Computational Meta-genomics Workshop
December 2014

Fundamentals of Computing

Shakuntala Baichoo

Overview

- What is a computer?
- Hardware
- Software
- Databases
- Algorithms

Computers

- Computers are in widespread use today:
  - Business
  - Education
  - Home
- Computers are used for many things:
  - Office work
  - Science and Research
  - Games and Entertainment
  - Multimedia production
  - Data storage
  - …and loads more...

What Is a Computer?

- It is a **programmable**, electronic device that **accepts data**, **performs operations** on that data, and **stores the data** or **outputs** the same as needed
- It follows instructions, called **programs**, which determine the tasks the computer will perform.

Basic operations of a computer

- **Input**: Entering data into the computer
- **Processing**: Performing operations on the data
- **Output**: Presenting the results
- **Storage**: Saving data, programs, or output for future use
- **Communications**: Sending or receiving data

Data vs. Information

- **Data**
  - Raw, unorganized facts
  - Can be in the form of text, graphics, audio, or video
- **Information**
  - Data that has been processed into a meaningful form
- **Information processing**
  - Converting data into information

Computers Then and Now

- The computer as we know is a fairly recent invention
- The history of computers is often referred to in terms of generations
- Each new generation is characterized by a major technological development

- Pre-computers and early computers (before 1945)
  - Abacus, slide rule, mechanical calculator
  - Punch Card Tabulating Machine and Sorter

Computers Then and Now

- First-generation computers (1946-1957)
  - Enormous and powered by vacuum tubes
  - Used a great deal of electricity, and generated a lot of heat
  - ENIAC and UNIVAC
- Second-generation computers (1958-1963)
  - Used transistors
  - Computers were smaller, more powerful, cheaper, more efficient, and more reliable
  - Punch cards and magnetic tape were used to input and store data

Computers Then and Now

- Third-generation computers (1964-1970)
  - Used integrated circuits (ICs)
  - Keyboards and monitors introduced

- Fourth-generation computers (1971-present)
  - Use microprocessors
  - IBM PC, Apple Macintosh
  - Use keyboards, mice, monitors, and printers
  - Use magnetic disks, flash memory, and optical disks for storage
  - Computer networks, wireless technologies, Internet introduced

Computers Then and Now

- Fifth-generation (now and the future)
  - Infancy stage
  - No precise classification
  - May be based on artificial intelligence (AI)
  - Likely use voice input
  - May be based on optical computers and utilise nanotechnology
    - Nanotechnology will allow smaller computers at the atomic scale
    - Optical computer is a computer that uses light instead of electricity (photons rather than electrons) to manipulate, store and transmit data

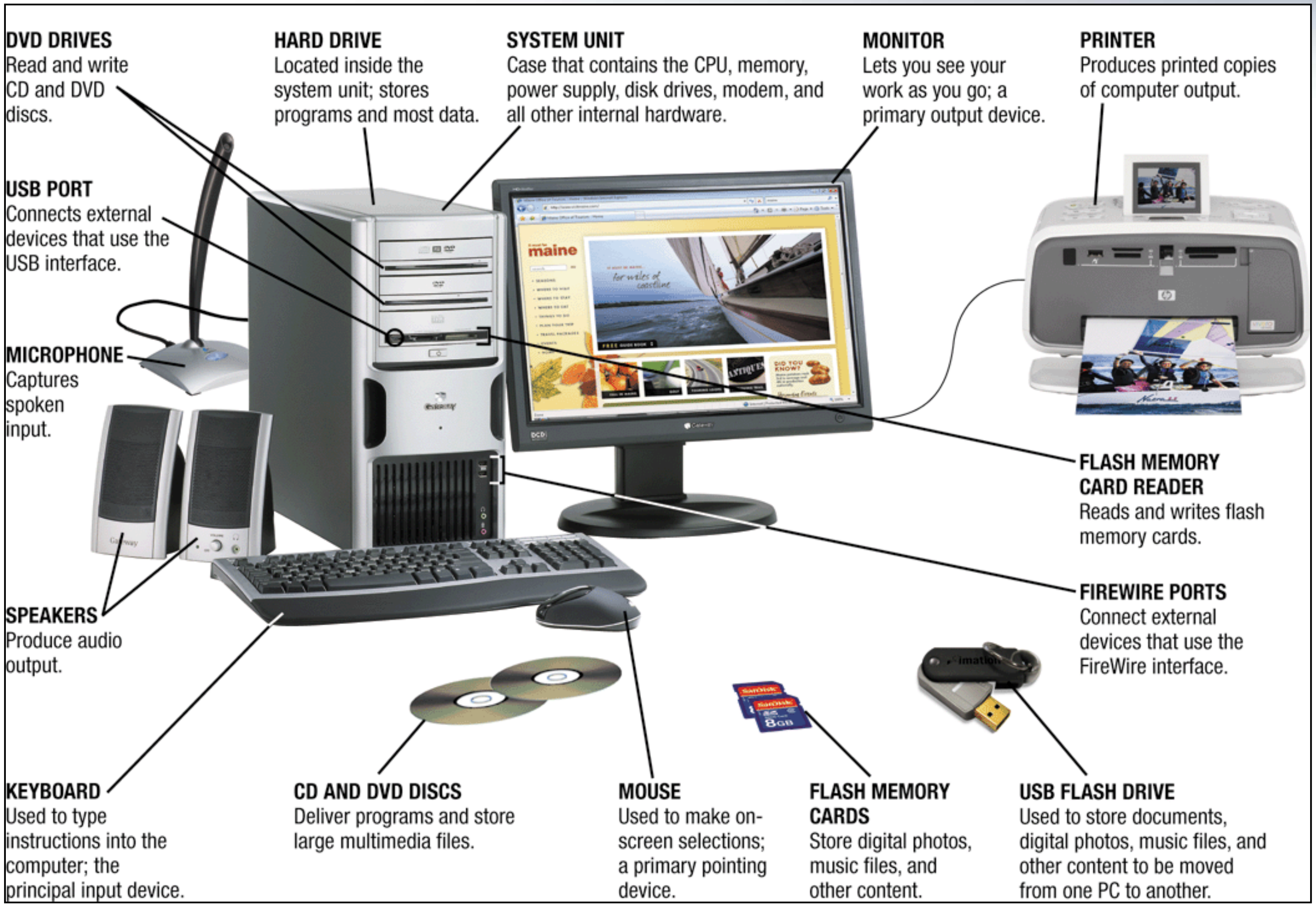
Computer System

- Computer System consists of two main areas of functionality:
  - Hardware
    - Physical components to a computer system
    - The tangible parts
  - Software
    - Instructions given to the hardware
      - Programs or applications to run on the hardware
      - The “intangible” parts

