to save the file if the script file is being run for the first time. You can choose to save the file at a chosen destination within the local computer's storage.

- In any case, the output is displayed at the command prompt, unless graphical output is directed to a graphical terminal.
- These two methods of working with Octave (using REPL and using files) each has its own merits and usage. Interactive sessions are best for quickly checking for a small part of complex code. Files are best with a project involving detailed calculations and are linked with one another to perform a computational task.

Using the Workspace

- A ***workspace*** is the space in the memory reserved for objects in the Octave session.
- All the objects used in calculations are displayed. This is usually placed as the second option in the left panel of the main Octave session window.
- The command clear clears all ***global*** and ***local*** variables in the workspace and makes it fresh, just as when an Octave session is initially launched.
- If the semicolon symbol **;** is used at the end of a command, the output is not displayed upon the execution of the command. This is useful when you expect too much output would be displayed. For example, when you are dealing with a multitude of data points, say a million data points, it would be pointless to invest time and computer memory in displaying them at Octave's command prompt.
- This feature can also be used within Octave scripts, when you don't want to print a particular output at the command prompt.

Octave as a Calculator

- In its simplest form, Octave works as a calculator with mathematical operators like **multiplication** (*), **division** (/), **addition** (+), **subtraction** (-), and **exponentiation** (^). The following code illustrates this behavior:

```
>> 3+5
   ans = 8
>> 3.0+5.0
   ans = 8
>> 3.1+5.0
   ans = 8.1000
>> 2-3
   ans = -1
>> 3.0*5
   ans = 15
>> 2/3
   ans = 0.66667
```

- When a command is entered at the Octave REPL command prompt **>>**, it is executed and an answer is displayed in the next line as **ans =**.

- Ans is a global variable that stores the value of the last executed expression. The commands written at Octave REPL are called expressions and are evaluated by REPL.
- A number of physical constants are defined as follows: **pi**, **e** (Euler's number), **i** and **j** (the imaginary number $\sqrt{-1}$), **inf** (infinity), and **NaN** (not a number, which results from undefined operations such as Inf/Inf).

```
>> pi
   ans = 3.1416
>> e
   ans = 2.7183
>> i
   ans = 0 + 1i
>> j
   ans = 0 + 1i
>> Inf/Inf
   ans = NaN
```

- A number of built-in mathematical functions exist in Octave. A few of the more common ones are:
  - Absolute value **abs()**
  - Natural logarithm **log()**
  - Base-10 logarithm **log10()**
  - Trigonometric functions **sin()**, **cos()**, and **tan()**. Arguments are taken in radians.
  - Inverse-trigonometric functions **asin()**, **acos()**, and **atan()**.

Using Variables

- Until now, we have been feeding numbers into Octave REPL with on-the-spot evaluation. Alternatively, you can designate **a memory location** where values are stored and this memory location can be known **by a name** for ease of usage. Such a programming construct is known as **a variable**.
- To store values temporarily, you use variables that store the value at a particular memory location and address it with a symbol or set of symbols (called ***strings***). For example, you can store the value of 1/10*pi as a variable a and then use it in an equation:

```
>> a=1/10*pi
   a = 0.31416
>> a^2 + 10* sqrt(a)
   ans = 5.7037
```

- The symbol **=** works as an **assignment operator** because it assigns the value on the right side to the variable name on the left side. Its behavior is markedly different than its mathematical counterpart (which checks the equality of its right side and left side).
- Multiple assignments can be performed using the **comma** (**,**) operator. Also if you do not want to produce results on-screen, you can suppress this by using the **;** operator. Try the following commands:

```
>> a1 = 1, a2 = 10, a3 = 100
>> a1 = 1, a2 = 10, a3 = 100;
>> a1 = 1; a2 = 10; a3 = 100;
```

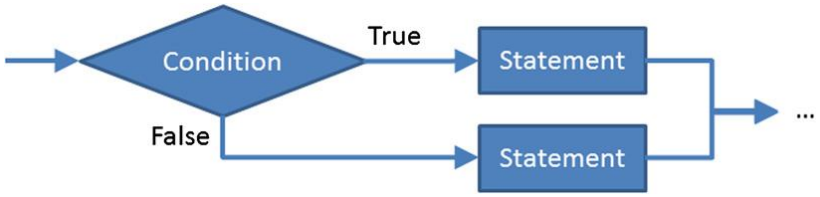
- While assigning data to a variable, it is important to understand that data can be defined as a variety of objects defined by a data type*,* as follows:
  - **Logical**: This type of data stores boolean values 1 or 0. Boolean values can be operated on by boolean operators, like AND, OR, XOR, etc.
  - **Char**: This type of data stores alphabetic characters and strings (groups of characters written in a sequence).
  - **Int8**, **int16**, **int32***, and* **int64**: This type of data is stored as integers within 8 bits, 10 bits, 32 bits, and 64 bits. The size of the integer is given by its bit counts. Both logical and char are 1 byte (8 bits) wide.
  - **uint8**, **uint16**, **uint32***, and* **uint64**: This type of data stores unsigned integer data in 8, 16, 32, and 64 bits.
  - **double**, **singl**e: This type of data is stored as double and single precision floating types, respectively. Decimal numbers are represented by floating point data types. Single precision occupies 4 bytes (32 bits) and double precision occupies 8 bytes (64 bits) to store the floating point numbers.

❹ CONTROL STRUCTURES

- The normal flow of control in an Octave script is *sequential*, i.e. each program statement is executed in sequence, one after the other.
- If all of our programs featured only sequential control flow they would be limited in their power. To write more complex and powerful programs we need to make use of programming constructs that alter this normal control flow.

If Statement

- One type of programming construct is the ***conditional statement***. The control flow of a conditional statement is illustrated below.



- In this example there are two possible paths through the program, involving execution of different statements. Which path is taken depends on the result of applying a ***test condition***.
- Enter these statements and save them as a script *m*-file:

```
a = input('Enter a number:');
if (a >= 0)
    root = sqrt(a);
   disp(['Square root = ' num2str(root)]);
else
   disp(['Number is negative and there is no square root']);
end
```

- Pay particular attention to the alternative ***paths of execution*** that the program can take depending on the result of the comparison (a >= 0). This comparison corresponds to the ***condition* box** in above figure.
- The two statements after the if will be executed only if the condition is true. The statement after the else will be executed if the condition is false. After one of these two paths has been taken control will resume immediately after the end statement.
- Note that the if and end statements are both compulsory and should always come in pairs.
- The else statement is optional and if omitted the program execution path will jump to after the end statement if the condition is false.
- A list of common comparison (or *relational*) operators:

Equal to:	==
Not equal to:	~=

Greater than:	>
Less than:	<
Greater than or equal to:	>=
Less than or equal to:	<=

- There are also the common ***logical operators*** *for* ***combining the results of different comparisons***.

