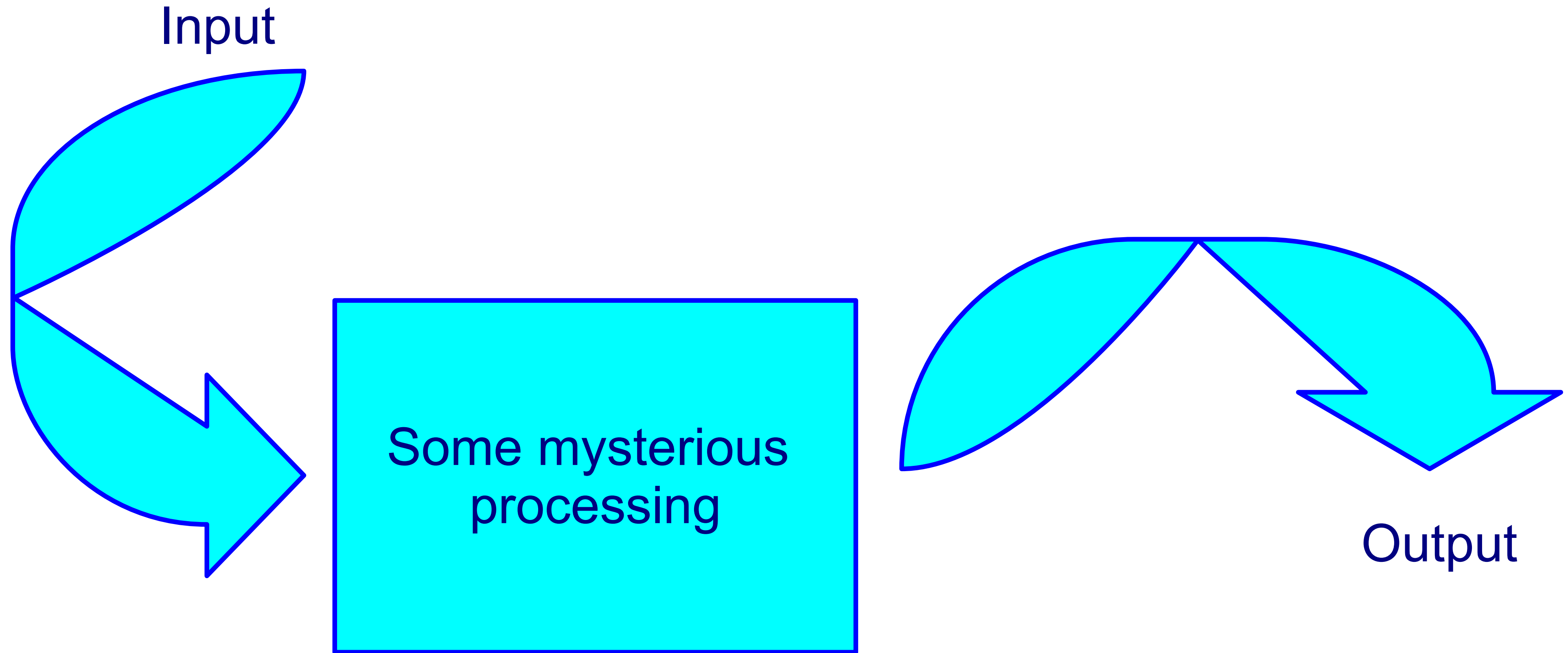
Data Structure

- Before we go into details, what is a data structure exactly?

- To answer that, we must first understand:
- What is a computer program?

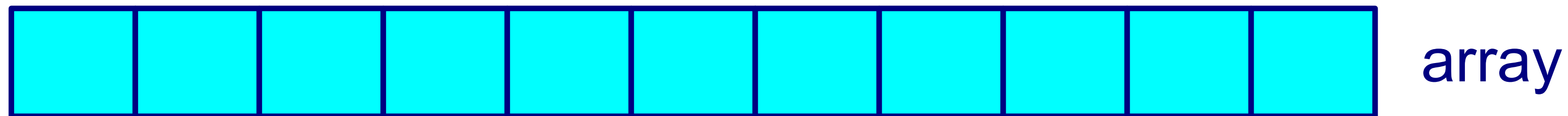
Computer Program

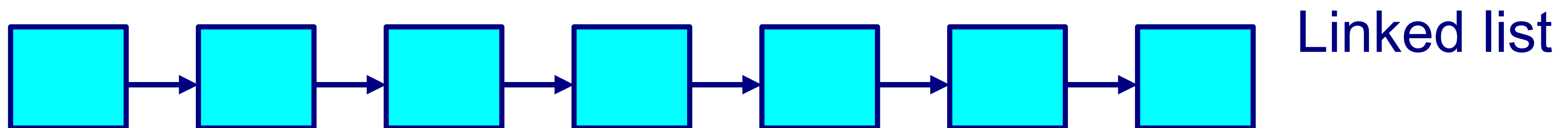


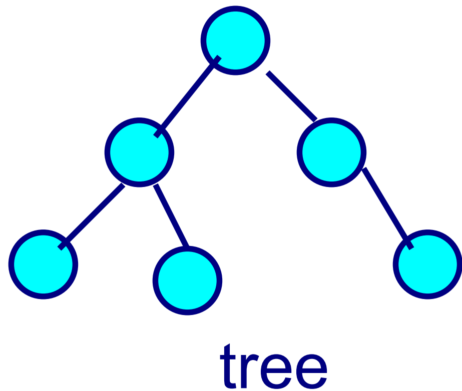
- How to solve the following problems:
  - Input 3 numbers, print out the maximum.
  - Input 30000 numbers, print out the largest 10 numbers.
  - Input the list of cities and the distances among them, find the shortest path from city A to city B

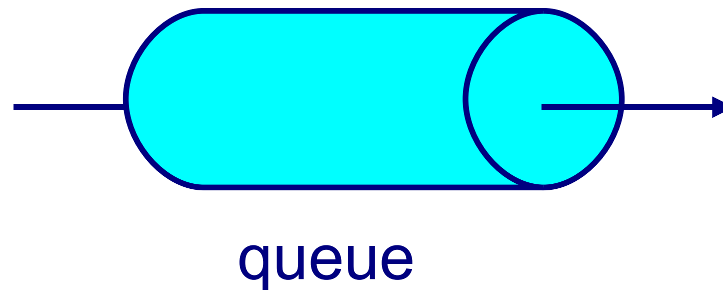
Data Structures

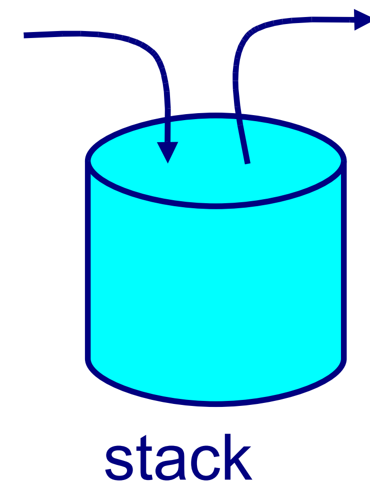
- Data structures let the input and output be represented in a way that can be handled efficiently and effectively.
- Data structures + Algorithms = Programs



