Computing

The study of systematic processes that describe and transform *information*: their theory, analysis, design, efficiency, implementation, and application.

Fundamental Question of Computing:

What can be automated?

Computer Science vs. Mathematics

— *Mathematics* is concerned mainly with *declarative knowledge*: identifying the characteristics of and relationships among distinct abstract notions.

— *Computer Science* is concerned mainly with *imperative knowledge*: how to construct effective methods for manipulating information.

Basics

Information is a *primitive*; i.e., it cannot be defined in terms of simpler concepts. "Facts" and "ideas" are dependent on context.

Data is information represented with symbols; e.g., numbers, words, and images.

An *algorithm* is a finite sequence of unambiguous instructions for the completion of a specific task in finitely many steps.

A *program* is the implementation of an algorithm suitable for execution by a computing machine. Programs are also called *applications* or *software*.

Digital vs. Analog E.g., an analog clock (big hand and little hand) represents time continuously. A digital clock represents time discretely.

An analog machine stores and represents information in a continuous range of values.

— No inherent limit on accuracy other than cost.

— May be difficult or expensive to convert data from one medium to another.

A digital machine stores and represents information in a finite set of discrete values.

— Accuracy limit is clear from the output.

— Relatively easy and cheap to convert data from one medium to another.

Abstraction

The extraction of the essential, defining characteristics of an object from the collection of all its common instances. An abstraction is a simplified model of something.

E.g., one abstraction for a car is "something with wheels that you can drive." Other abstractions are more appropriate in other contexts: design, marketing, maintainance and repair, safety testing, etc.

Levels of Abstraction

A *high level* of abstraction is a superficial view.

— The "big picture," the "forest."

— Ignore small-scale details.

— Concentrate on relevant aggregate properties.

A *low level* of abstraction is a precise, technical view.

— The "details," the "trees."

— Ignore large-scale aggregate properties.

— Concentrate on localized primitives.

Each level is useful in its own context.

— A *hierarchy of intermediate levels* extends from the superficial to the minute.

— Imagine flying in an airplane over the countryside. The lower the plane is, the more detail you see, and the more restricted in scope the view is.

Some Traditional Applications of Computing

Data Archiving and Database Management

Scheduling and Operations Research —manufacturing and distribution, *logistics*

Mathematical and Scientific Computation and Simulation — cryptography, geology, meteorology, astronomy, chemistry

Systems Engineering and Design — weapons systems, space exploration, aircraft design and maintenance, expert systems

Text Processing — document preparation and analysis

Graphics and Visualization

Some Emerging Applications of Computing

Artificial Intelligence — speech recognition, robotics, cybernetics, computer vision