Hardware

- The physical parts of a computer
  - Internal hardware
    - Located inside the main box (system unit) of the computer
  - External hardware
    - Located outside the system unit and plug into ports located on the exterior of the system unit
  - Hardware associated with all five computer operations (i.e. i/p, processing, storage, o/p and communications)

Hardware


DVD DRIVES
Read and write CD and DVD discs.
HARD DRIVE
Located inside the system unit; stores programs and most data.
SYSTEM UNIT
Case that contains the CPU, memory, power supply, disk drives, modem, and all other internal hardware.
MONITOR
Lets you see your work as you go; a primary output device.
PRINTER
Produces printed copies of computer output.
USB PORT
Connects external devices that use the USB interface.
MICROPHONE
Captures spoken input.
FLASH MEMORY CARD READER
Reads and writes flash memory cards.
SPEAKERS
Produce audio output.
FIREWIRE PORTS
Connect external devices that use the FireWire interface.
KEYBOARD
Used to type instructions into the computer; the principal input device.
CD AND DVD DISCS
Deliver programs and store large multimedia files.
MOUSE
Used to make on-screen selections; a primary pointing device.
FLASH MEMORY CARDS
Store digital photos, music files, and other content.
USB FLASH DRIVE
Used to store documents, digital photos, music files, and other content to be moved from one PC to another.

Hardware

- Input devices
  - Used to input data into the computer
  - Keyboards, mice, scanners, cameras, microphones, joysticks, etc.
- Processing devices
  - Perform calculations and control computer's operation
  - Central processing unit (CPU) and memory
- Output devices
  - Present results to the user
  - Monitors, printers, speakers, projectors, etc.

Hardware

- Storage devices
  - Used to store data on or access data from storage media
  - Hard drives, DVD disks and drives, USB flash drives, etc.
- Communications devices
  - Allow users to communicate with others and to electronically access information
  - Modems, network adapters, etc.

Software

- The programs or instructions used to tell the computer hardware what to do
  - **System software**: Operating system allows a computer to operate
    - Boots the computer and launches programs at the user's direction
    - Most use a GUI to interact with the user via windows, icons, menus, buttons, etc.
    - Windows, Mac OS, Linux, etc.

Application Software

- **Application software**: Performs specific tasks or applications
  - Creating letters, budgets, etc.
  - Managing inventory and customer databases
  - Editing photographs
  - Scheduling appointments
  - Viewing Web pages
  - Sending and receiving e-mail
  - Recording / playing CDs
  - Designing homes
  - Playing games

Computer Users and Professionals

- Computer users (*end users*)
  - People who use a computer to obtain information

- Computer professionals include:
  - Programmers
  - Systems analysts
  - Computer operations personnel

DATA STORAGE

What is a database?

- A database is any organized collection of related data stored in a manner so that it can be retrieved when needed.
- Some examples of databases you may encounter in your daily life are:
  - a telephone book
  - airline reservation system
  - Bank account details
  - genome sequences
  - gene sequences along with annotations

What is a database management system (DBMS)?

- a system used to create, maintain, and access computer databases
  - Includes **database engine**: Part of the program that stores and retrieves the data
  - **Various tools**: Used to perform specific tasks
- Examples: MySql, Oracle, Ms Access

What does a database consist of?

- **Tables**: Contain fields and records
  - **Fields** (columns): Single category of data to be stored in a database
  - **Records** (rows): Collection of related fields in a database
    - A record associated with a *Nucleotide Sequence Database* may contain the following information:
      - A **contact name**;
      - The **input sequence** with a description of the type of molecule;
      - The **scientific name** of the source organism from which it was isolated; and, often,
      - **Literature citations** associated with the sequence.

Importance of Databases

- Databases are developed to **collect**, **archive**, **visualize** and **organize data** to enable intelligent data description/interpretation, discovery, retrieval & invocation.
- □ Databases exist as a way of ensuring **interoperability** and **integration** between different research institutes, **research databases**, **data mining tools**, **software** and ordinary **end-user** with minimal restriction.

How do we access databases?

- Basically there are several approaches to the use of databases:
  - Direct access to database tables
  - Using predefined operations which can be packaged in programs
- Either way, we need **queries** to access databases

Querying Bioinformatics databases

- A **database query** is “a method to retrieve information from the database”.
- Organization of database records into predetermined fields, enables end-users to query on fields.
- Database querying is made easier using a special programming language known as SQL

Database use 1: direct access to database tables

- Run your own database on your own computer (e.g. MySql Server)
- Use a program on your PC which gives you direct access to the tables in the remote database (a client for the database)

In both cases, you need instructions as to what the tables are and what they contain, such as SQL.

SQL statements

- SQL ("Structured Query Language") is a language for specifying the creation of databases and the updating and retrieval of information in them.
- It is general and "portable" – so that it can be used with a variety of different database systems without having to learn a new language for each one.
- The language goes far beyond this scope of this workshop. Briefly, it can be used to:
  - Specify the tables in the database and the fields (columns) they contain
  - Make additions and updates to the data in those tables
  - Retrieve information from one or more of the tables

SQL for data retrieval

- A typical SQL statement for data retrieval would look something like this:

```
SELECT <some fields> FROM <table> WHERE
  <condition>;
```

- The condition effectively selects certain rows from the table.
- Thus the result is often a smaller table than the one being queried.
- Tables can be “joined” together to combine information from more than one table, for example when extracting a molecular sequence from one table and the bibliographic details of the reference to where it was published from another table.

Database use 2: predefined operations

- Alternatively, you might have forms and queries already set up for you, which you can just run in order to perform predefined kinds of searches. These predefined operations can be made directly available to you by:
  - Browsing a web page, typically containing a form, which gives you access to a database somewhere else.
  - Using or even writing a small program (sometimes called a script to make it seem less scary) to fetch the data for you. This allows you to process the data in useful ways:
    - to search for features you're interested in,
    - to summarise the data in the way you want, or
    - to extract data for statistical analysis to test hypotheses.

Database use 2: using predefined operations

- The predefined operations may be packaged as CGI programs or Web Services or in a variety of other ways, but basically you just send a request to the service, optionally with some ‘parameters’ to specify what you want, and wait for the reply.
- The reply may come back, usually,
  - in HTML (as a web page containing the data requested) or
  - as some other sort of file to be downloaded (i.e. stored on your PC), either
    - in one of a number of formats invented by the data providers,
    - in XML, a standard but flexible (and verbose) way to structure a data file, so that other programs (rather than humans) can process it easily.