~	NOT
&&	AND (scalar *short-circuit* operation)
&	AND (scalar/array operation)
||	OR (scalar *short-circuit* operation)
|	OR (scalar/array operation)

- Most of these operators are fairly intuitive. However, note the distinction between **&** and **&&** (and likewise between **|** and **||**). The & and | operators perform **AND** and **OR** operations respectively. They will evaluate the expressions on both sides of the operator and return a true or false value depending on whether both (&) or either (|) of them evaluated to true.
- These operators will work with either scalar (i.e. single value) logical expressions or array expressions (i.e. that evaluate to an array of true/false values). The only restriction is that Octave must be able to match the expressions on either side of the operator. Either both should be scalars, both should be arrays of the same size, or one should be an array and the other a scalar (More on arrays later).
- The **&&** and **||** operators perform the same AND and OR operations, but using what is known as a *short-circuiting behavior*. This means that, if the result of the overall AND/OR operation can be determined from the left-hand expression alone, then the right-hand expression will not be evaluated. For example, if the left-hand expression of an AND operation is false then the result of the AND will also be false, regardless of the value of the right-hand expression. Therefore, the advantage of short-circuiting is that unnecessary operations are not performed. However, *note that* **&&** *and* **||** *can only be used with scalar values, not arrays*.
- There are three different operators in the following condition: ==, && and >. In what order would MATLAB evaluate them? An expression containing multiple operators is evaluated using the rules for ***operator precedence***.

```
...
if gender == 'm' && calories > 10+2*75
...
```

- Read about ***operator precedence*** using help and doc commands.
- Brackets have the highest precedence. It is a good idea to include them to make our code easier to understand and less prone to errors.
- Sometimes we may have many if statements which all use conditions based on the same variable. It is not incorrect to use if statements in such cases, but it can lead to a large number of consecutive if statements in our code, making it harder to read and more prone to errors. In this case, it is preferable to use a switch statement.
- The switch statement offers an easy way of writing code where ***the same variable*** needs to be checked against a number of different values.

```
switch day
        case 1
           day_name = 'Monday';
        case 2
           day_name = 'Tuesday';
        case 3
           day_name = 'Wednesday';
        case 4
           day_name = 'Thursday';
        case 5
           day_name = 'Friday';
        case 6
           day_name = 'Saturday';
        case 7
           day_name = 'Sunday';
        otherwise
           day_name = 'Unknown';
 end
```

- Note that the switch statement is used only for equality tests – we cannot use it for other types of comparison (e.g. >, <, etc.).
- In the above example the switch expression was compared to a single value in each case. It is possible to compare the expression to multiple values by enclosing them within curly brackets and separating them by commas. The corresponding statements are executed if *any* of the values are matched. This is equivalent to an if statement with multiple equality tests combined using a || operator.

```
switch day
       case {1,2,3,4,5}
          day_name = 'Weekday';
       case {6,7}
          day_name = 'Weekend';
       otherwise
          day_name = 'Unknown';
 end
```

Using If Statements Effectively

- First, we will implement the switch command using if-else statements only.
- We will provide multiple solutions for this problem, discussing the efficiency of each one.
- *Solution a:*

```
a = input('Enter a number for day:');
strA = num2str(a);
if (a == 1) disp(['Day  ' strA ' is weekday']); end
if (a == 2) disp(['Day  ' strA ' is weekday']); end
if (a == 3) disp(['Day  ' strA ' is weekday']); end
if (a == 4) disp(['Day  ' strA ' is weekday']); end
if (a == 5) disp(['Day  ' strA ' is weekday']); end

if (a == 6) disp(['Day  ' strA ' is weekend']); end
if (a == 7) disp(['Day  ' strA ' is weekend']); end
```

- Although this program works correctly for proper values, it does not provide an error message if the user enters a number like -1 or 9. The following solution fixes this problem. Note that we cannot say 'Unknown day' using only a simple comparison.
- *Solution b:* ***Solving the "unknown" problem***

```
a = input('Enter a number for day:');
strA = num2str(a);

sB = ['Day  ' strA ' is unknown‘];

if (a == 1) sB = (['Day  ' strA ' is weekday']); end
if (a == 2) sB = (['Day  ' strA ' is weekday']); end
if (a == 3) sB = (['Day  ' strA ' is weekday']); end
if (a == 4) sB = (['Day  ' strA ' is weekday']); end
if (a == 5) sB = (['Day  ' strA ' is weekday']); end

if (a == 6) sB = (['Day  ' strA ' is weekend']); end
if (a == 7) sB = (['Day  ' strA ' is weekend']); end

sB
```

- This program has a different logic from the previous one. It does not display a message immediately after making comparisons. Instead, it prepares a message whis says the “Day is unknown”. If the users enters a valid value, this message is updated. Otherwise, the “Day is unknown” message stays as it is. At the end of the program sB is displayed, either with its original or updated content, depending on the value that the user has entered.
- The following solutions uses if statements with multiple equality tests combined using logical OR (||) and logical AND (&&) operators.
- *Solution c:*