Electronic Design Automation (EDA), e.g., in VLSICAD

Bioinformatics — genomics, proteomics, drug design

Distributed networks of semi-autonomous agents

Telecommunications

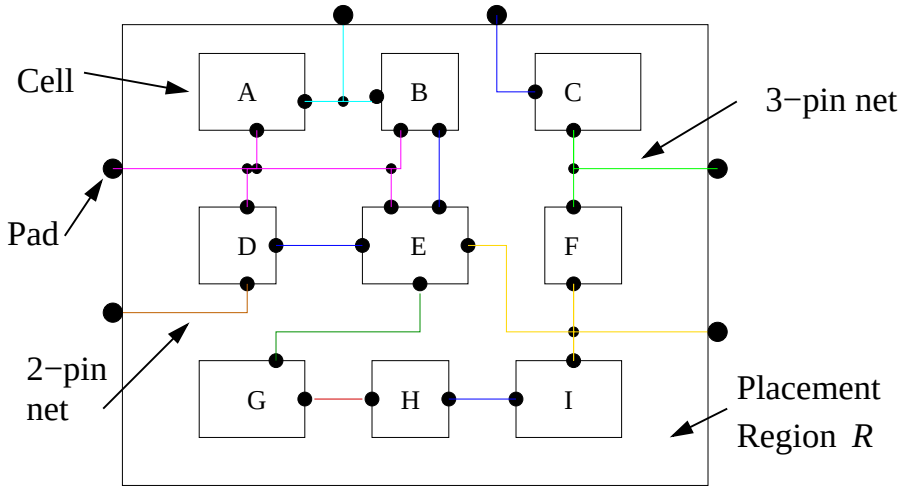
Data Mining

Entertainment

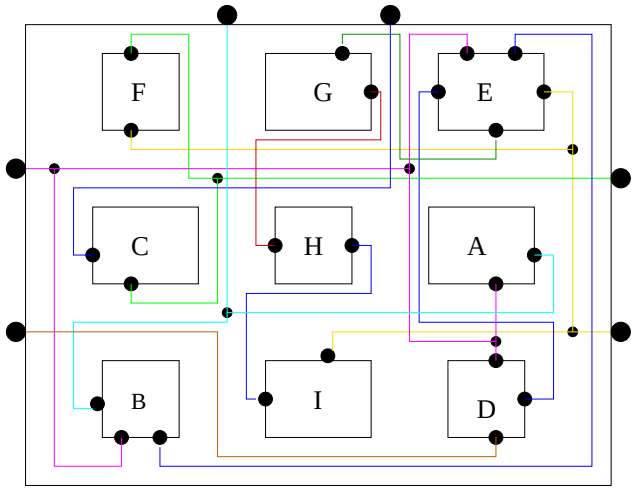
Example: VLSICAD Physical Design

(Computer-aided design of very large-scale integrated circuits)

Layout: arrange the components of an integrated circuit in the most efficient configuration feasible.



A Good Layout



A Poor Layout

Computer

A machine used for computing. A collection of physical components (hardware), each controlled through a software interface, including memory, input & output (I/O), a processing unit (CPU), and an interconnect.

Hierarchy of Abstraction Levels:

User (Systems)

Programming

Functional Organization (Architecture)

Electrical Engineering

Physics, Chemistry

The Von Neumann Model of the Computer

I/O, CPU/ALU, Primary Memory (RAM), Secondary Memory, Interconnect Network

- serial operation
- binary representation
- stored program
- interchangeability of instructions and data

A CPU has

(i) a finite number (100 – 1000) of executable *instructions* implemented as miniature electronic circuits on a chip.

(ii) a small number of memory *registers* for temporarily storing data needed by the current instruction

Binary Representation of Data

By assigning each character symbol a unique integer value and converting it to binary, we can store *all* data in binary.

In the usual *decimal representation (base-10)* 386.7103 means

$$3*10^2+8*10^1+6*10^0+7*10^{-1}+1*10^{-2}+0*10^{-3}+3*10^{-4}.$$

In a *binary representation (base-2)*, 10110.101 means

$$1*2^4+0*2^3+1*2^2+1*2^1+0*2^0+1*2^{-1}+0*2^{-2}+1*2^{-3}.$$

Any number can always be expanded in powers of 2. Different numbers always have different expansions.

Memory, a.k.a. "Storage"

— Can be viewed as one long sequence of cells, each capable of storing a *binary digit* (*bit*): 0 or 1.

— Bits are grouped in consecutive 8-bit chunks called *bytes*.

— Bytes are addressed sequentially but can usually be accessed directly.

The largest number of bits that can be simultaneously manipulated in the hardware circuitry is a fundamental parameter of the machine's architecture known as *word length*. Most machines today (2003) use 32-bit (i.e., 4-byte) words.

File

Any named collection of data or instructions stored in memory. Programs and data are both stored as files. Files are organized hierarchically in *directories* and *subdirectories*.

For *text files*, a.k.a. *formatted files*, each symbol in the user's character set is assigned a unique positive integer value (usually, between 32 and 127) according to a standard code (usually, ASCII). The binary representation of this integer is what is actually stored in memory. Each character code will occupy the same fixed number of bytes, e.g., 1, 2, or 4.

In a *binary* or *unformatted* file, the bits typically do not represent simple alphanumerical characters and instead follow some other pattern understood only by certain programs.