Database use 3: using predefined operations

- Example
  - NCBI web site
  - Entrez
    - a biological data retrieval system that acts as the search engine for NCBI databases.
    - Searching can be made more precisely by using Boolean operators like AND, OR or NOT with the search statement.

Analyzing the data fetched from databases

- In order to make any sense of the data, we need to carry out some processing using **algorithms**
  - That have already been developed and packaged in software/programs
  - That you need to write on your own using a programming language
    - This could be more efficient

What is an algorithm?

- An ***algorithm*** is a sequence of instructions that one must perform in order to solve a well-formulated ***problem***
- *First you must identify exactly what the problem is!*
- A ***problem*** describes a class of computational tasks. A problem ***instance*** is one particular input from that task
- In general, you should design your algorithms to work for ***any*** instance of a problem (although there are cases in which this is not possible)

Properties of algorithms

- ✓ Correctness
- ✓ Composed of a series of concrete steps
- ✓ No ambiguity as to next step
- ✓ Composed of a finite number of steps
- ✓ Will terminate

C. A. Shaffer, *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall.

Examples of algorithms

- shampoo instructions
  - “Lather. Rinse. Repeat.”
  - Does not terminate

    No:
- most recipes
  - “Stir-fry until crisp-tender.”
  - Next step ambiguous

    No:

If not an algorithm, then what?

- Procedure
- Heuristic

Examples of algorithms(Continued)

long division

Yes:

```
  14.5
 ____
4)58
  4
  --
  18
  16
  --
   20
   20
   --
    0
```

Another Algorithm Example

- For example, we might need **to sort a sequence of numbers into non-decreasing order**.
- This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools.
- Here is how we formally define the sorting problem:

Input: A sequence of n numbers $\langle a_1; a_2; \ldots; a_n \rangle$.

Output: A permutation (reordering) $\langle a'_1; a'_2; \ldots; a'_n \rangle$ of the input sequence such that $a'_1 \quad a'_2 \qquad a'_n$.

- For example, given the input sequence <31; 41; 59; 26; 41; 58>, a sorting algorithm returns as output the sequence <26; 31; 41; 41; 58; 59>.
- An algorithm is said to be ***correct*** if, for every input instance, it halts with the correct output.

WHAT KINDS OF PROBLEMS DO ALGORITHMS SOLVE?

The Human Genome Project

- The Human Genome Project has made great progress toward the goals of identifying all the 100,000 genes in human DNA, determining the sequences of the 3 billion chemical base pairs that make up human DNA, storing this information in databases, and developing tools for data analysis.
- Each of these steps requires sophisticated algorithms.
- The savings are in time, both human and machine, and in money, as more information can be extracted from laboratory techniques.

The Internet

- The Internet enables people all around the world to quickly access and retrieve large amounts of information.
- With the aid of clever algorithms, sites on the Internet are able to manage and manipulate this large volume of data.
- Examples of problems that make essential use of algorithms include finding good routes on which the data will travel (techniques for solving such problems appear in and using a search engine to quickly find pages on which particular information resides)

E-commerce

- Electronic commerce enables goods and services to be negotiated and exchanged electronically, and it depends on the privacy of personal information such as credit card numbers, passwords, and bank statements.
- The core technologies used in electronic commerce include public-key cryptography and digital signatures, which are based on numerical algorithms and number theory.

Manufacturing and other Commercial Applications

- Manufacturing and other commercial enterprises often need to allocate scarce resources in the most beneficial way.
  - An oil company may wish to know where to place its wells in order to maximize its expected profit.
  - A political candidate may want to determine where to spend money buying campaign advertising in order to maximize the chances of winning an election.
  - An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that government regulations regarding crew scheduling are met.
  - An Internet service provider may wish to determine where to place additional resources in order to serve its customers more effectively.
- All of these are examples of problems that can be solved using linear programming.

Other examples of Algorithms

- a function for computing n!, given n
- an instruction for assembling a piece of furniture
- Sequence Alignment algorithms
- Genome assembly algorithms
- various sorting and searching algorithms

Expressing Algorithms

- Reasoning about an algorithm is impossible without careful description of the sequence of steps to be performed. The three most common forms of algorithmic notations are:
  - English
    - Most natural but least precise
  - Pseudocode
    - Best defined as a programming language that never complains about syntax errors!
  - A real programming language
    - Precise but difficult to write and understand

Programs and algorithms

- A **program** is an implementation of an algorithm in a programming language

- Many programs exist for the same algorithm
  - Different programming languages
  - Different programming standards
  - Different details, e.g., error checking

- **Algorithm**: abstract mathematical object,
- **Program**: practical realization of it

Problem, Algorithm & Program [Example]

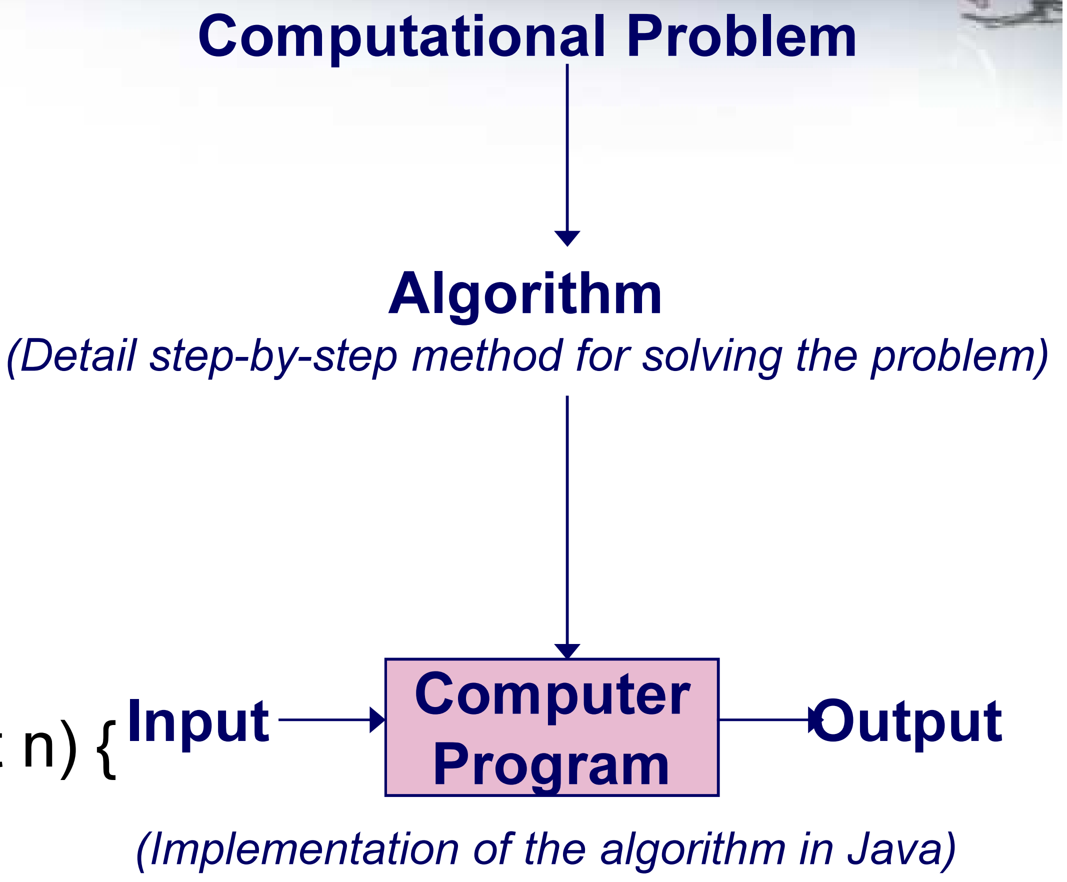
- **Problem**:
  - Given n, find n!