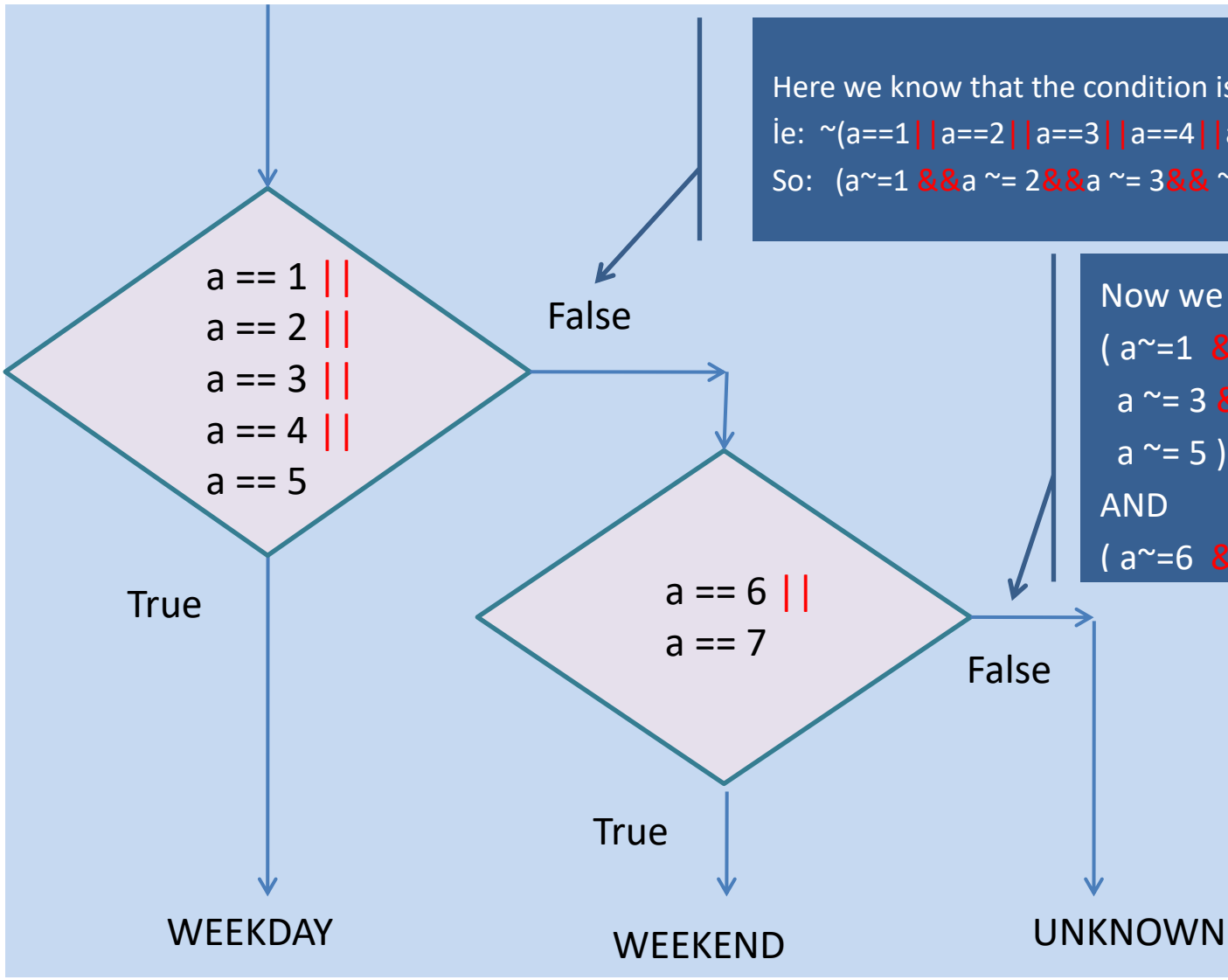
```
a = input('Enter a number for day:');
strA = num2str(a);

if (a == 1 || a == 2 || a == 3 || a == 4 || a == 5)
    disp(['Day  ' strA ' is weekday']);
end

if (a == 6 || a == 7)
    disp(['Day  ' strA ' is weekend']);
end

if (a ~= 1 && a ~= 2 && a ~= 3 && a ~= 4 &&
    a ~= 5 && a ~= 6 && a ~= 7)
    disp(['Day  ' strA ' is unknown']);
end
```

- This solution is better since it reduces the number of if statements down to three. However the condition of the last if statement is unnecessarily cumbersome. A much better alternative exists, as seen in the following solution.

- *Solution d:*

```
a = input('Enter a number for day:');
strA = num2str(a);

if (a == 1 || a == 2 || a == 3 || a == 4 || a == 5)
    disp(['Day  ' strA ' is weekday']);
else
   if (a == 6 || a == 7)
       disp(['Day  ' strA ' is weekend']);
   else
       disp(['Day  ' strA ' is unknown']);
   end
end
```

- This solution offers the following improvements over the previos ones.
  - Faster execution with less number of comparisons executed, since number of if stetments are reduced from 7 to 3.
  - The long logical condition for the"unknown day " is no longer necessary and therefore eliminated. Below, we draw the flowchart for this solution to see how elimination become possible.



- Now we introduce the recommended solution.
- This solution uses the so-called **elseif statement** **(not else if)**.
- The elseif statement is easy to understand and is more similar to switch statement. However, note that,
  - The alternative cases can have complex conditions and each if does not have to have its own end statement. In other words, an **elseif** statement has a single **else** and a single **end** part.
  - The last (*and only*) **else** corresponds to **otherwise** in the *switch* statement.

- *Solution e: The recommended solution*

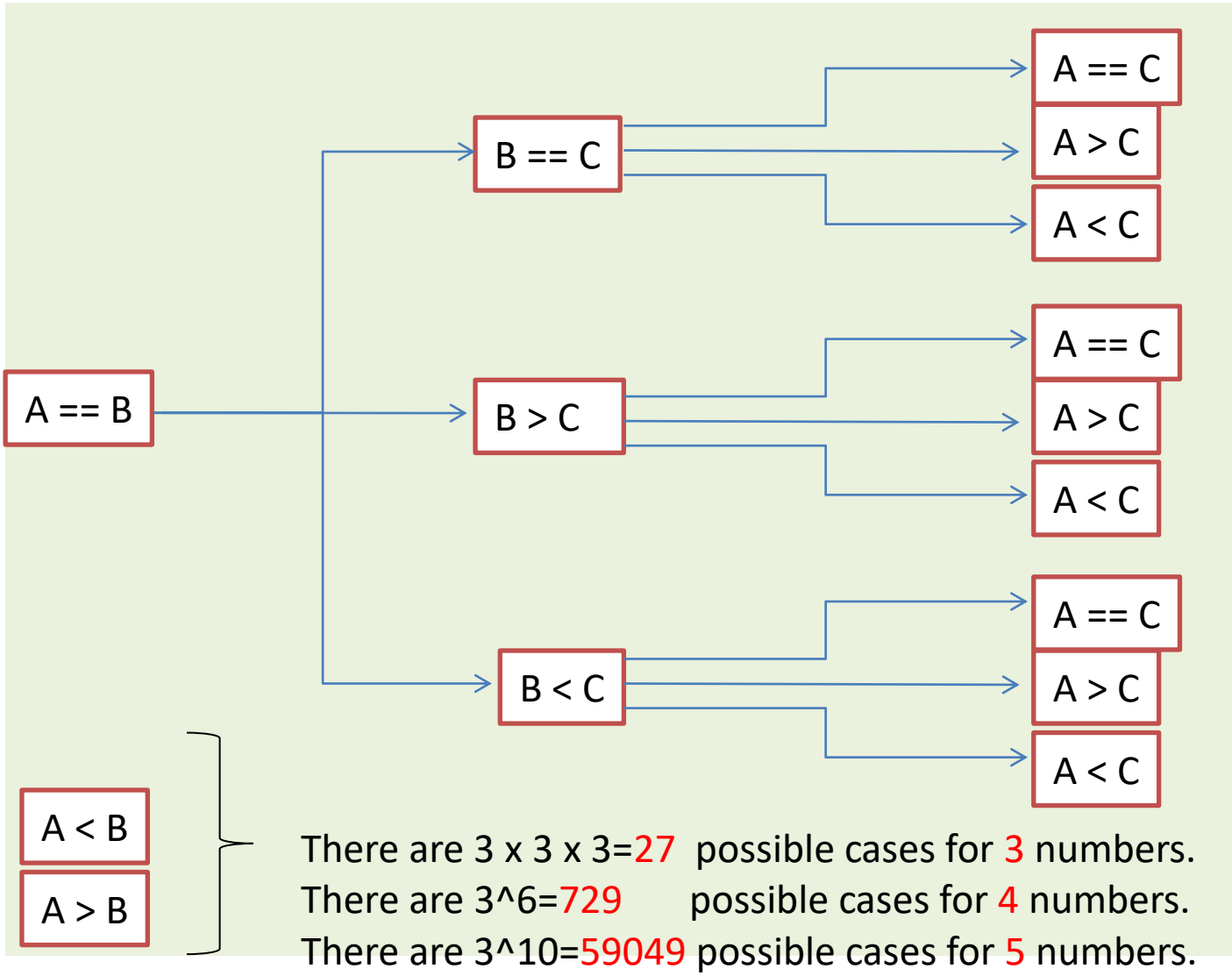
```
a = input('Enter a number for day:');
strA = num2str(a);

if (a == 1 || a == 2 || a == 3 || a == 4 || a == 5)
    disp(['Day  ' strA ' is weekday']);
elseif (a == 6 || a == 7)
    disp(['Day  ' strA ' is weekend']);
```

```
else
    disp(['Day  ' strA ' is unknown']);
end
```