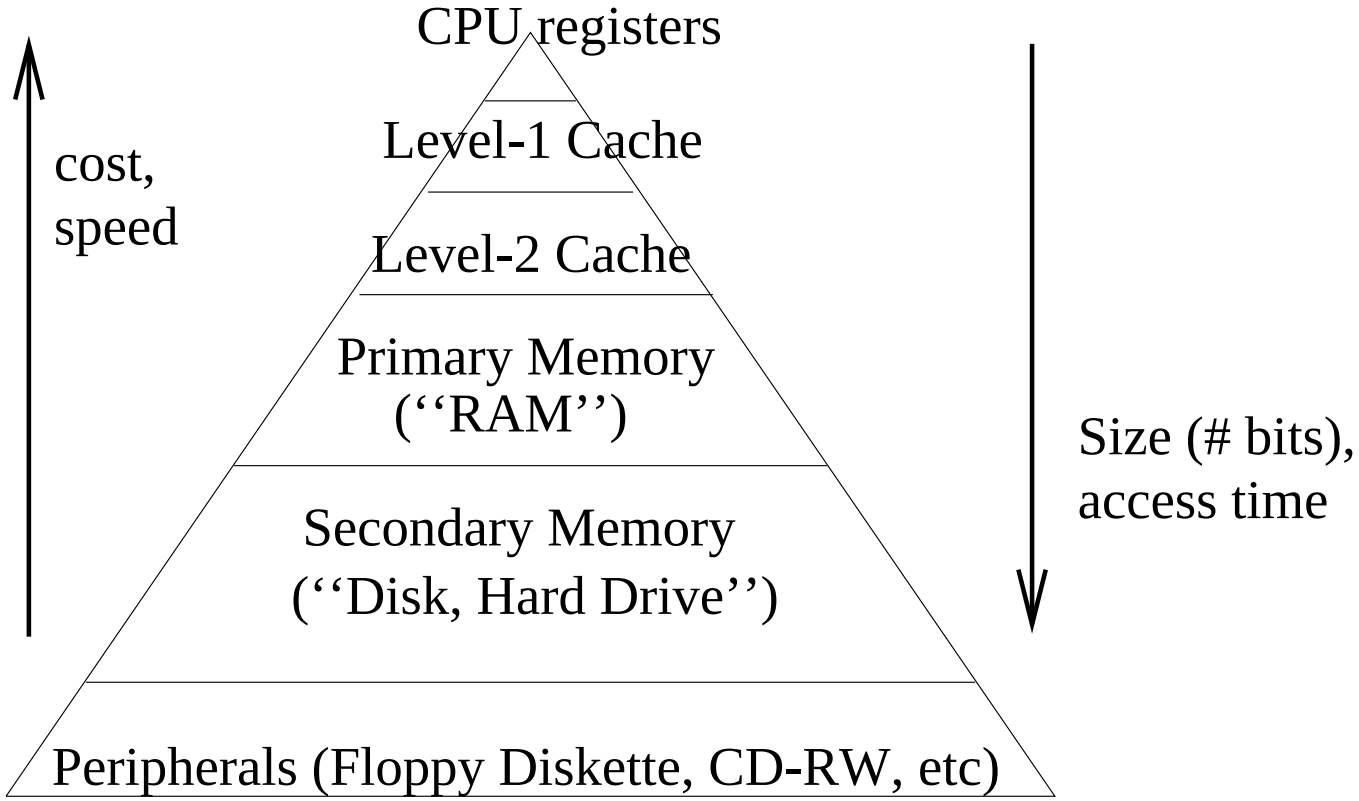
Kinds of Memory

Any location in a *Random Access Memory* (RAM) can be accessed directly within a small fixed amount of time.

In contrast, accessing a particular location in a *Sequential Access Memory* (e.g., a tape memory) requires traversing past all the locations preceding it.

Volatile memory requires a steady source of power to hold its content, e.g., DRAM and SRAM chips. Nonvolatile memory doesn't, e.g., optical and magnetic disks.

The Memory Pyramid— the closer the memory is to the CPU, the faster it can be accessed, the less there is of it, and the more it costs.



The Memory Pyramid

Operating System (OS)

The collection of programs responsible for the smooth coordination of the hardware (physical components) and software (programs) in a computer.