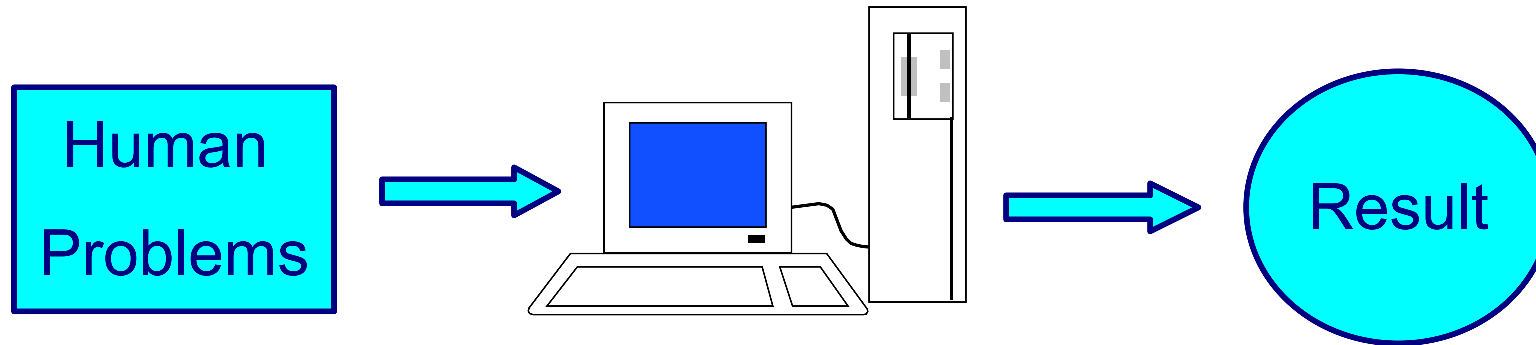


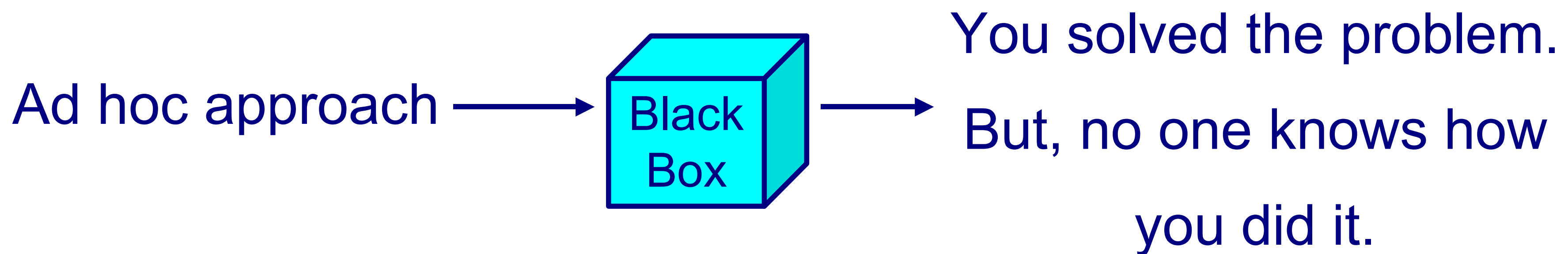






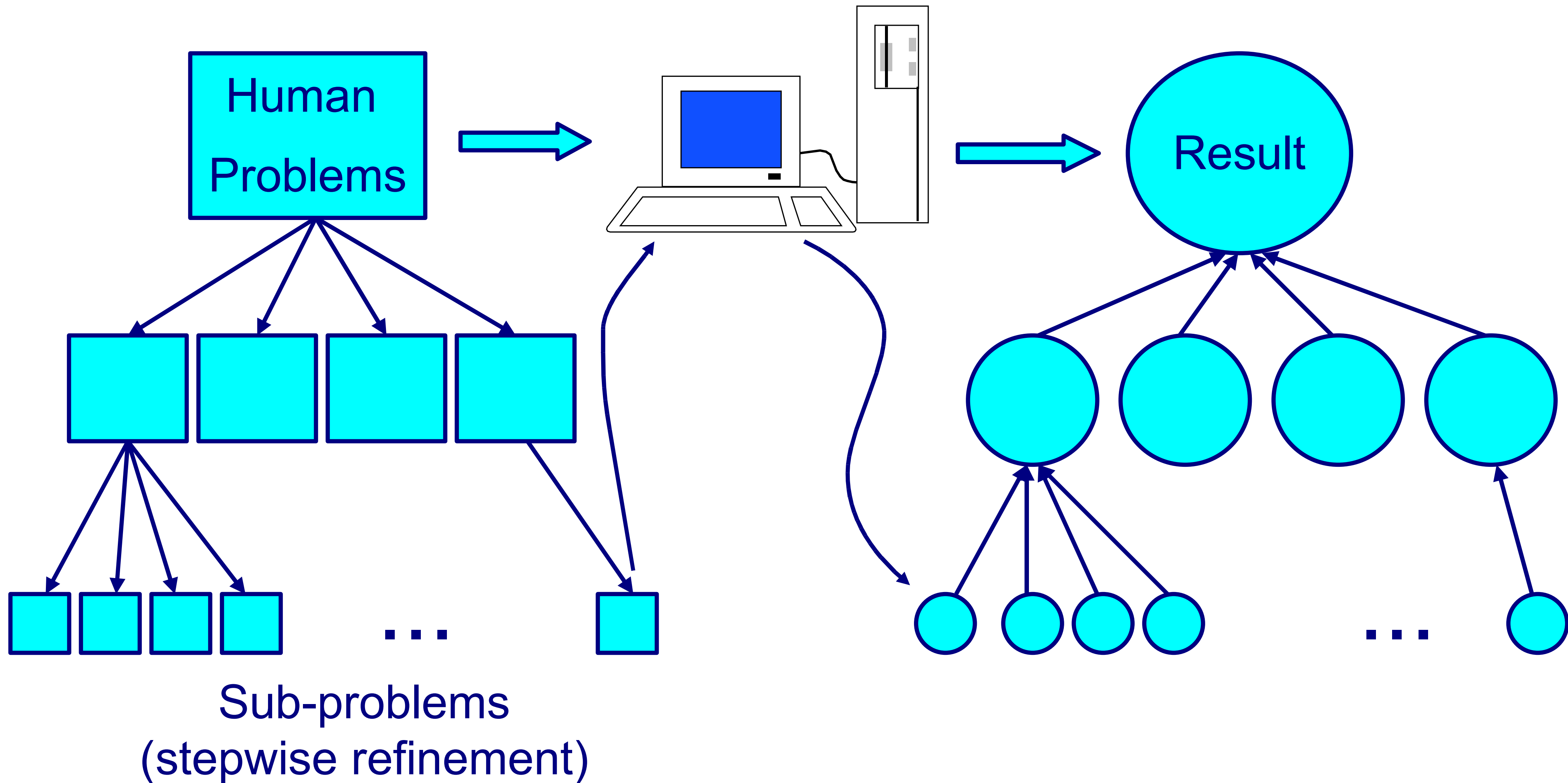
Top - Down Design





Top - Down Design

Systematic approach



Top-Down Design

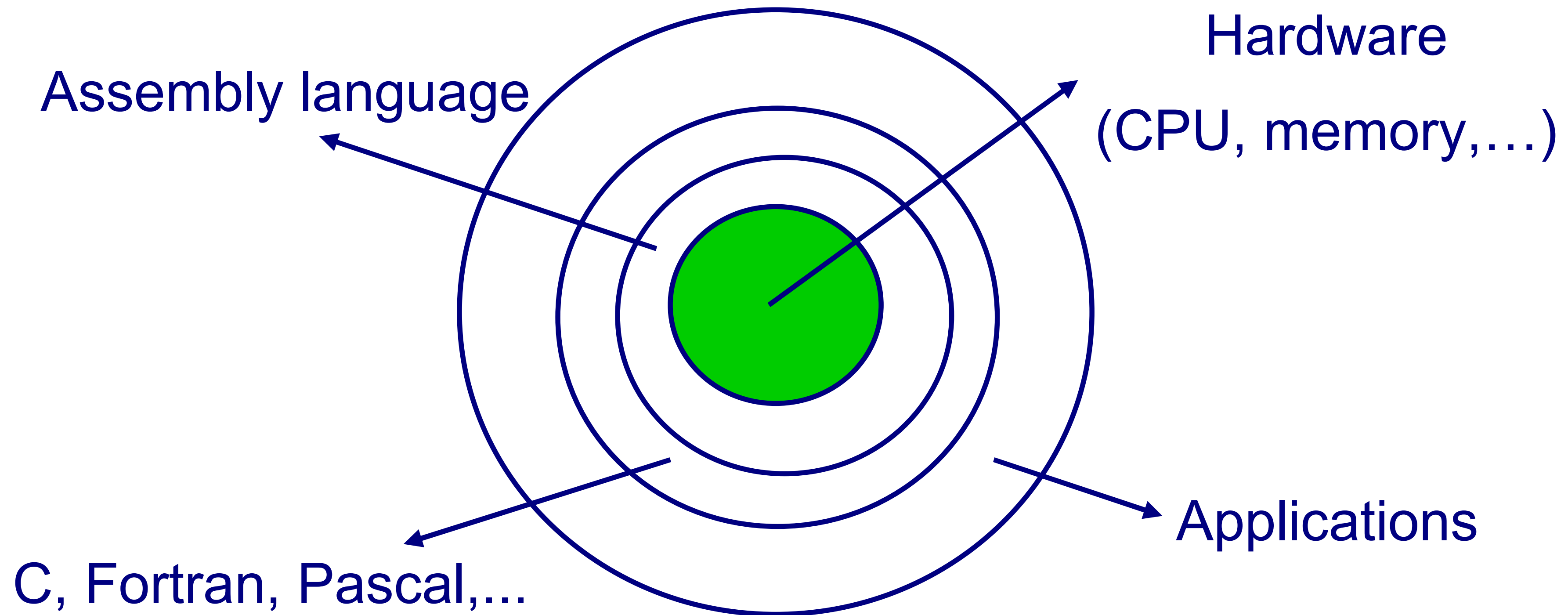
- The algorithm chosen to solve a particular programming problem helps to determine which data structure should be used.
- The data structure selected has a great effect on the details and the efficiency of the algorithm.

Top-Down Design

- The advantages of top-down design include:
  - Top-down design provides a systematic way of solving problems.
  - The resulting solution is modular. The steps can be coded, debugged, modified, and enhanced independently without affecting other steps.
  - The resulting solution is simpler to follow, because one can digest it piece by piece, rather than having to swallow the whole thing at once.
  - Well-designed pieces can be re-used in other tasks.
  - By beginning at the top, one may identify common subprograms from the start and solve them once, instead of redesigning a solution each time one reappears.

Bottom-Up Design - Virtual Machines

The onion model



Bottom-Up Design

- Bottom-up design does not break down into steps that the language can handle. Instead it builds the language up by adding more and more powerful constructs until the language can be used to solve the program directly.
- For example, consider writing a program that reads in, evaluates, and prints out rational expressions:
  - input : (3/28 + 2/7) * 4/3 - 1
  - output: 61/84
- C does not have the capability to represent and manipulate rational numbers directly.
- Therefore, the first step would be to *extend C by adding functions* to implement the rational operations of +, -, *, and /, in addition to writing functions that read in and print out rational values.
- A rational number package may be useful for a wide variety of programs, not just the one currently being written.

Program Design with C

- Usually, programs are designed using a combination of top-down method and the bottom-up method. A programmer starts with the problem, breaks it down into subprograms, then discovers that a particular package of functions would be useful.

Design for Reuse - Separate Compilation

```
#ifndef RATIONALS
#define RATIONALS
#include <stdio.h>
typedef struct {
   int num, denom;
} Rational;

int readRat(Rational *);
int writeRat(Rational);
Rational addRat(Rational,Rational);
Rational subRat(Rational,Rational);
Rational multRat(Rational,Rational);
Rational divRat(Rational,Rational);
#endif RATIONALS
```

- The advantages of separate compilation are threehold:
  - Should a change be necessary, only the affected file need to be recompiled. After recompilation all files are relinked together. This process usually takes much less time than recompiling everything.
  - The unit can be used in different programs without modifying any of its code.
  - The header file contains all the information necessary to use the package of functions.

Abstract Data Types

- There is one major improvement that can be made to this method of designing programs.
- Often the operations in a package do not depend on the particular data types manipulated by the main program.
- It is useful to implement such a package of operations so that the code does not depend on the particular data types.