- You should be careful when writing conditions for the if-statements. You may find yourself writing conditions impossible to satisfy, i.e., conditions that **never** can be **true**. The following exercise illustrates this point.
- Assume that you write a program where user enters two numbers and program selects the greatest one. This is a fairly easy program and can be coded as follows:

```
n1 = input('Enter the first  number:');
n2 = input('Enter the second number:');

if (n1 == n2)
    disp('Numbers are equal');
end

if (n1 > n2)
    disp('The first number is greater than the second number');
end

if (n1 < n2)
    disp('The first number is less than the second number');
end
```

- The logic is easy to explain: There are only 3 possible cases for 2 numbers: Numbers can be equal to each other, or the first number can be greater than the second number or the first number can be less than the first number. Furthermore, these cases are **mutually exclusive**, meaning that only one of them can be true for a given pair of numbers.
- In the program, each case is handled by a seperate if statement.
- Now, repeat the exercise for 3 numbers. How many possible cases are there? For 4 numbers? For 5 numbers?
- The most important question is this: When writing the program, should you simply enumerate all possible cases first and then write an if statetement for each one?
- The short answer is no. Such a blind approach can easily lead to a situation where a lot of unnecessarily written if statements sitting there doing nothing but cluttering your code and slowing down your program.
- Consider the case of three numbers. Without too much thinking, we can use a simple logic: Since any two number can be related in 3 possible ways (>, ==,<), 3 numbers can be related in 3^3=27 possible ways. This logic can be drawn partially as follows:



- We have drawn only 1/3 of the possible cases, which is sufficient to show that the above logic is simply wrong! Let's write the code for the first two branches in the above diagram:

```
A = input('Enter the first  number:');
B = input('Enter the second number:');
C = input('Enter the third  number:');

if (A == B && B == C && A == C)
    disp('Numbers are equal to each other');
end

if (A == B && B == C && A > C)
    disp(???) ;
end
```

- The second if can never evaluate to true. Why? Consider the case where A is equal to B. Then, if B is equal to C, A and C must be equal to each other (because A and C both are equal to the same number B). It is clear that, in that case (A > C) can never be true. Since all conditions are tied together with logical AND operator, three numbers that can make the whole expression true simply **do not exist**.

- Here, an impossible case is coded. The programmer has written unnecessary code, the computer check this statement every time it runs and program becomes larger. Avoid these kind of mistakes in your programs and by learning that simply handling combinatorial cases one by one does not necessarily yield logically correct results.
- Think well before writing if-else conditions.
  - You may be writing impossible to satisfy conditions.
  - You may be missing some critically important conditions.
  - You may be writing contradictory conditions in different places in code.
- Do the following self-study excersies to test your understanding of the subject:
  - Exercise 1: User enters 3 numbers from keyboard. **Sort** these numbers in descending order and solve the "find the greates number" problem after the sort operation.
  - Exercise 2: Write a program that displays the smallest of five input values that may include duplicate values (e.g., 6, 4, 8, 6, 7).
  - Exercise 3: Write a program that inputs a number between 1 and 10 and displays the number with the appropriate two-letter ending (e.g., 1st, 2nd, 3rd, 4th, 5^{th}…).
  - Exercise 4: Write a program that will receive three test scores as input. The program should determine and display their average and the appropriate letter grade based on the average. The letter grade should be determined using a standard 10-point scale (A 90–100; B 80–89.999; C 70–79.999, F 70–69.999)

Excersises with solutions:

Question 1

The ticket price for a concert varies according to the age of the customer.

- When the person is under 16, the charge is 70 TRY;
- When the person is 65 or over, the charge is 50 TRY;
- All others are charged 100 TRY.

Given the age of the customer, calculate the ticket price.

a) Draw the flowchart of the solution.
b) Write the Octave code.

Solution 1

First, try to visualize how your program will work.

It should work as follows if the age of the customer is 15:

```
Enter the age of the customer: 15
The ticket price for this customer is 70 TRY
```

It should work as follows if the age of the customer is 67:

```
Enter the age of the customer: 67
The ticket price for this customer is 50 TRY
```

It should work as follows if the age of the customer is 42:

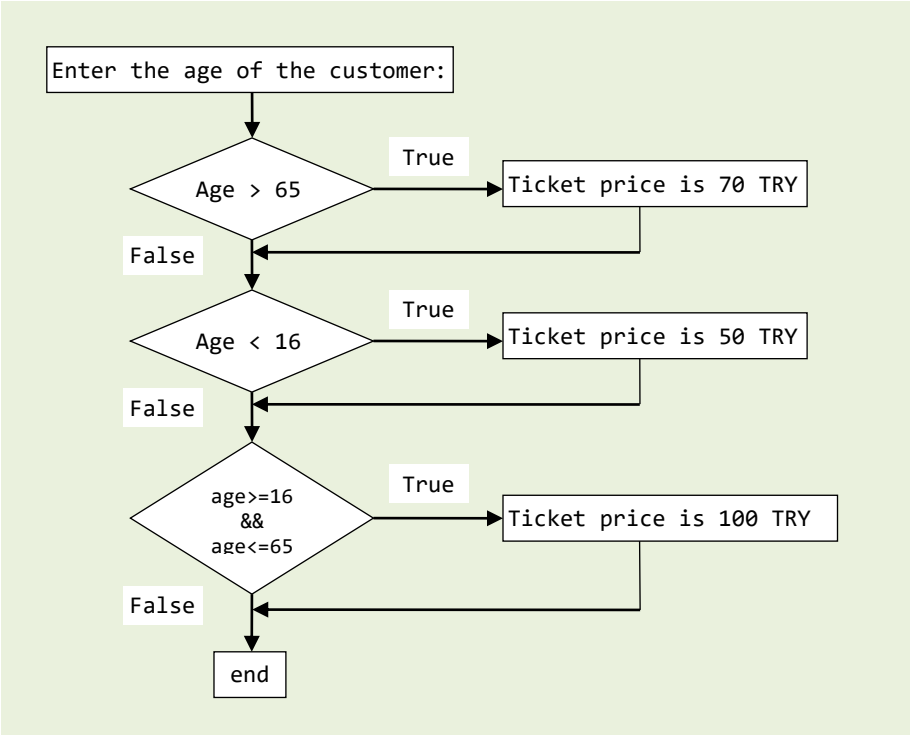
```
Enter the age of the customer: 42
The ticket price for this customer is 100 TRY
```

It is clear that the program should read *the age of the customer* from the keyboard and display *the proper ticket price*. Consequently, the program has one input (*the age of the customer*) and one output (*the ticket price*).