- Input/Output (I/O)
- Command interpreter (shell)
- File management and security
- Program development tools
- Time sharing and accounting
- Communication

Measuring Difficulty

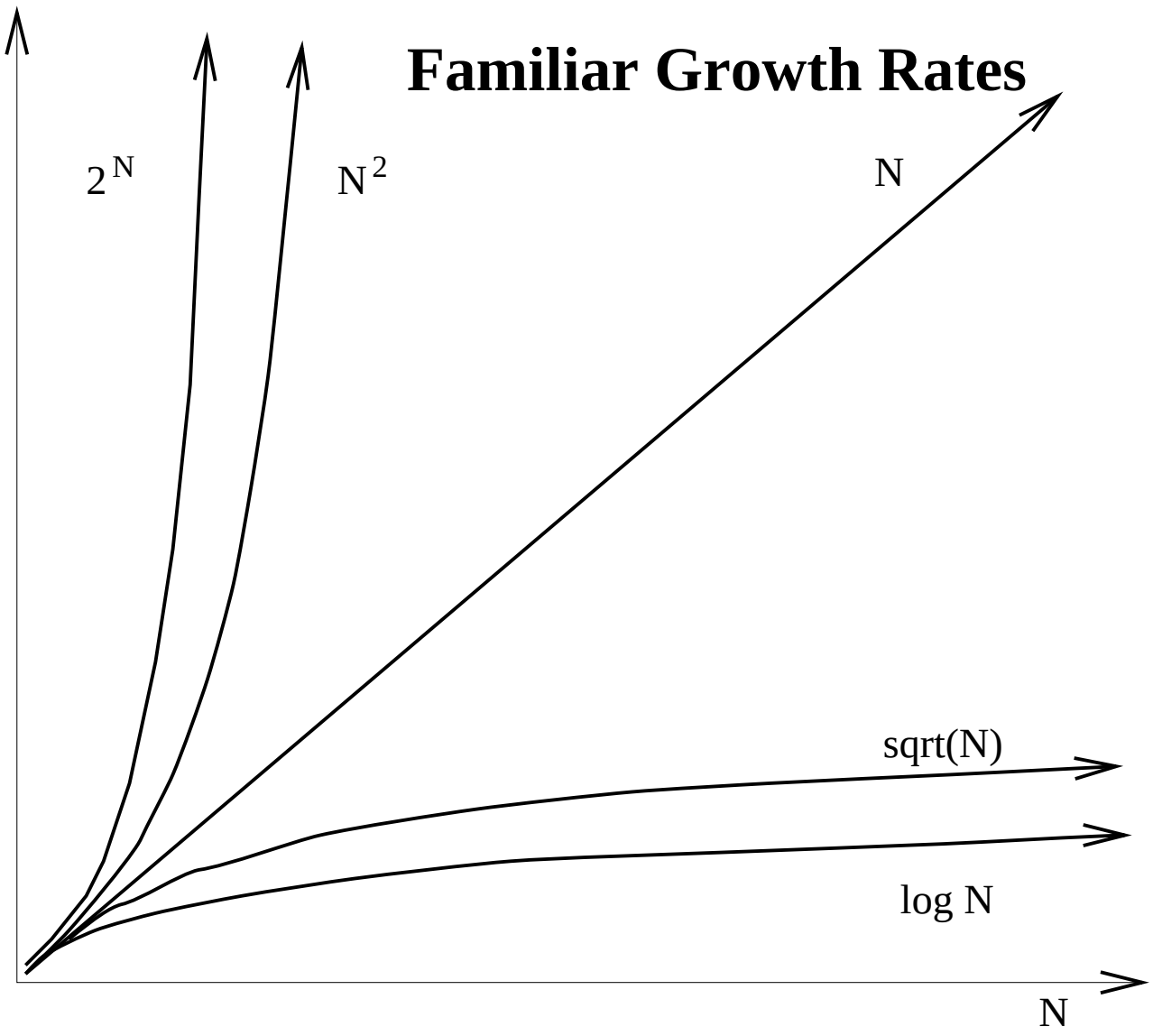
Problems are classified by the amount of computational effort it takes to solve them.