Abstract Data Types

- A data type whose operations are independent of its specific characteristics is called an abstract data type (ADT).
- For example, if a SET is an ADT, we can define:
  - **`size()`**: return the number of elements in the set
  - **`isEmpty()`**: return if the set is empty or not
  - **`insertElement()`**, **`isMember()`**, **`union()`**, **`intersect()`**, **`subtract()`**, **`isEqual()`**.
  - All the above is independent of what are in the SET.

Abstract Data Types

- Consider the following two programming tasks:
  - Task 1. A concordance program. This program will read in text from the standard input and then print out each word that occurs in the input along with the number of times that the word occurs in the input.
  - Task 2. A banking program. This program will read in a list of transactions. Each transaction consists of an account number, a date, and an amount. Then a statement is printed for each account, showing the balance and the list of transactions for that account listed in the order in which they were read into the program. For the sake of simplicity, assume that accounts have at most ten transactions, all account balances are initially zero, and that the account numbers, dates, and amounts are all integers.

Abstract Data Types

- At first glance these two programs seem to have nothing to do with each other.
- But, let us take a closer look.

Abstract Data Types

- Top-down program breakdowns

Concordance Program:

While not at the end of the input

Read in a word.

Look up the word (and its associated count) in some data structure

If the word is found, then increment its count and store the new value back in the data structure.

If the word is not found, then add it to the data structure with an associated count of 1.

For each word in the data structure,

print out the word and its associated count.

Abstract Data Types

- Top-down program breakdowns

Banking Program:

While not at the end of the input

Read in a transaction

Look up the account number (and its associated balance and list of transactions) in some data structure.

If the account number is found, then add the amount to the account's balance, and add the transaction to the account's list of transactions and store this new information back in the data structure.

If the account number is not found, then add the account number to the data structure along with the transaction amount as the balance, and a list of transactions containing just the one transaction.

For each account in the data structure,

print out the account number, its balance, and its list of transactions.

Abstract Data Types

- The common part:
  - Routine for finding some data record based on a key
  - Routine for replacing the data with new information
  - Routine for creating a new record with a new key
- These routines should not depend on
  - the type of the data stored
  - the type of the key used
- But should be affected by:
  - how the data records are stored
  - what algorithms are used to find and retrieve the data

Abstract Data Types

- The process of separating what operations are done to the data and the means of storing the data from the particular type of data is the basis of an abstract data type.
- An abstract data type can be defined by the operations that are done on the data regardless of its type.