- **Algorithm**:
  - If n = 0, n! = 1

    Else n! = n * (n – 1)!

- **Program in Java**:
  - public long factorial(int n) {

```
  if(n == 0) return 1;
  else return n * factorial(n – 1);
}
```


Computational Problem
Algorithm
(Detail step-by-step method for solving the problem)
Input
Computer Program
Output
(Implementation of the algorithm in Java)

Algorithm classification

- Algorithms that use a similar problem-solving approach can be grouped together
- We’ll talk about a classification scheme for algorithms
- The purpose is not to be able to classify an algorithm as one type or another, but to highlight the various ways in which a problem can be attacked
- *Note: Classification scheme is neither exhaustive nor disjoint*

A short list of categories

- Some Common Algorithm types :
  - Simple recursive algorithms
  - Backtracking algorithms
  - Divide and conquer algorithms
  - Dynamic programming algorithms
  - Brute force algorithms
  - Heuristic Algorithms
  - Randomized algorithms

Simple recursive algorithms

- Recursion is one of the most ubiquitous algorithmic concepts.
- An algorithm is said to be recursive if it calls itself.
- A simple recursive algorithm:
  - Solves the base cases directly
  - Recurs with a simpler sub-problem
  - Does some extra work to convert the solution to the simpler sub-problem into a solution to the given problem
- We call these “simple” because several of the other algorithm types are inherently recursive

Iteration: an alternative to recursion

Recursion vs **Iteration**

- In general, an iterative version of a method will execute more efficiently in terms of time and space than a recursive version.
- This is because the overhead involved in entering and exiting a function in terms of stack I/O is avoided in iterative version.
- Sometimes we are forced to use iteration because stack cannot handle enough activation records - Example: power(2, 5000)

Why Recursion?

- Usually recursive algorithms have less code, therefore algorithms can be easier to write and understand - e.g. Towers of Hanoi.
  - However, avoid using excessively recursive algorithms even if the code is simple.
- Sometimes recursion provides a much simpler solution. Obtaining the same result using iteration requires complicated coding - e.g. Quicksort, ***Towers of Hanoi***, etc.
- Recursive methods provide a very natural mechanism for processing recursive data structures.
  - A recursive data structure is a data structure that is defined recursively – e.g. Tree.
- Some recursive algorithms are more efficient than equivalent iterative algorithms. Example, each of the following recursive power methods evaluate x^n more efficiently than the iterative one.

Example recursive algorithms

- To calculate N!
  - N!=N x (N-1)!
    - Given 1!=1
    - We can keep on decreasing the value of N until it reaches 1 and then we stop the recursion

What is a Backtracking Algorithm?

- For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path.
  - If you make the correct set of choices, you end up at the solution.
  - On the other hand, if you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different path.
- Algorithms that use this approach are called **backtracking algorithms.**
- **Backtracking** is a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem.

Backtracking algorithms

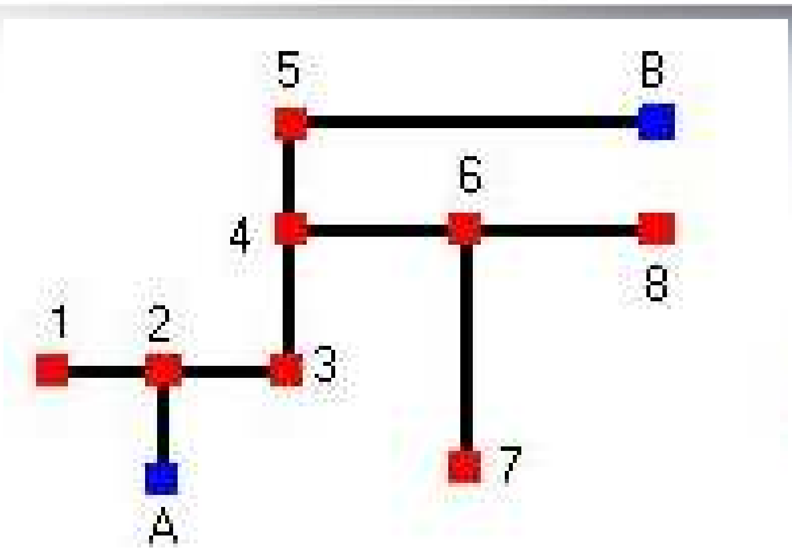
- Backtracking algorithms are based on a depth-first recursive search
- A backtracking algorithm:
  - Tests to see if a solution has been found, and if so, return it; otherwise
  - For each choice that can be made at this point,
    - Make that choice
    - Recur
    - If the recursion returns a solution, return it
  - If no choices remain, return failure

How do we solve backtracking algorithms?

- If we think about a backtracking algorithm as the process of repeatedly exploring paths until we encounter the correct solution, the process appears to be of an iterative nature.
- As it happens, however, most problems of this form are easier to solve recursively.
- The fundamental recursive insight is simply this:
  - a backtracking problem has a solution if and only if at least one of the smaller backtracking problems that results from making each possible initial choice has a solution.