Each problem can be given a size, N = the number of bits needed to encode it. Solving a given problem of size N requires a certain amount of memory and run-time, expressible as an increasing function of N, called a *growth rate.* E.g.,

$\log N$	logarithmic
$\sqrt{N}$, $\sqrt[3]{N}$, ...	nth root
N	linear; $f(kx) = kf(x)$; "scalable"
$N \log N$	
N^2, N^3, ...	polynomial
2^N	exponential

Difficult problems require large amounts of *resources* (memory and time) relative to their sizes; e.g. N^4 or 2^N.

Familiar Growth Rates



Scalability and Intractability

Scalable Problems Exhibit Linear Growth

$f(N) = cN$	e.g., $1, 5, 9, 13, 17, \ldots$
$f(N+1) = f(N) + f(1)$	constant rate of change
$f(kN) = kf(N)$	difficulty proportional to size

Intractable Problems Exhibit Exponential Growth

$f(N) = c2^N$, e.g., $1, 2, 4, 8, 16, \ldots$

$f(N+1) = kf(N) \Rightarrow$ "explosive" growth.

For large values of N (e.g., $N > 1000$), the problem cannot be solved with *any* reasonable amount of resources.

Moore's Law

Since 1950, computing power has increased exponentially.
About every 18 months,

- the minimum feature length on a chip decreases by $\times 0.7$.
- the on-chip density of logic devices (per mm^2) doubles.
- the cost of memory (per bit) decreases by $\times 0.5$.
- the number of operations performed per second doubles.
- the cost of building a chip fabrication plant doubles.

Every 15 years, computers become 1000 times more powerful!
The increase since 1950 is more than one billion!
Although the trend cannot continue forever, it is expected to continue through at least 2008.

Programming Languages

The CPU understands only its own special binary *machine language*, which represents primitive CPU operations called *instructions* at the lowest level of abstraction.

High-level programming languages like C++ enable the construction of *portable source codes*, i.e., programs that run correctly on many different types of machines and operating systems.

A *compiler* is a program that translates high-level source-code in some specific programming language into machine language for some specific machine (e.g., C, Fortran, C++). An *interpreter* simultaneously translates and executes source-code statement by statement (e.g., Java, Lisp).

Source-Code Program

A finite sequence of *statements* in some higher-level programming language; e.g.,

```
read*, a
read*, b
c = a + b
print*, c
```

Executable Program

A finite sequence of binary CPU *instructions*. It is relatively straightforward to translate between these binary instructions and their *assembly* language codes, illustrated below.

```
11010  00100                    read  a
11010  00110                    read  b
01011  00100  00001              fetch a r1
01011  00100  00010              fetch b r2
01101  00001  00010  00011      add   r1 r2 r3
01000  00010  01000              store r3 c
11001  01000                    write c
```

Here `a`, `b` and `c` are programmer-given names of variables and `r1`, `r2`, and `r3` are fixed names of CPU registers.

Essential Elements of Programming Languages

Primitive Expressions can be evaluated directly without simplification.

Means of Combination are used to build compound entities from simpler ones.

Means of Abstraction are used to bind names to new entities and manipulate them like old ones.

Elements of Programming

Programs and subprograms; libraries

Arithmetic and logical expressions

Variables and constants

Assignment of value

Conditional branching (decisions)

Unconditional transfer of control (jumps)

Loops (repetition)

Input and Output (I/O)

Primitive or "built-in" data types: `bool`, `int`, `char`, `double`, etc.

Derived types, compound types, and data structures: pointers, references, lists, arrays, stacks, trees, graphs, etc.

Comments

Programming Paradigms

Procedural: Define your tasks and use the best algorithms you can find.

Object-Oriented: Define appropriate data types endowed with suitable operations.

Procedural programming is algorithm-centered. Object-oriented programming is data-centered.

Some Procedural Languages:

FORTRAN, BASIC, Pascal

Some Object-Oriented Languages:

C++, Java, Ada

Primitive C++ Programs

The empty program:

```
int main () { return 0; }
```

The "Hello, world!" Program:

```
#include <iostream>
int main () {
   std::cout << "Hello, world!\n";
   return 0;
}
```

C++ Basics

A C++ source-code program is a collection of files containing type definitions and function definitions, with *exactly one* of the functions named `int main()`, where execution begins.