Abstract Data Types

- Common operations for simple database ADT
  1. Find the data associated with a particular key.
  2. Replace the data associated with a key with new data.
  3. Add a new key and its associated data.
  4. Perform an operation on each key and its associated data.

Abstract Data Types

- The use of ADT divides the programming task into two steps:
  1. Implement the operations of the ADT, choose a particular data structure to represent the ADT, and write the functions to implement the operations.
  2. Write the main program which calls the functions of the ADT.

Abstract Data Types

- The main program can access the data in the ADT only by calling the functions of the ADT; it cannot directly access or change any of the values stored in the internal data structures of the ADT.
- Does it make the programming more difficult?
- This is actually the key to the usefulness of an ADT.

The Simple Database ADT

```
int findRec(dbkeytype key,dbdatatype *data,indextype *idx);
/* Action: Looks up key in the simple database.
   If a record with that key exists, findRec returns 1,
   data is the data of the record, and idx is an indicator
   of where the record is stored. If no record is found,
   then findRec returns 0, idx is an indicator of where
   the record should be put, and data is undefined. */

int createRec(dbkeytype key,dbdatatype data,indextype idx);
/* Precondition: idx should indicate where the new record is to go.
   Action: Adds a new record with key and data to the simple database.
   Returns 1 for success, 0 for failure (database is full). */

void setRec(indextype idx,dbdatatype data);
/* Precondition: idx should indicate where the new record is to go.
   Action: Changes the data of the record indicated by idx to be data. */

void eachRec(void (*fn)(dbkeytype key,dbdatatype data));
/* Action: Applies the function fn to each record (using key and data)
   in the simple database. */
```

- **`key`** is:
  - the word as **`char[21]`** in a concordance program
  - account number as an integer in a banking program

The Simple Database ADT

- The main program

```
/* concordance.c */
# include <stdio.h>
# include "database.h"

void printCount(char *word,int n)
{  printf("%20s %5d\n",word,n); }

int main()
{ char word[21];
  int count;
  indextype idx;
  while ( scanf ( "%20s " , word) ==1 ) {
   if (findRec(word, &count,&idx))
     setRec ( idx, count + 1 ) ;
   else if (!createRec (word, 1,idx))
     printf("Error: cannot add word  %s. Database full.\n”,word);    }
   eachRec(&printCount);
   return 0;
}
```

The Simple Database ADT

- **`database.h`** for an array implementation

```
/* database.h */
#ifndef DATABASE
#define DATABASE
#include <stdio.h>
#include "words.h"

typedef struct dbrec {
  dbkeytype key;
  dbdatatype data;
} Dbrec;

typedef Dbrec *indextype;

int findRec(dbkeytype, dbdatatype *,indextype *);
int createRec(dbkeytype, dbdatatype, indextype);
void setRec(indextype, dbdatatype);
void eachRec(void (*)(dbkeytype, dbdatatype));
#endif
```

The Simple Database ADT

- Definitions of data types and operations for the concordance program

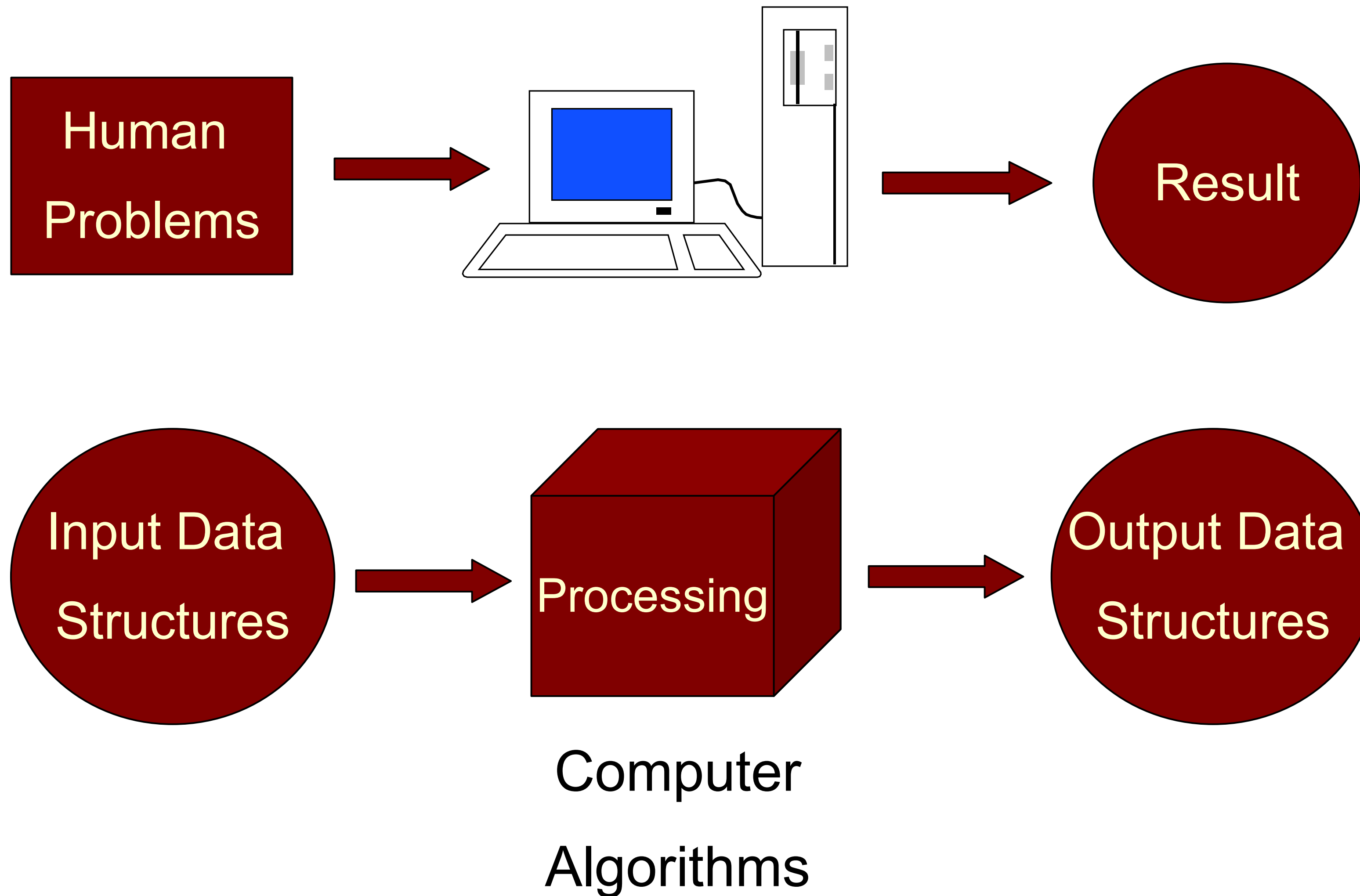
```
/* words.h */
#ifndef WORDS
#define WORDS
#include <stdio.h>
#include <string.h>

typedef char dbkeytype[21];
typedef int dbdatatype;

#define comparedbkey(s1,s2)         ((int)(strcmp((s1),(s2))))
#define copydbkey(s1,s2)            (strcpy((s2),(s1)))
#define copydbdata(d1,d2)           ((d2)=(d1))
#endif
```