Example backtracking algorithm I

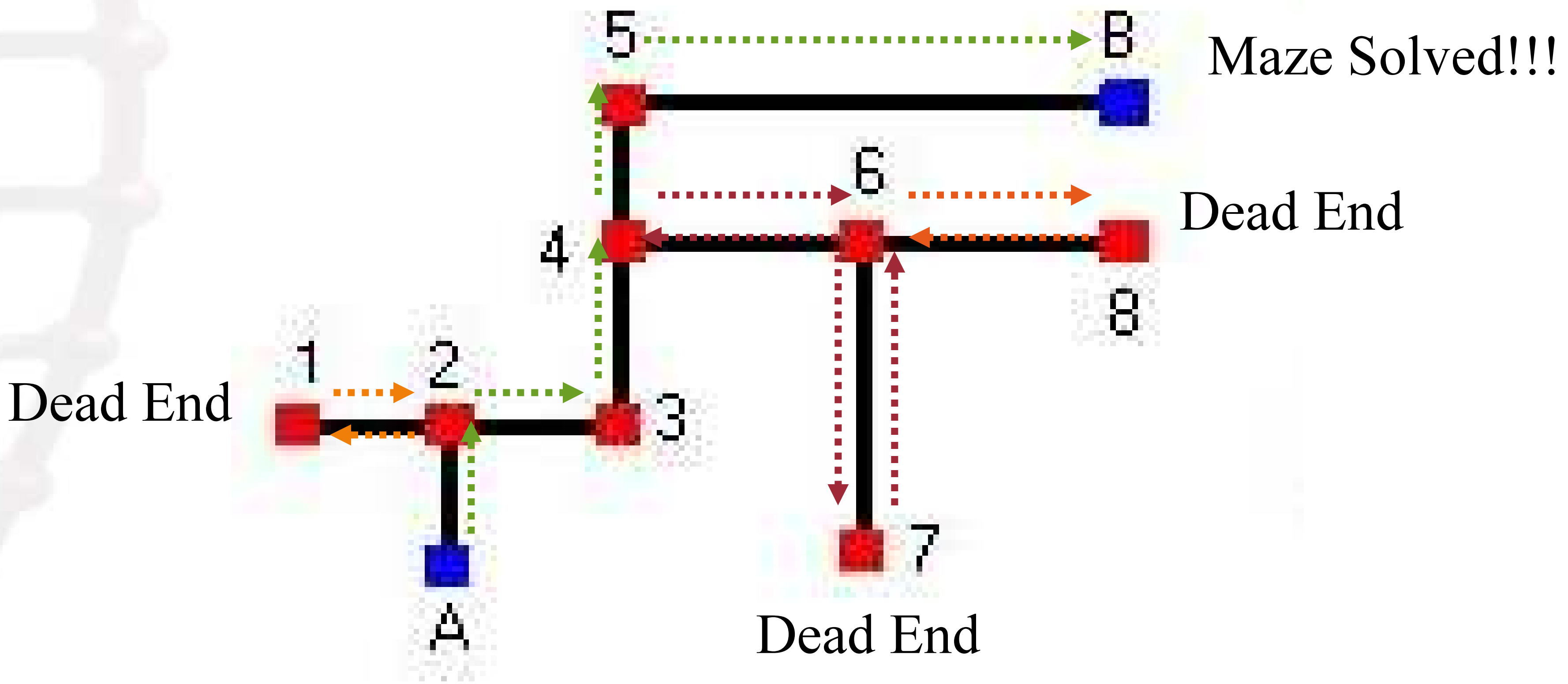
Solving a maze



- Consider a maze, shown in the diagram above, where a player has to decide how to get from room A to room B.
- The player can move from room to room through the corridors provided, but has no way of telling how many corridors a room is connected to until he reaches that room.

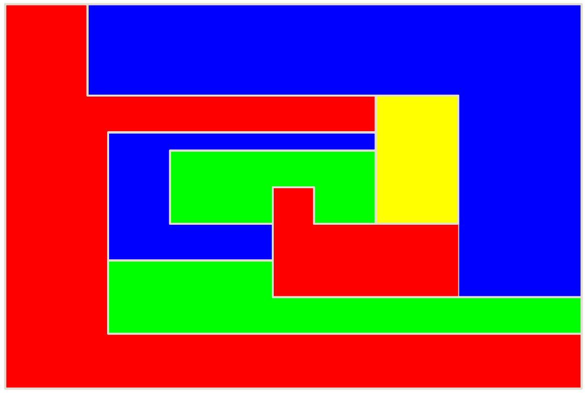
Example backtracking algorithm I (cont.)

- We will now see a sample session to understand the approach the player uses to solve the maze.



Example backtracking algorithm II

- To color a map with no more than four colors:
  - color(Country n):
    - If all countries have been colored (n > number of countries) return success; otherwise,
    - For each color c of four colors,
      - If country n is not adjacent to a country that has been colored c
        - » Color country n with color c
        - » recursively color country n+1
        - » If successful, return success
    - If loop exits, return failure