- To read the input, you use the **`input`** command in Octave.
- To display the output, you use the **`disp`** command in Octave.

In order to decide what the ticket price should be, we must consider three cases. These cases are already given in the problem definition, as seen in the following figure.

The simplest solution is to write three if statements, one for each case. The flowchart is shown below:



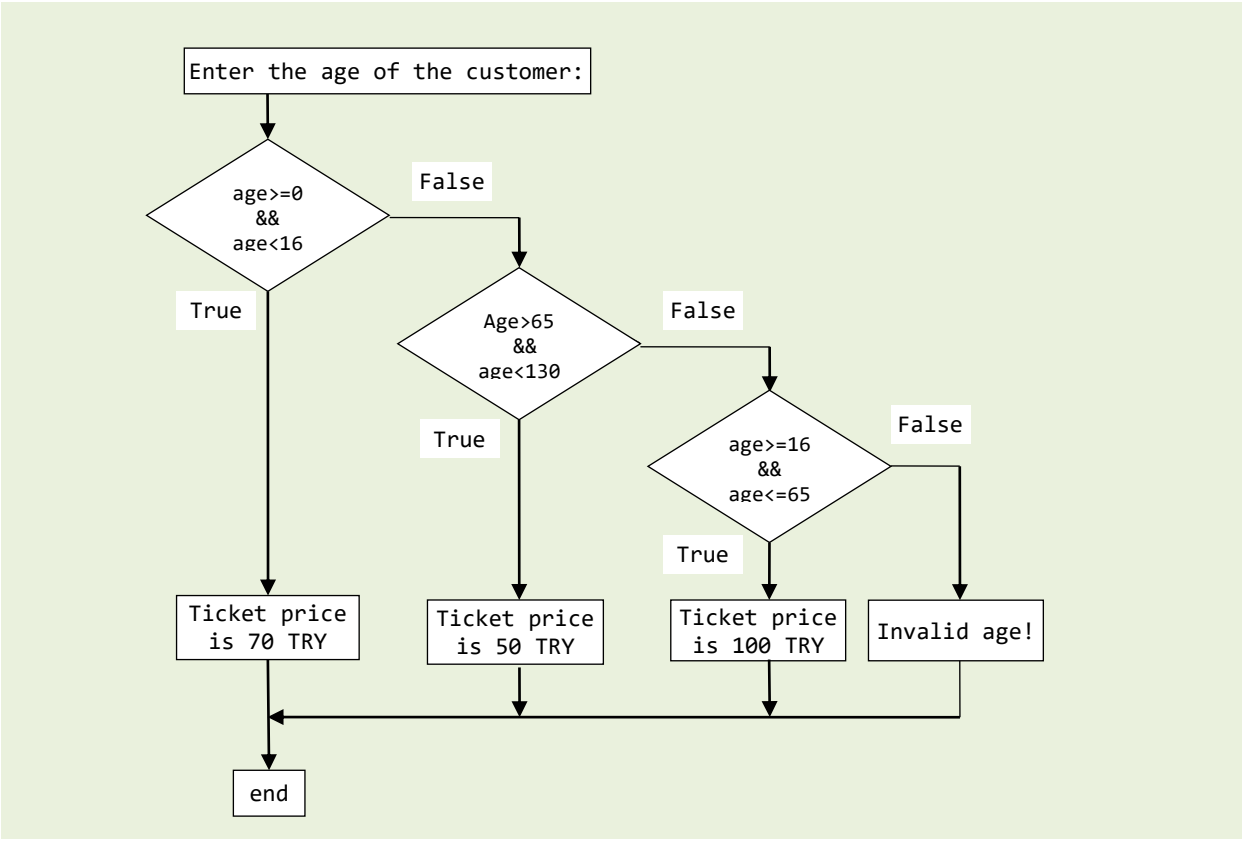
The Octave code for this flowchart is:

```
age = input('Enter the age of the customer:');

if (age < 16)
    disp('Ticket price for this customer is 70 TRY');
end

if (age > 65)
    disp('Ticket price for this customer is 50 TRY');
end

if (age >= 16 && age <= 65)
    disp('Ticket price for this customer is 100TRY');
end
```

As discussed previously, this is not the recommended solution. Also, suppose that the user enters -5 as the age, or 5000? What should the program do? The following code is a better solution. It only fails when a person older than 130 shows up.

```
age = input('Enter the age of the customer:');

if     (age < 16 && age >= 0)
       disp('Ticket price for this customer is 70 TRY');
elseif (age > 65 && age < 130)
       disp('Ticket price for this customer is 50 TRY');
elseif (age >= 16 && age <= 65)
       disp('Ticket price for this customer is 100 TRY');
else
       disp('Invalid age information is entered');
end
```

Note that, the flowchart for this solution will be different from the one above. Examine the following flowchart and be sure that you understand how the redundant execution of if statements are eliminated.



Here, the program receives an input value and decides into which predefined interval (0–16, 16–65, 65–130) this value falls into, before taking the necessary action. This mechanism can be used for creating many, seemingly different problems. Do not be fooled by the story presented in the problem. In fact, all these problems are <u>the same</u>. Two of such seemingly different but essentially the same questions are presented in questions 2 and 3.

Question 2

A company calculates the commission rate for a salesperson, given the amount of sales.