List ADT

- List operations

```
void initList(List *list);
/* Action: Creates an empty list */
int addList(List *list, listdatatype data);
/* Action: Adds data to the list.
   Returns 1 on success, 0 otherwise (the list is full) */
void eachElement(List list, void (* fn)(listdatatype));
/* Action: Applies the function fn to each element of the list */
```

Algorithm Design

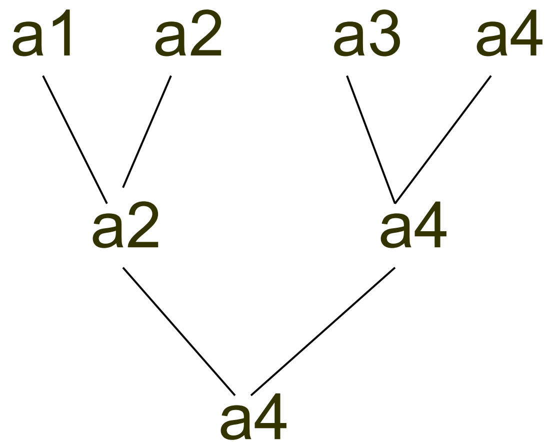


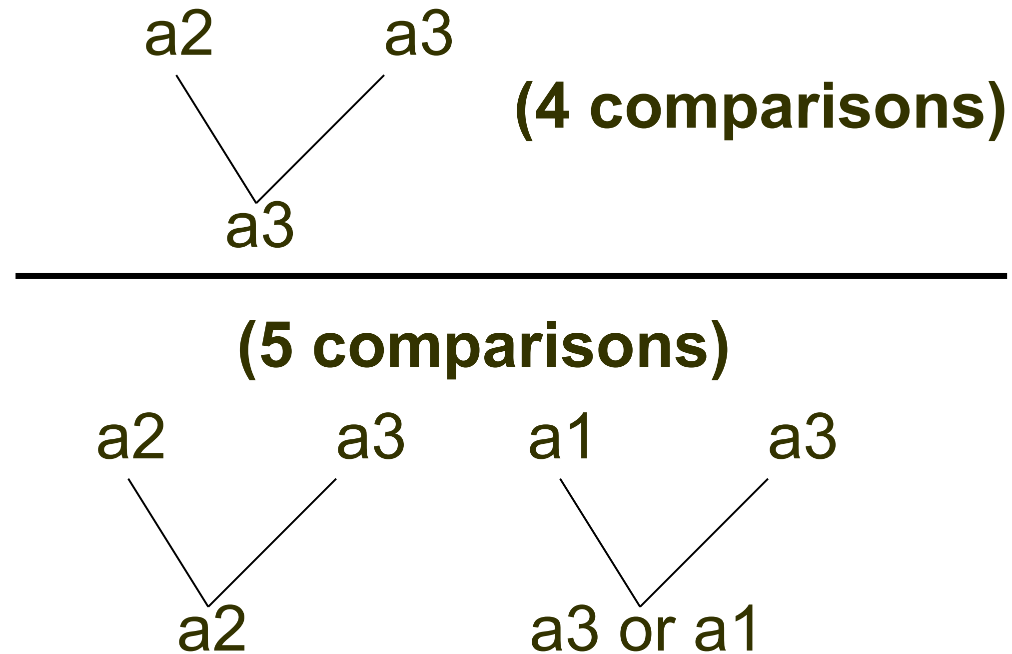
Algorithm Design

- For a problem, what is an optimal solution ?
  - CPU time
  - Memory
- Example: Given 4 numbers, sort it to nonincreasing order.
- Method 1: Sequential comparison
  1. Find the largest (3 comparisons)
  2. Find the second largest (2 comparisons)
  3. Find the third largest (1 comparisons)
  4. Find the fourth largest
- A total of 6 comparisons

Algorithm Design

- Example: Given 4 numbers, sort it to nonincreasing order.
- Method 2: Somewhat clever method



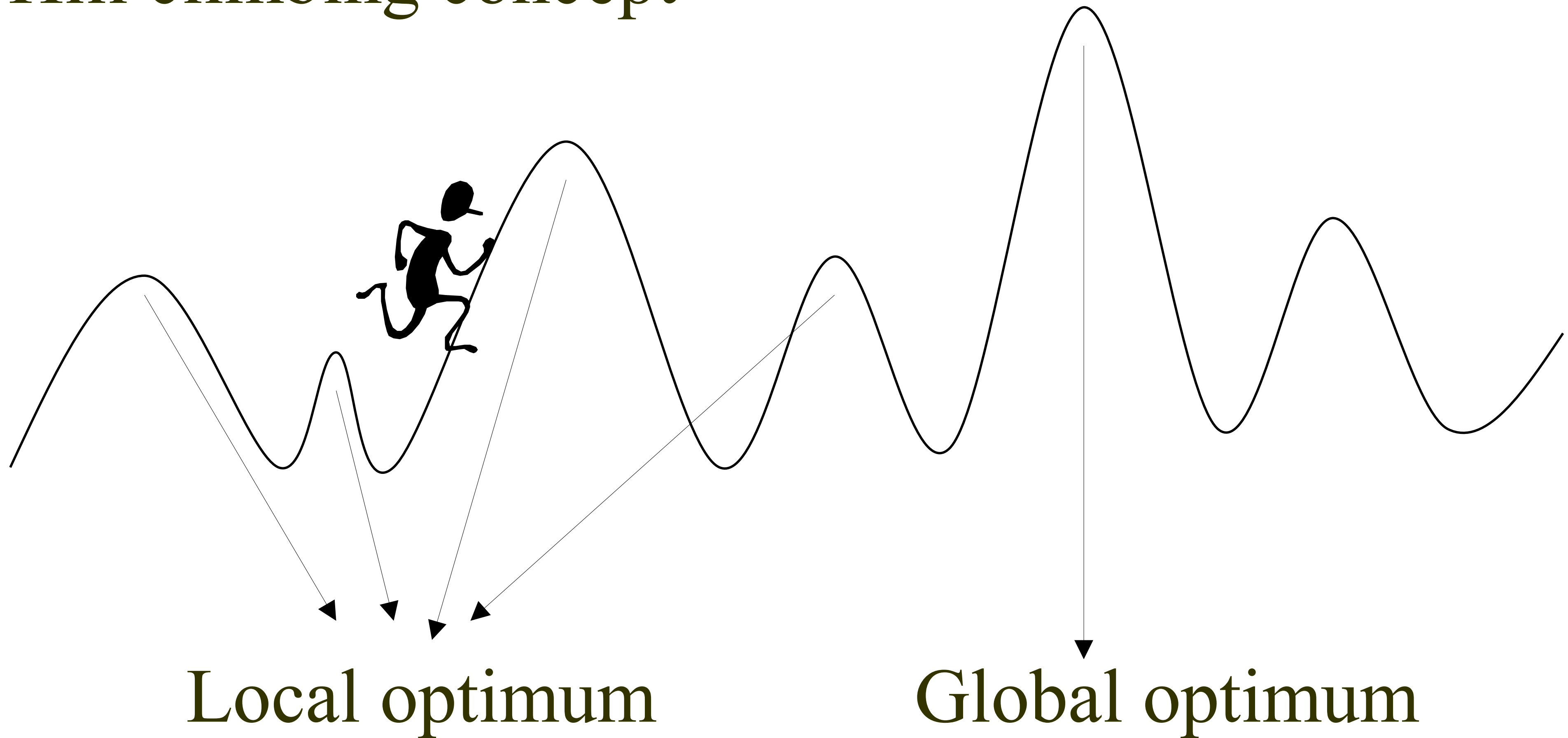


Greedy Algorithms

- A greedy algorithm takes an action that seems the best at the given time, without consideration of future actions.
- Example: Coin changing problem
- Make change for any amount from $0.01 to $0.99 using the fewest number of coins. The available coins are $0.5, $0.25, $0.1, $0.05, and $0.01.
  - $0.94=$0.5+$0.25+$0.1+$0.05+4*$0.01
  - A total of 8 coins
  - Greedy algorithm works for U.S. coins. But not necessarily so for others. See Chapter 2 Exercise 1 in D&S