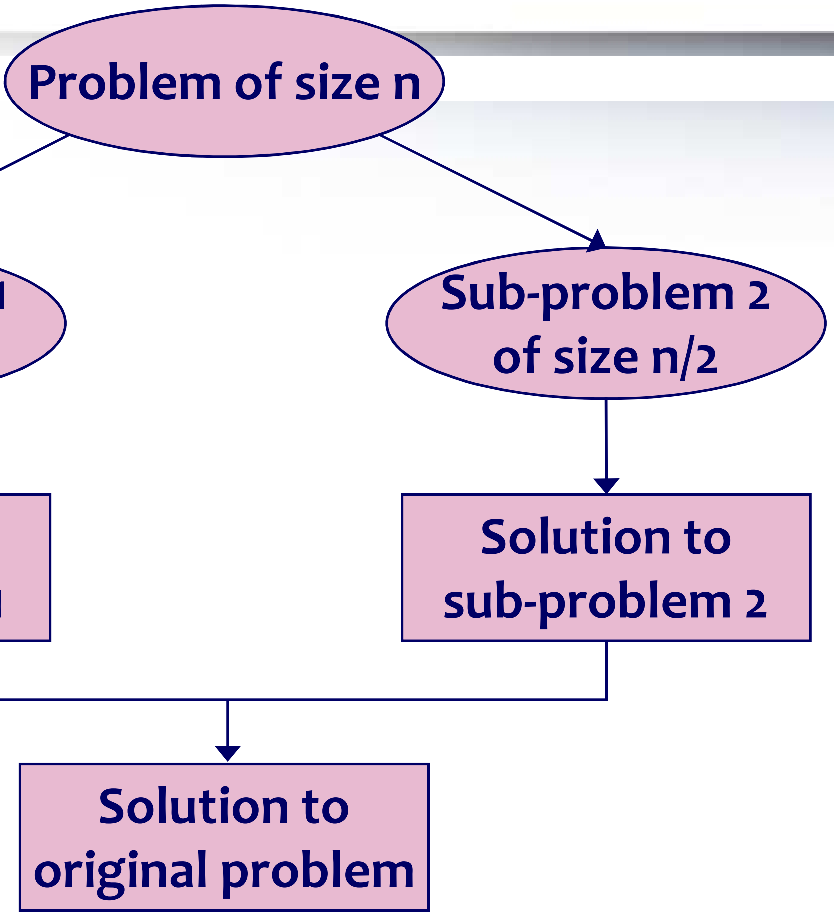
Divide and Conquer I

- A successful military strategy
  - Easier to defeat one army of 50,000 men, followed by another army of 50,000 men than to beat a single 100,000 man army
  - Wise general would attack so as to divide the enemy army into two forces and then mop up one after the other

Divide and Conquer II

- A divide and conquer algorithm is based on dividing problem into sub-problems
- Approach
  1. Divide problem into smaller subproblems
     - Sub-problems must be of same type
     - Sub-problems do not need to overlap
  2. Solve each sub-problem recursively
  3. Combine solutions to solve original problem
- Traditionally, an algorithm is only called “divide and conquer” if it contains at least two recursive calls

Divide and Conquer III


Problem of size n
Sub-problem 1 of size n/2
Sub-problem 2 of size n/2
Solution to sub-problem 1
Solution to sub-problem 2
Solution to original problem

- ***Note:***
  - *Divide and conquer algorithms often implemented using recursion*
  - *But, not all recursive functions are divide-and-conquer algorithms*

Divide and Conquer – Examples

- Quicksort:
  - Partition the array into two parts (smaller numbers in one part, larger numbers in the other part)
  - Quicksort each of the parts
  - No additional work is required to combine the two sorted parts
- Mergesort:
  - Cut the array in half, and mergesort each half
  - Combine the two sorted arrays into a single sorted array by merging them

What is Dynamic programming?

- Dynamic Programming is an algorithm design technique for *optimization problems:* often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to sub-problems.
- Unlike divide and conquer, sub-problems are not independent.
  - Sub-problems may share sub-sub-problems, …

Dynamic programming algorithms I

- Based on remembering past results and uses them to find new results
- **Approach**
  1. Divide problem into smaller subproblems
     - Sub-problems must be of same type
     - Sub-problems must overlap
  2. Solve each sub-problem recursively
     - May simply look up solution
  3. Combine solutions to solve original problem
  4. Store solution to problem

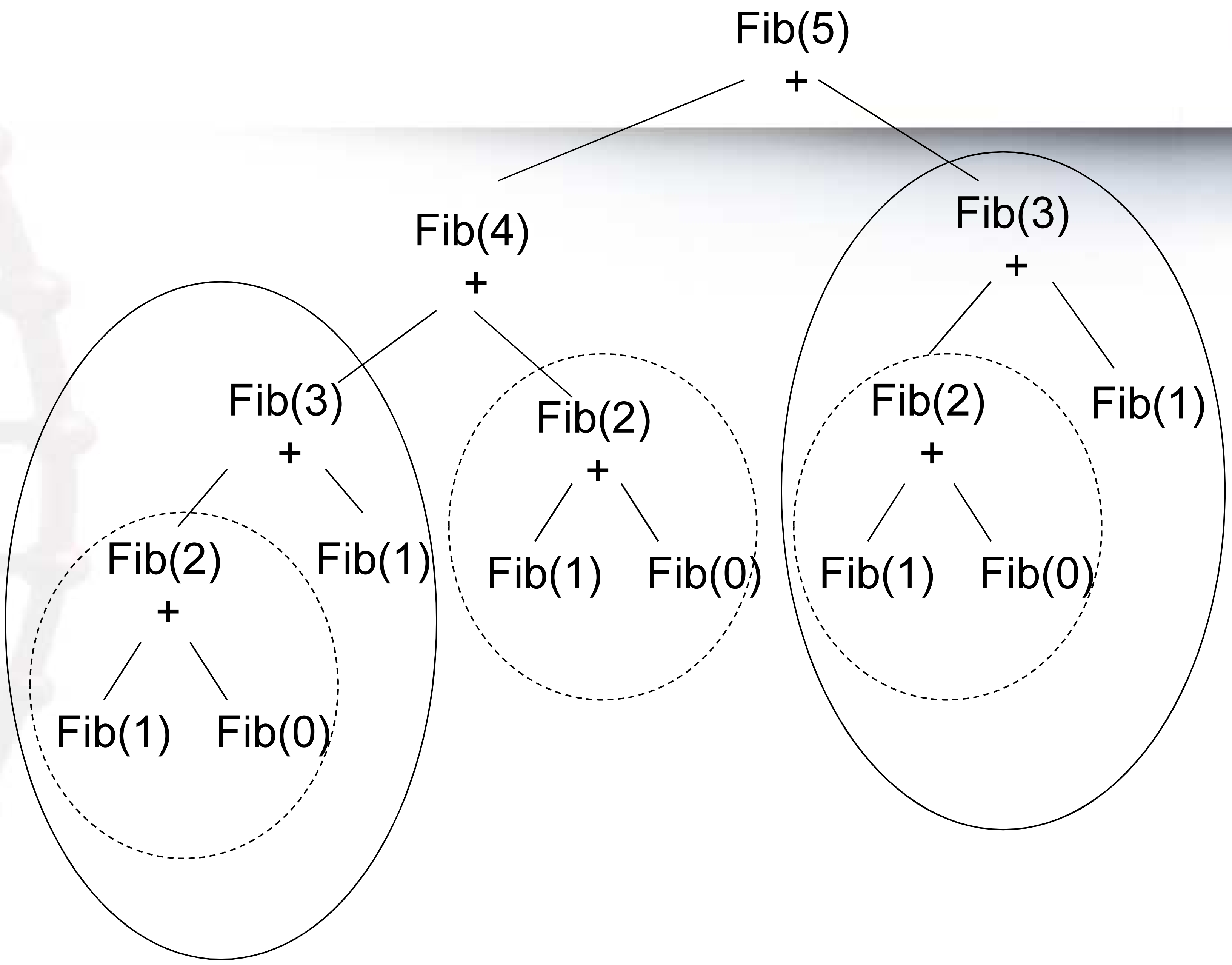
Dynamic programming algorithms II

- Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the “best” one
  - Requires “optimal substructure” and “overlapping subproblems”
    - Optimal substructure: Optimal solution contains optimal solutions to sub-problems
    - Overlapping sub-problems: Solutions to sub-problems can be stored and reused in a bottom-up fashion