A useful program will almost always also contain

- *preprocessor directives*, e.g., `#include <iostream>`
- *using declarations, e.g.,* `using std::cout;`
- *globally defined constants and namespaces*
- *Comments* `/* like this */  // or this`
- *Named local variables* in which data are stored.

Many programs also contain *global* variables. These are accessible to all functions and data types in the program. *Their use should generally be avoided as much as possible.*

In C++, all data are classified into *types*. A *class* or *struct* is a programmer-defined type. E.g.,

```
struct Date { int month, day, year; };
```

defines a `Date` type to hold three integer members: `month`, `day` and `year`.

A *variable* is an *object* with an *identifier*, i.e., a named chunk of memory. E.g., `Date today;` creates a variable of type `Date` named `today`.

A legal C++ identifier is any sequence of letters, digits, and underscores (_) that begins with either a letter or an underscore and is not one of the reserved C++ keywords (e.g., "return").

The programmer must assign types and identifiers to his/her variables, functions, and classes by means of *definitions*.

Every object of the same type requires the same amount of storage, but the exact amount also depends on the operating platform (machine and operating system). The C++ operator `sizeof` can be used to determine the size requirements in bytes for a given type on a given machine. E.g., the C++ statement

```
std::cout << sizeof( int ) ;
```

displays the value 4 on most 32-bit machines.

Data conversion from one type to another is *usually* supported whenever it makes sense. For instance, the integer value of the letter 'A' is obtained by evaluating the expression `int('A')`. On an ASCII machine, this value is 65 (decimal), or 1000001 (binary).

Control Structures

The order in which a program's instructions are executed is known as the *flow of control*. A *control structure* is a block of statements grouped together for a common purpose.

Selection Control structures ("`IF` blocks") are used to decide which instruction to execute next. E.g.,

```
if      ( weight > 300 )          // The "if" condition
   cout << "overweight";          // The "if" body
else if ( weight < 100 )
   cout << "underweight";
else
   cout << "normal";
```

Iteration Control structures ("loops") are used to repeat a block of statements until some condition is violated. E.g.,

```
int main()
{
     int N, i = 1, sum = 0;
     cin >> N;

     while ( i <=  N )       // The loop condition.
     {
       sum = sum + i;        // The loop body.
       i = i+1;
     }
     cout << "1+2+ ... +"<< N << " = " << sum;
     return  0;
}
```

Nesting

A control structure may be placed inside another control structure. *Example:* print out a multiplication table.

```
int main() {
  int N = 5;
  for ( int i = 1; i <= N; i = i+1 )
  {
     for ( int j = 1; j <= N; j = j+1 )
         cout << setw(4) << i*j;
     cout << endl;
  }
  return 0;
}
```

```
Output:
   1    2    3    4    5
   2    4    6    8   10
   3    6    9   12   15
   4    8   12   16   20
   5   10   15   20   25
```

The **principle of structured programming** dictates that a control structure have a *single* point of *entry* and a *single* point of *exit*.

Modularity

The hierarchical organization of a task or system into self-contained subtasks or subsystems, each having a prescribed interface to the others.

Benefits:

- error detection and correction
- modifiability and extensibility
- portability and reuse
- division of labor

Modularity is the essential feature of a well-designed program.

Recursion

To solve a hard or large problem, replace it by a sequence of easier or smaller problems.

An algorithm is *recursive* if it solves a problem by reducing it to a smaller instance of itself.

E.g., let sum(N) denote $1 + 2 + \ldots + N$ for integers $N > 0$. Then

$$\begin{aligned} \text{sum}(N) &= N + \text{sum}(N-1) \\ \text{sum}(1) &= 1 \end{aligned}$$

is a recursive algorithm for calculating sum(N).

Syntactically Recursive Calculation of sum(N)

```
int sum( int N ) {
   if ( N == 0 )
      return 0;
   else
      return N + sum(N-1);   // sum() "calls itself" !
}

int main() {
  int N;
  cin >> N;
  cout << "1+2+ ... +"<< N << " = " << sum(N);
  return 0;
}
```

The Program Development Cycle

Specification

Design

Coding

Compiling

Debugging

Testing and Verification

Risk Analysis

Maintenance

Careful documentation of each stage is crucial!