Greedy Algorithms

Hill climbing concept



Divide-and-Conquer Algorithms

Example:

Finding the minimum number in a list of numbers

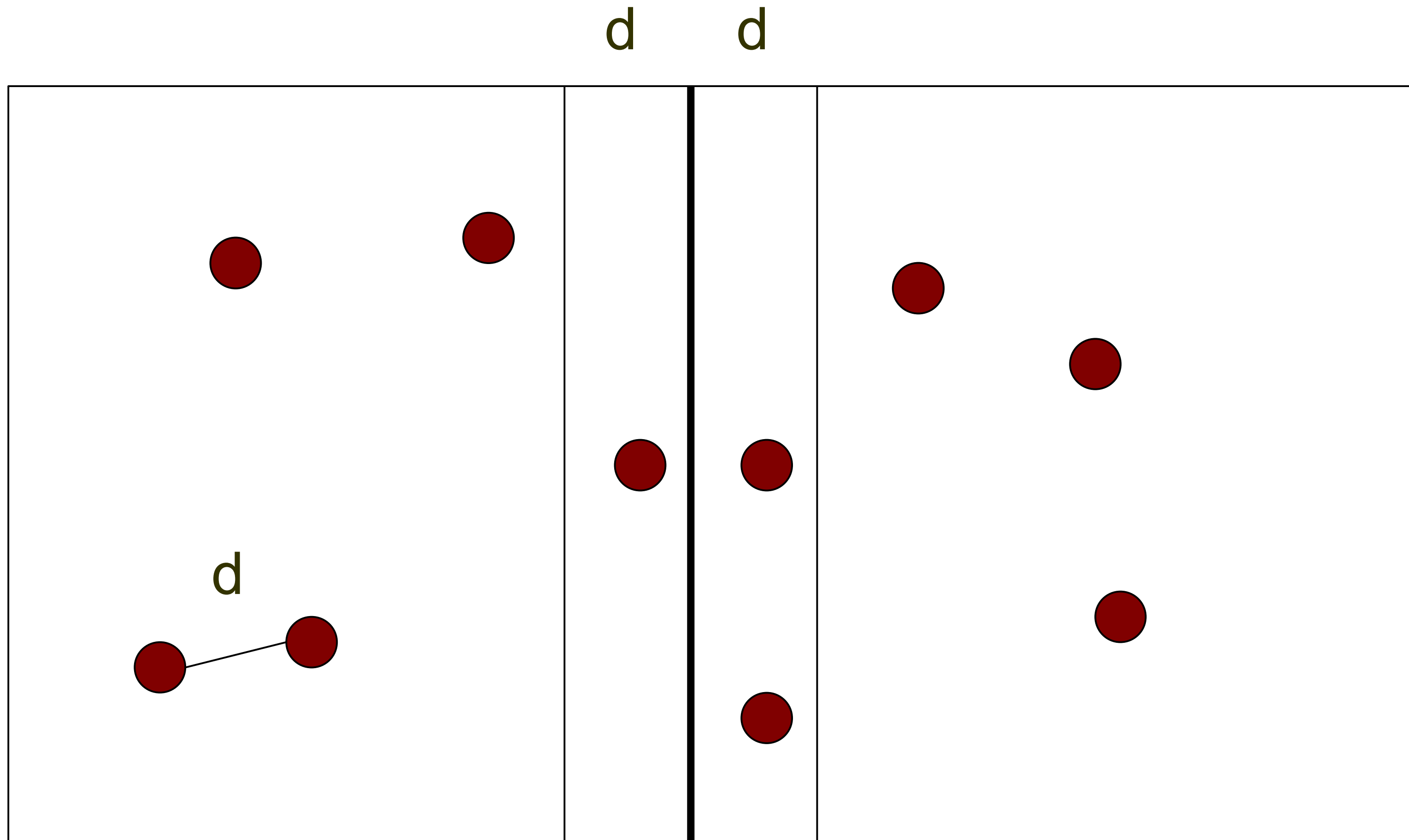
```
int minimum(int a[],lower,upper)
{
 if (upper==lower) return a[upper];
 if ((upper-lower) == 1) return min(a[upper],a[lower]);
 else return min(minimum(a,lower,(lower+upper)/2),
                 minimum(a,(lower+upper)/2+1,upper));
}
```

Merge sort

Quick sort

Divide-and-Conquer Algorithms

- The closest pair problem



- For each point near the border, there can be at most six candidates for the closest neighbor across the border.

Dynamic Programming Algorithms

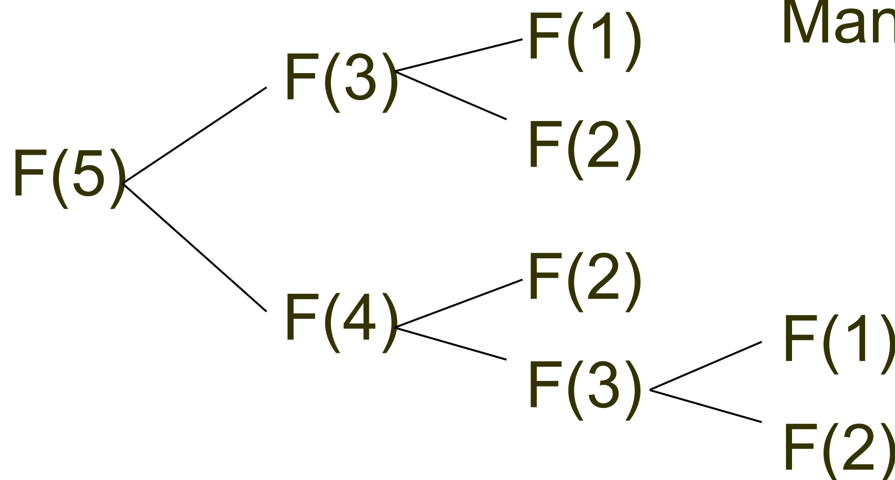
- Often a problem can be solved by divide-and-conquer, but it may turn out to be *an inefficient algorithm* because *much work is duplicated* when solving the subproblems (or because the conquer (merge) step is too complicated).