Fibonacci numbers (Remember)

- Fibonacci numbers
  - fibonacci(0) = 0
  - fibonacci(1) = 1
  - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
- Recursive algorithm to calculate fibonacci(n)
  - If n is 0, Fibonacci number is 0
  - If n is 1, Fibonacci number is 1
  - Else compute fibonacci(n) as the sum fibonacci(n-1) and fibonacci(n-2)

Simple algorithm $\Rightarrow$ exponential time $O(2^n)$ =>Very time-consuming


Fib(5)
+
Fib(4)
+
Fib(3)
+
Fib(3)
+
Fib(2)
+
Fib(1)
Fib(2)
+
Fib(1)
Fib(0)
Fib(1)
Fib(0)
Fib(2)
+
Fib(1)
Fib(1)
Fib(0)

Dynamic Programming – Example

- Dynamic programming version of fibonacci(n)
- To find the nth Fibonacci number:
    - If n is zero or one, return n;
  - otherwise,
    - Compute, *or look up in a table,* fibonacci(n-1) and fibonacci(n-2)
    - Find the sum of these two numbers
    - Store the result in a table and return it
- Since finding the nth Fibonacci number involves finding all smaller Fibonacci numbers, the second recursive call has little work to do
- The table may be preserved and used again later

Another Example: Pairwise sequence alignment

What is a Greedy algorithm?

- A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.
- In many problems, a greedy strategy **does not in general produce an optimal solution**, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a global optimal solution in a reasonable time.

Greedy algorithms

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A “greedy algorithm” sometimes works well for optimization problems
- A greedy algorithm works in phases: At each phase:
  - You take the best you can get right now, without regard for future consequences
  - Avoid backtracking, exponential time $O(2^n)$
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum
- Based on trying best current choice

The main components of a Greedy algorithm?

- In general, greedy algorithms have five components:

1. A **candidate set**, from which a solution is created
2. **A selection function**, which chooses the best candidate to be added to the solution
3. A **feasibility function**, that is used to determine if a candidate can be used to contribute to a solution
4. **An objective function**, which assigns a value to a solution, or a partial solution, and
5. **A solution function**, which will indicate when we have discovered a complete solution

Greedy algorithm Example: Counting money

- Suppose you want to count out a certain amount of money, using the fewest possible bills and coins
- A greedy algorithm that would do this would be: At each step, take the largest possible bill or coin that does not overshoot
  - Assuming we have 1¢, 5¢, 10¢, 25¢, 50¢, \$1, \$2, \$5, \$10, \$20, \$50, \$100 …
  - Example: To make \$6.39, you can choose:
    - a \$5 bill
    - a \$1 bill, to make \$6
    - a 25¢ coin, to make \$6.25
    - A 10¢ coin, to make \$6.35
    - four 1¢ coins, to make \$6.39
- For US money, the greedy algorithm always gives the optimum solution
- **Does the greedy algorithm always give the optimum solution?**

A failure of the greedy algorithm

- In some (fictional) monetary system, “krons” come in 1 kron, 7 kron, and 10 kron coins
- Using a greedy algorithm to count out 15 krons, you would get
  - A 10 kron piece
  - Five 1 kron pieces, for a total of 15 krons
  - This requires six coins
- A better solution would be to use two 7 kron pieces and one 1 kron piece
  - This only requires three coins
- The greedy algorithm results in a solution, but not in an optimal solution

What is a Brute force algorithm?

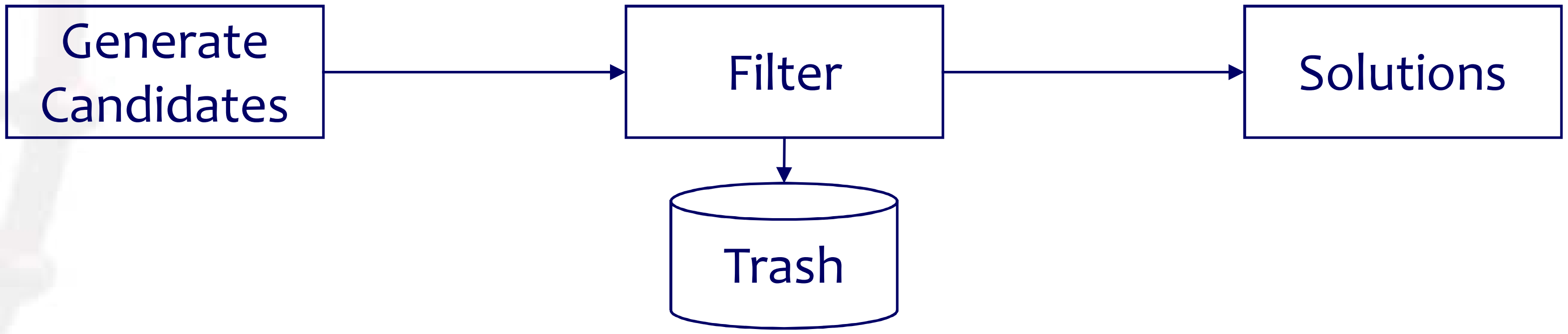
- It is a straightforward approach, usually based directly on the problem's statement and definitions of the concepts involved.

Examples:

1. Computing a^n ($a > 0$, n a nonnegative integer)
2. Computing $n!$
3. Multiplying two matrices
4. Searching for a key of a given value in a list