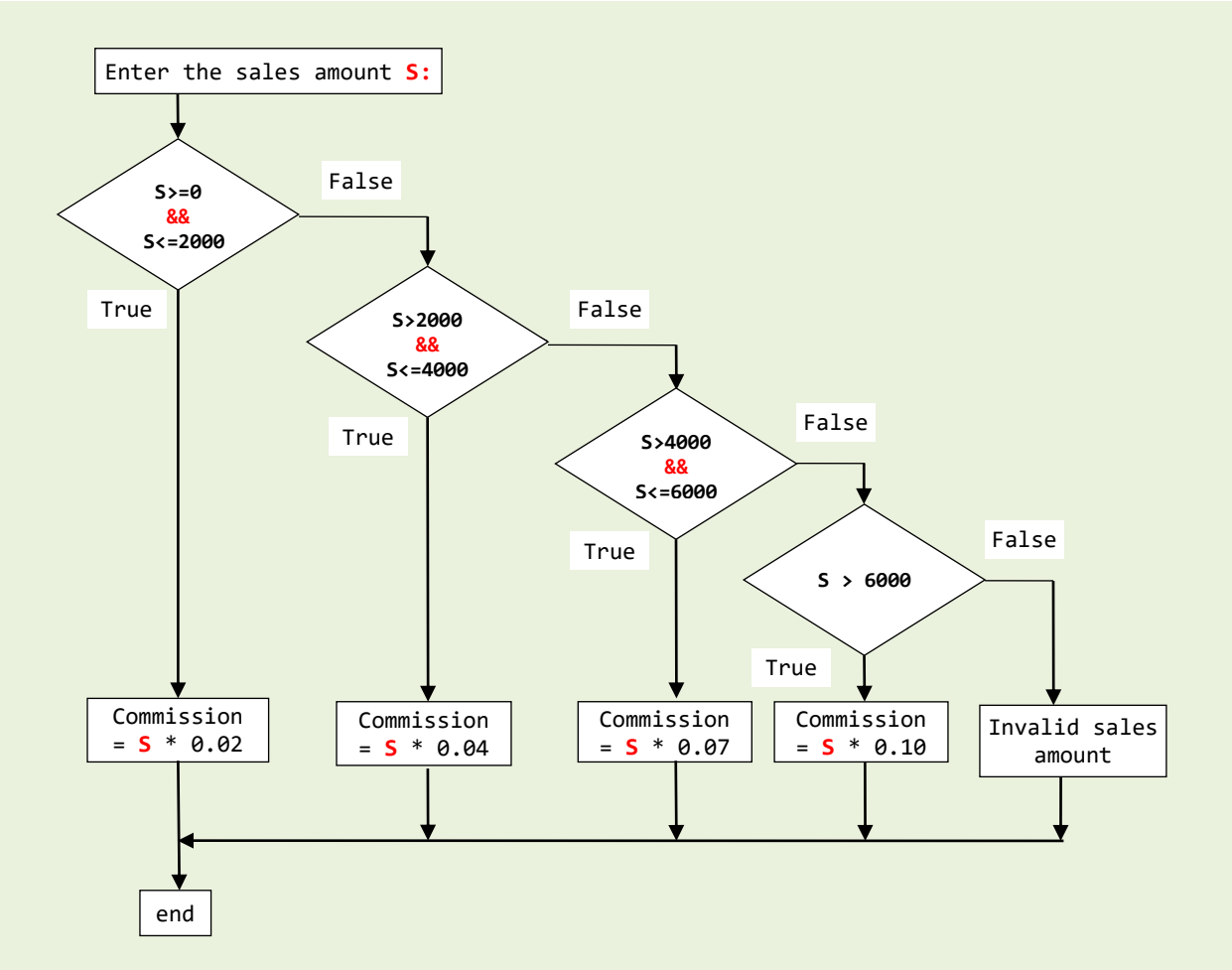
- When the amount of sales is less than or equal to 2,000 TRY, the commission is 2%.
- When the amount of sales is more than 2,000 TRY and less than or equal to 4,000 TRY, the commission is 4%.
- When the amount of sales is more than 4,000 TRY and less than or equal to 6,000 TRY, the commission is 7%.
- When the amount of sales is more than 6,000 TRY, the commission is 10%.

Given the amount of sales, calculate the commission.

a. Draw the flowchart of the solution.
b. Write the Octave code.

Solution 2

Draw the flowchart and see that how it is similar to the flowchart of the previous problem:



And the Octave code is:

```
S = input('Enter the sales amount:');

if      (S >= 0 && S <= 2000)
        disp(['Comission is ' num2str(S*0.02)]);
elseif (S > 2000 && S <= 4000)
        disp(['Comission is ' num2str(S*0.04)]);
elseif (S > 4000 && S <= 6000)
        disp(['Comission is ' num2str(S*0.07)]);
elseif (S > 6000)
        disp(['Comission is ' num2str(S*0.10)]);
else
        disp('Invalid sales amount is entered');
end
```

Question 3

An admission charge for a music concert varies according to the age of the person. Develop a solution to print the ticket charge given the age of the person. The charges are as follows:

- Over 55: 10.00 TRY
- 21–54: 20.00 TRY
- 13–20: 15.00 TRY
- 3–12: 5.00 TRY
- Under 3: Free

a) Draw the flowchart of the solution (10 points).
b) Write the Octave code.

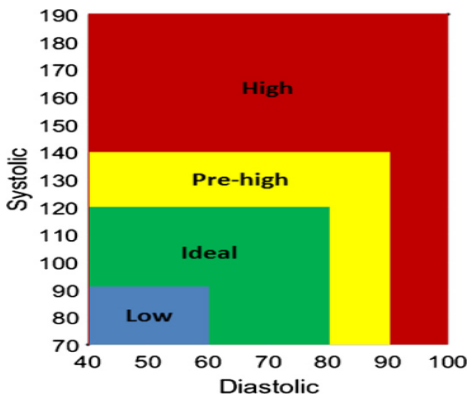
Solution 3

The Octave code is provided below. Draw the flowchart yourself.

```
age = input('Enter the age of the customer:');

if     (age >= 0 && age < 3)
        disp('Ticket for this customer is free');
elseif (age >= 3 && age < 13)
        disp('Ticket price for this customer is 5 TRY');
elseif (age >= 13 && age < 21)
        disp('Ticket price for this customer is 15 TRY');
elseif (age >= 21 && age < 55)
        disp('Ticket price for this customer is 20 TRY');
elseif (age >= 55 && age < 130)
        disp('Ticket price for this customer is 10 TRY');
else
        disp('Invalid age information is entered');
end
```

Question 4

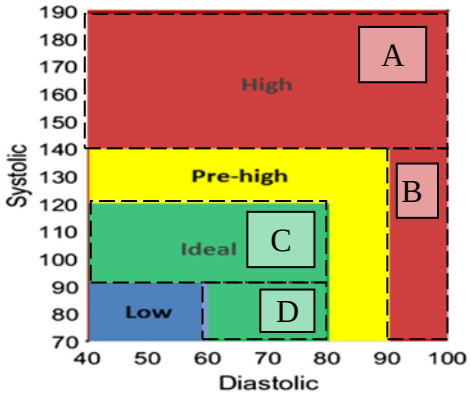
The figure below shows how blood pressure can be classified based on the diastolic and systolic pressures. Write a MATLAB script *m*-file to display a message indicating the classification based on the values of two variables representing the diastolic and systolic pressures. The two blood pressure values should be read in from the keyboard.



Solution 4

This problem differs from those above in two respects:

- It receives two inputs (diastolic pressure and systolic pressure).
- The decision intervals are more complicated to express programmatically.



For example consider the High region.

- When systolic pressure is between 140 and 190 we don't need to consider the value of the Diastolic pressure. This is the region A in the graph.
- In region B, the blood pressure is also high.