Dynamic Programming Algorithms

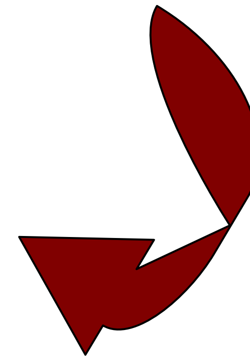
- The Fibonacci numbers

$$F(i) = \begin{cases} 1 & \text{if } i = 1, \\ 1 & \text{if } i = 2, \\ F(i-2) + F(i-1) & \text{if } i > 2. \end{cases}$$

```
int Fibonacci(n)
{
  if (n<2) return 1;
  else return Fibonacci(n-1)+Fibonacci(n-2);
}
```



Many repetitive calculations



Dynamic Programming Algorithms

- The Fibonacci numbers
- Better implementation

```
/* C99 only */
int Fibonacci(int n)
{
   int data[n];
   data[0]=1; data[1]=1;
   for (j=2; j<n; j++) data[j]=data[j-2]+data[j-1];
   return data[n-1];
}
```

- Example of one-dimensional dynamic programming

Dynamic Programming Algorithms

- The number of combinations

$$\binom{n}{r} = \frac{n!}{r! \times (n-r)!}$$

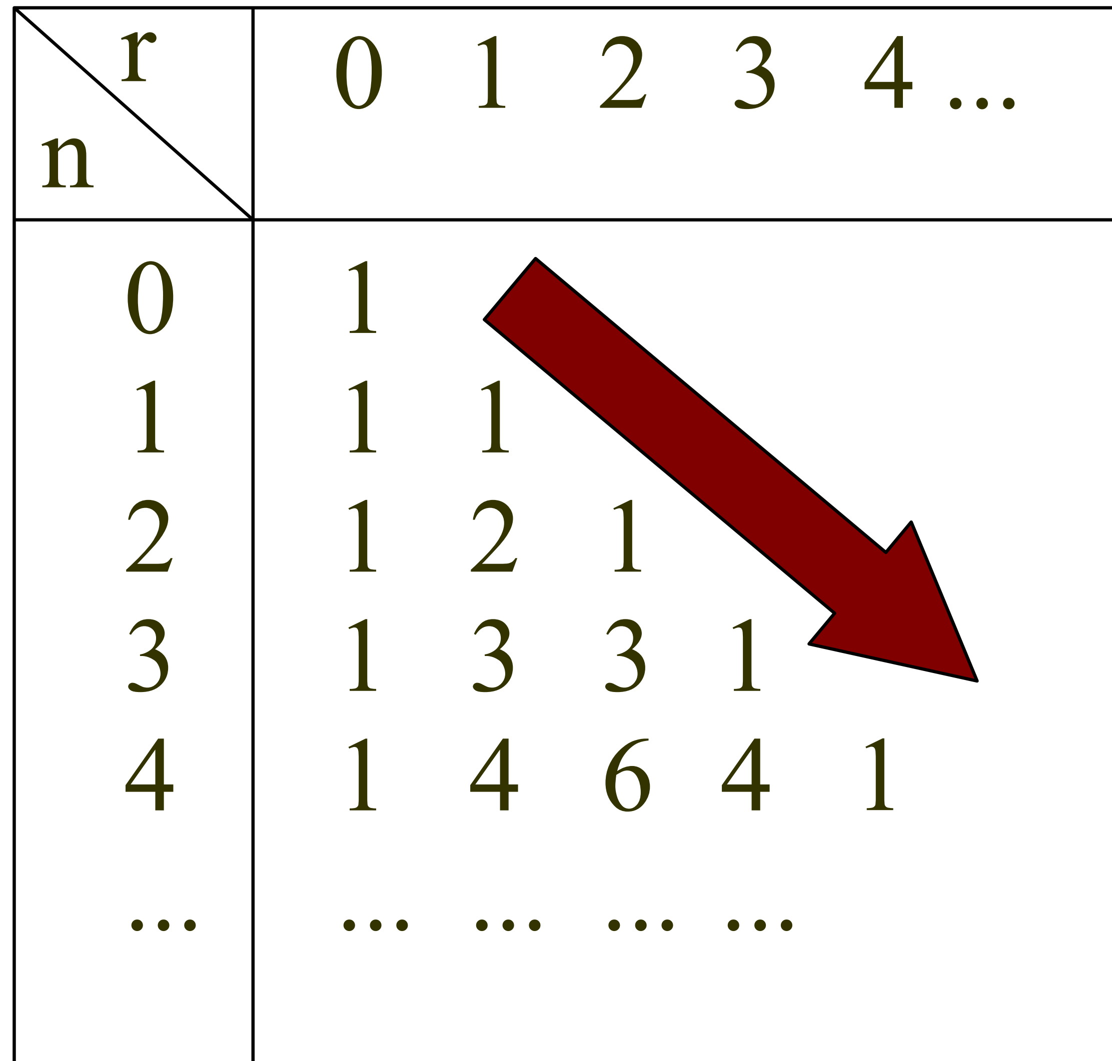
$$\binom{n}{r} = 1 \text{ if } r = 0 \text{ or } n = r$$