Brute force algorithm I

- A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found

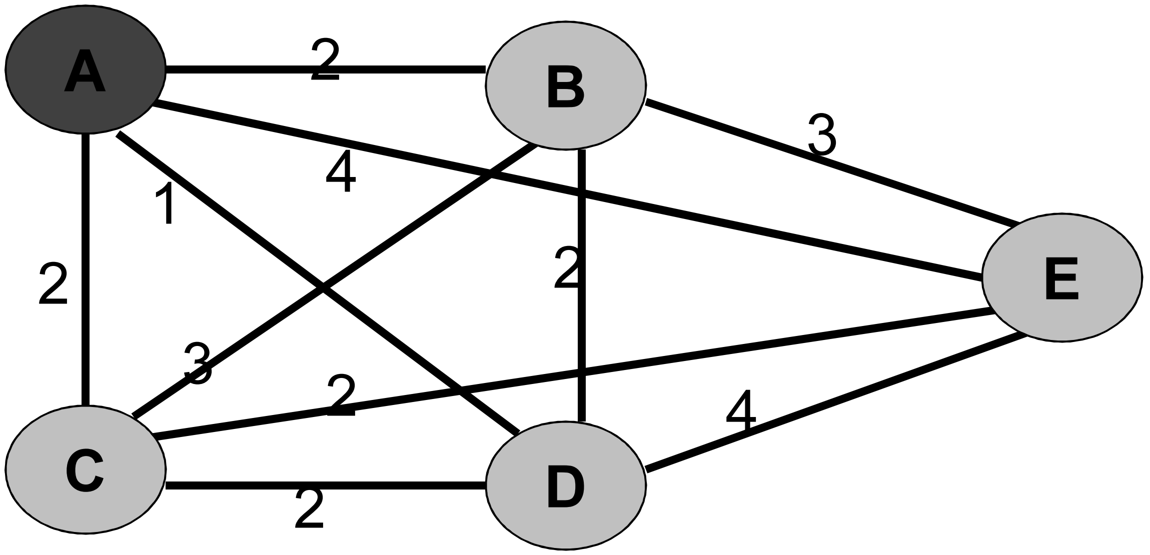


Brute force algorithm II

- A brute force algorithm can be:
  - Optimizing: Find the *best* solution. This may require finding all solutions, or if a value for the best solution is known, it may stop when any best solution is found
    - Example: Finding the best path for a traveling salesman
  - Satisfying: Stop as soon as a solution is found that is *good enough*
    - Example: Finding a traveling salesman path that is within 10% of optimal
  - Generally most expensive approach

Traveling Salesman Problem (TSP)

- A traveling salesman is getting ready for a big sales tour. Starting at his hometown, he will conduct a journey in which each of his target cities is visited exactly once before he runs home. Given the pairwise distances between cities, what is exactly the best order in which to visit them, so as to minimize the overall distance traveled?



Brute Force Algorithm – Example

- Traveling Salesman Problem (TSP)
  - Given weighted undirected graph (map of cities)
  - All cities are denoted by 1,…,n, the salesman's hometown being 1, and let D=(d_{ij}) be the matrix of intercity distances.
  - Find lowest cost path visiting all nodes (cities) once
  - No known polynomial-time general solution
  - The goal is to design a tour that starts and ends at 1, includes all the cities once and has minimum total length.
- Brute force approach
  - Find all possible paths using recursive backtracking
  - Calculate cost of each path
  - Return lowest cost path
- Requires exponential time $O(2^n)$

Improving brute force algorithms

- Often, brute force algorithms require exponential time
- Various *heuristics* and *optimizations* can be used
  - Heuristic: A “rule of thumb” that helps you decide which possibilities to look at first
  - Optimization: In this case, a way to eliminate certain possibilities without fully exploring them

Heuristic Algorithm

- Heuristic $\Rightarrow$ “rule of thumb”
- The term **heuristic** is used for algorithms which find solutions among all possible ones ,but they do not guarantee that the best will be found, therefore they may be considered as approximate and not accurate algorithms.
- Heuristic algorithms
  - Based on trying to guide search for solution
  - Approach
    - Generate and evaluate possible solutions
      - Using “rule of thumb”
      - Stop if satisfactory solution is found
  - Can reduce complexity
  - Not guaranteed to yield best solution
  - Heuristics used frequently in real applications

Heuristic Algorithms

- These algorithms, usually find a solution close to the best one and they find it fast and easily.
- Sometimes these algorithms can be accurate, that is they actually find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best.
- The method used from a heuristic algorithm is one of the known methods, such as greediness, but in order to be easy and fast the algorithm ignores or even suppresses some of the problem's demands.

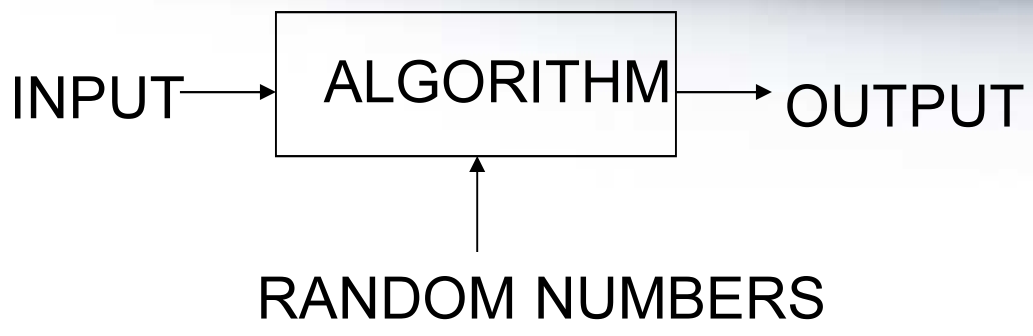
Heuristic Algorithm – Example

- Heuristic algorithm for TSP
  - Find possible paths using recursive backtracking
    - Search 2 lowest cost edges at each node first
  - Calculate cost of each path
  - Return lowest cost path from first 100 solutions
- Not guaranteed to find best solution
- Heuristics used frequently in real applications

Randomized algorithms

- A randomized algorithm is just one that depends on random numbers for its operation
- These are randomized algorithms:
  - Using random numbers to help find a solution to a problem
  - Using random numbers to improve a solution to a problem
- These are related topics:
  - Getting or generating “random” numbers
  - Generating random data for testing (or other) purposes

Randomized algorithms



- Makes some random (or pseudorandom) choices
- Uses a random number at least once during the computation to make a decision
  - Example: In Quicksort, using a random number to choose a pivot
  - Example: Trying to factor a large number by choosing random numbers as possible divisors
- Methods of last resort
  - Often used when no other feasible solution technique is known
- Used to solve problems for which the solution space is so large that an exhaustive search is infeasible
- **Example:** Gibbs Sampler for find motifs

Big-O Notation for representing the complexity of an algorithm

- We measure the complexity or efficiency of an algorithm in terms of its resource consumption.
  - Time
  - Space
- We use the Big-Oh (O) notation to represent the complexity of an algorithm
- We say that an algorithm has complexity:
  - T(n)=O(n)
  - It means that when the no. of input doubles then time taken also doubles
  - Same applies for space

Example of complexities

- **Linear**: it is in O(n)
- **Logarithmic**: it is in O(lgn)
- **Linearithmic**: it is in O(n lgn)
- **Quadratic**: it is in O(n^2)
- **Cubic**: it is in O(n^3)
- **Exponential**: It is in O(x^n)