So, we need to expess these two regions separately and tied them together using a logical OR operator (i.e., if values are in **(**(region A) || (region B)**)**, then the blood pressure is high).

Following the same logic for the other regions as well, the Octave code can be written as follows.

```
s = input('Enter the systolic pressure:');
d = input('Enter the diastolic pressure:');

if ( (s > 140 && s <= 190)  ||                          %this is region A
     (d > 90  && d <= 100)     )                        %this is region B
     disp('The blood pressure is HIGH');
end

if ( (s > 120 && s <= 140 && d > 40 && d <= 90)  ||
     (s >  70 && s <= 120 && d > 80 && d <= 90)     )
     disp('The blood pressure is PRE-HIGH');
end

if ( (s > 90 && s <= 120 && d > 40 && d <= 80)  ||    %this is region C
     (s > 70 && s <   90 && d > 60 && d <= 80)     )  %this is region D
     disp('The blood pressure is IDEAL');
end

if (s >= 70 && s <= 90 && d >= 40 && d <= 60)
    disp('The blood pressure is LOW');
end
```

Question 5

A company has two types of workers: Workers who get paid according to the number of hours they work each week (H-type) and workers that work for a predefined salary (S-type).

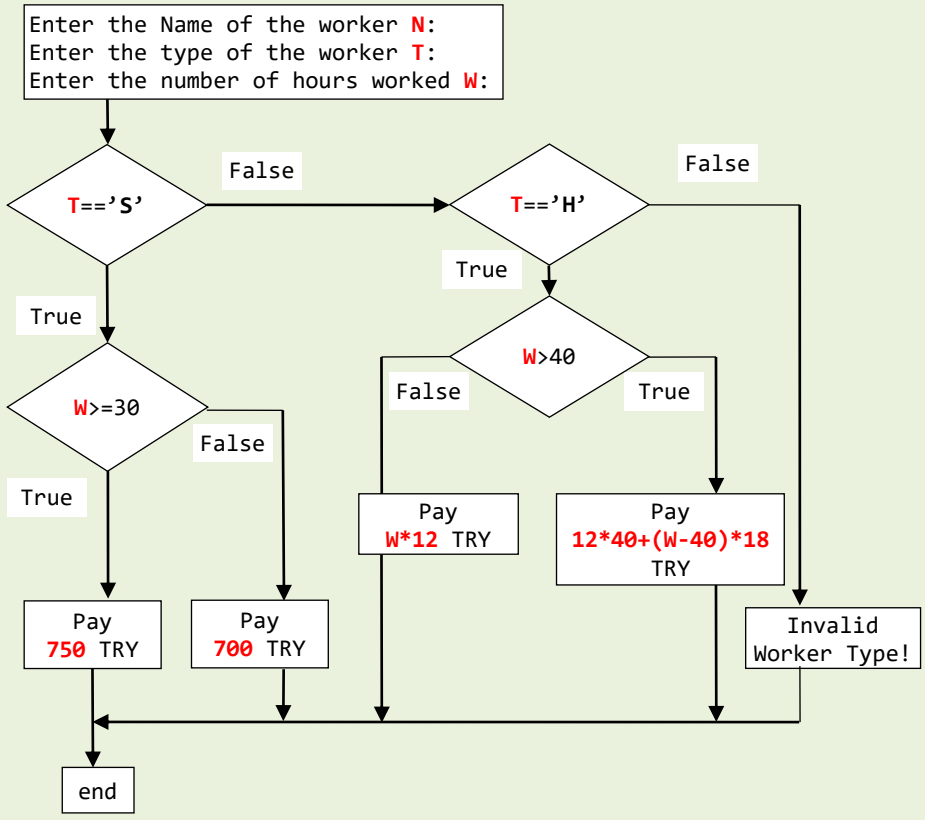
- Weekly pay for S-type workers is 750 TRY if they work 40 or more hours within the week. If they work less than 40 hours they still get 750 TRY. However, if they work less than 30 hours they will get 700 TRY.
- H-type workers are paid 12 TRY per hour up to 40 hours. For more than 40 hours, they are paid 18 TRY for each extra hour.

Write a solution to compute the weekly pay for a given worker. The data is entered from the keyboard in the following format:

Name of the Worker	Type of the Worker	Hours Worked:
Ahmet Genç	H	35
Mehmet Bal	S	42

a) Draw the flowchart of the solution.
b) Write the Octave code.

Solution 5



The flowchart is given above and Octave code can be written as follows:

```
N = input('Enter the Name of the worker: ', 's');
T = input('Enter the type of the worker: ', 's');
W = input('Enter the number of hours worked: ');

if (T=='S' && W >= 30)
    disp(['Pay ' N ' 750 TRY']);
end

if (T=='S' && W < 30)
    disp(['Pay ' N ' 700 TRY']);
end

if (T=='H' && W <= 40)
    disp(['Pay ' N ' ' num2str(W*12) ' TRY']);
end

if (T=='H' && W > 40)
    disp(['Pay ' N ' ' num2str(40*12+(W-40)*18) ' TRY']);
end

if (T~='H' && T~='S')
    disp('Invalid worker type!');
end
```

Question 6

A hotel has a pricing policy as follows:

- The cost of a room is 100 TRY.
- If the customer is staying on company business, there is a 20% discount.
- If the customer is over 60 years of age, there is a 15% discount.
- If eligible, a customer does not receive both discounts, only the larger one.

Given the above rules, print the cost of the room.

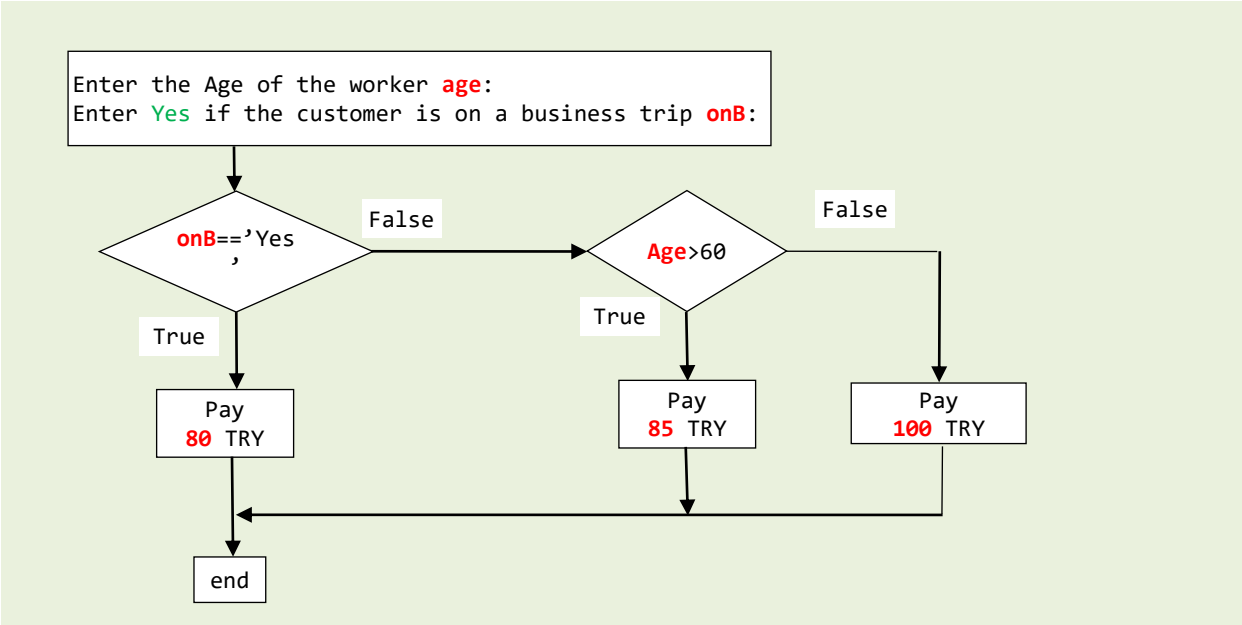
a) Draw the flowchart of the solution.
b) Write the Octave code.

Solution 6

In this problem, the order of logical tests is important. The rules say that there are two types of discount in this hotel, one for people who are on business trip (20%), and the other for people older than 60 (15%). The last rule states that anyone who is both older than 60 and on a business trip does not receive both discounts, only the larger one (that is 20%).

In other words, for people on business trip, the age is not important: they always receive the 20% discount, and nothing else. You should check the age only when the 20% discount is not possible.

So, the proper flowchart should be drawn as follows:



The Octave code is:

```
age = input('Enter the age of the customer:');
onB = input('Enter YES if the customer is on business trip:', 's');

if     (onB == 'YES')
       disp('Customer should pay 80 TRY');
elseif (age > 60)
       disp('Customer should pay 85 TRY');
else
       disp('Customer should pay 100 TRY');
end
```

Question 7

Ayşe needs to buy a present for her best friend.

- She can buy it online or she can travel by car to buy at the store.
- The online cost of the item may be different from the cost at the store.
- Ayşe's car uses 0.25 TRY worth of gas per Km.
- She is not sure which would be less expensive considering shipping & handling costs to buy online and gas costs to travel to the store.
- Write a solution to tell Ayşe which would be the best way to buy the present.

a) Draw the flowchart of the solution.
b) Write the Octave code.

Solution 7

This problem is extremely simple. Consider the following two examples:

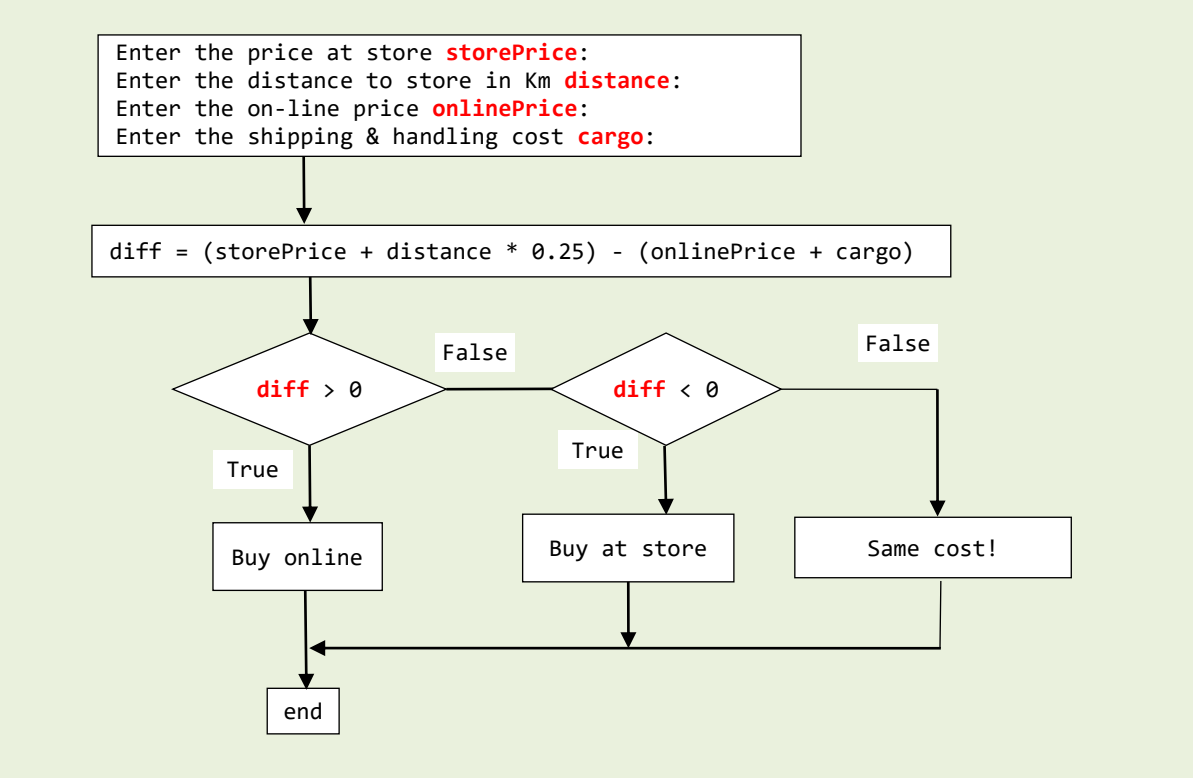
a) You will buy a smart phone. The phone is sold at 1000 TRY at a store 4 Km away. The same phone is also sold at an on-line store for 900 TRY with 10 TRY for cargo expenses. Then, the cost of buying the phone at the store is (1000+4*0.25=1001 TRY). Compare this cost to the cost of buying the phone on-line which is (900+10=910 TRY). In this case, It is clear that you should buy it on-line.

b) You will buy a roller pen. The pen is sold at 10 TRY at a store 2 Km away. The same pen is also sold at an on-line store for 9 TRY with 5 TRY for cargo expenses. Then, the cost of buying the pen at the store is (10+2*0.25=10,50 TRY). Compare this cost to the cost of buying the phone on-line which is (9+5=14 TRY). In this case, it is clear that you should buy it at the store.

Examples show that to make the decision, the program needs for values as input:

1. The cost of the item at the store
2. The distance to the store in Km
3. The cost of the item on-line
4. The shipping and handling cost (i.e., cargo expenses)

The flowchart:



The code:

```
storePrice  = input('Enter the price at store: ');
distance    = input('Enter the distance to store in Km: ');
onlinePrice = input('Enter the on-line price: ');
cargo       = input('Enter the shipping & handling cost: ');

diff = (storePrice + distance * 0.25) - (onlinePrice + cargo) ;

if     (diff > 0)
       disp('Buy online');
elseif (diff < 0)
       disp('Buy at the store');
else
       disp('No difference, buy wherever you want!');
end
```