$$\binom{n}{r} = \binom{n-1}{r} + \binom{n-1}{r-1} \text{ for } 0 < r < n.$$

Dynamic Programming Algorithms

- The number of combinations
- The Pascal's triangle

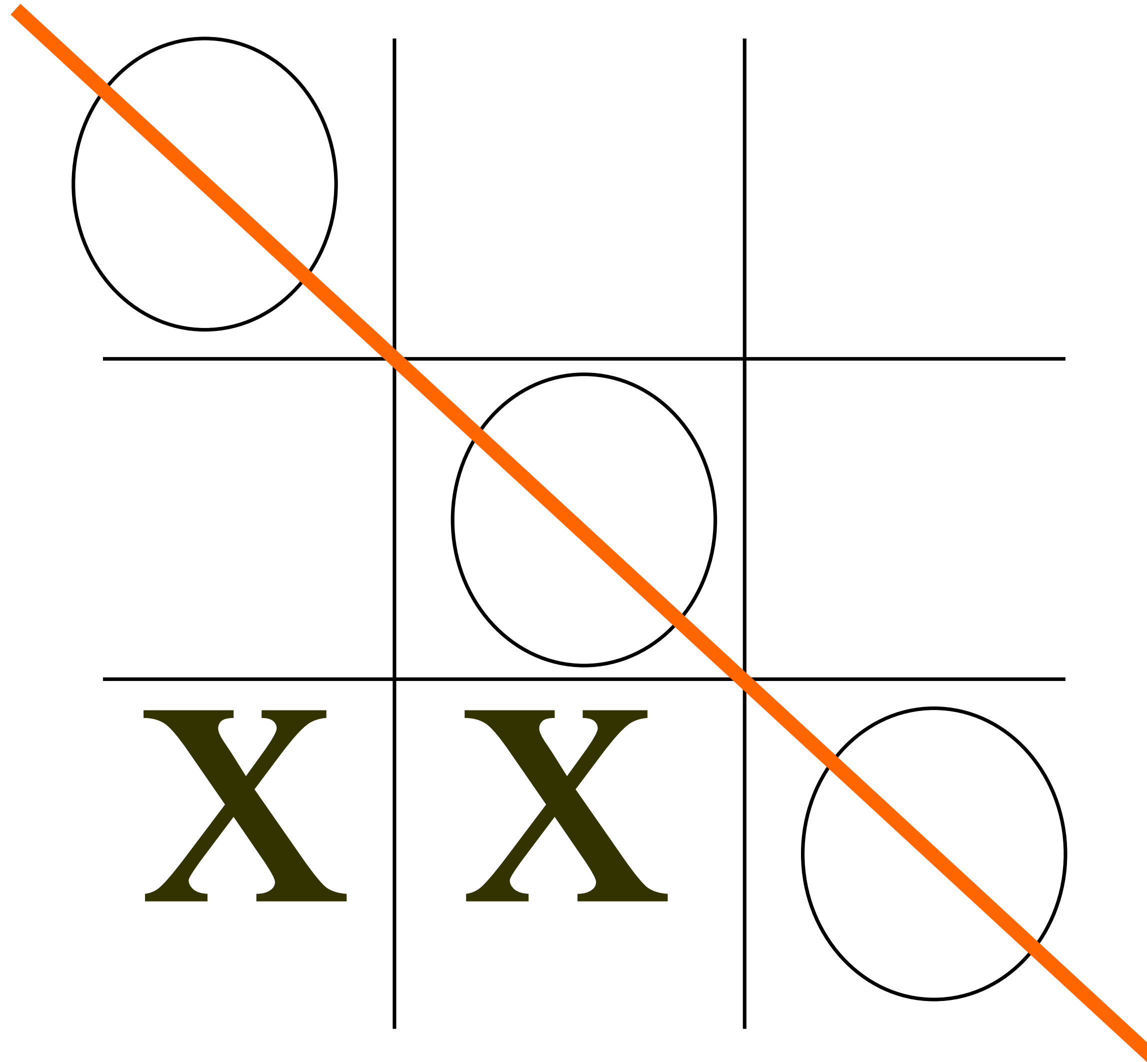
$$\binom{n}{r}$$



n \ r	0	1	2	3	4 …
0	1				
1	1	1			
2	1	2	1		
3	1	3	3	1	
4	1	4	6	4	1
…	…	…	…	…	

Backtracking Algorithms

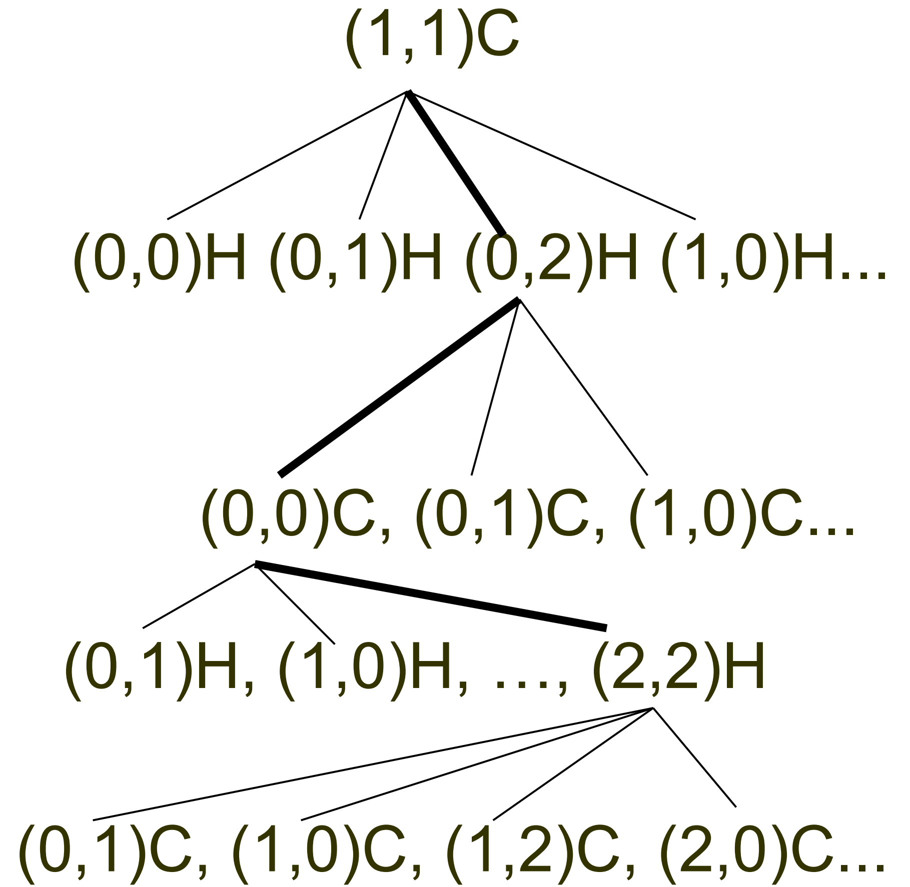
- The tic-tac-toe game
- How can a computer play the game?



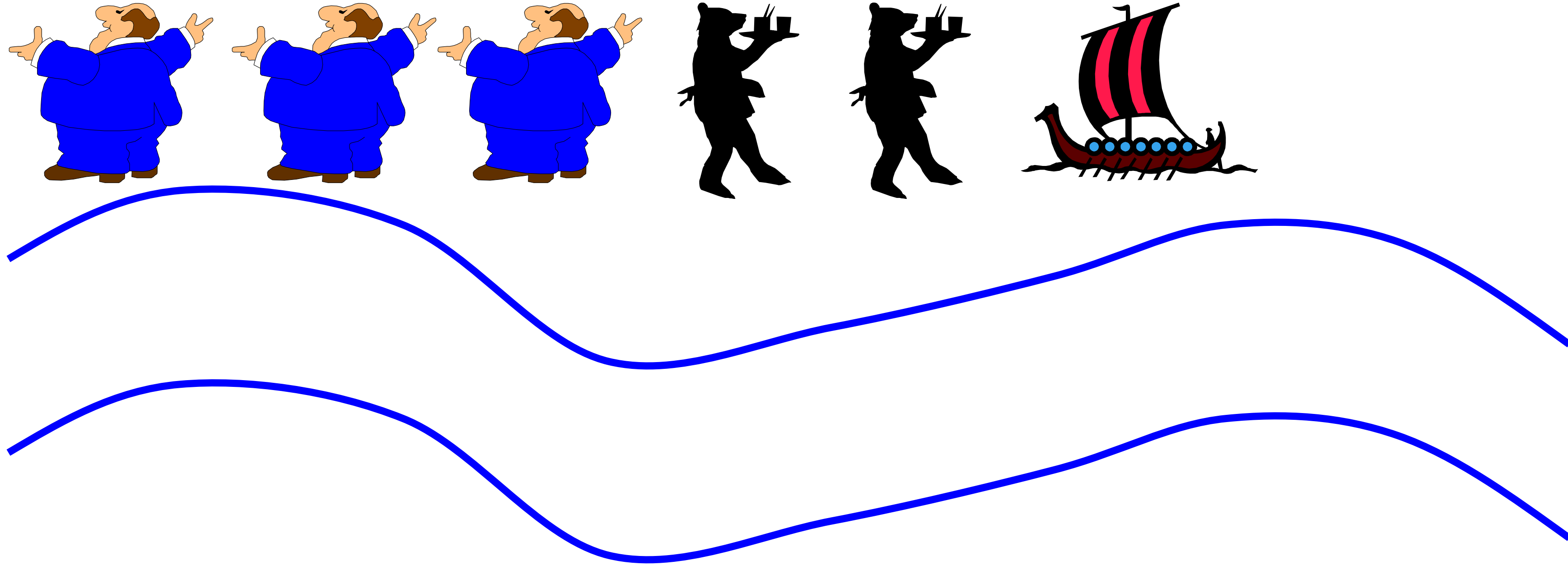
Backtracking Algorithms

	0	1	2
0	O		X
1		O	
2			X

O : Computer X : Human



Backtracking Algorithms



- 3 missionaries and 2 cannibals want to cross the river
- Condition:
  1. A boat can take one or two (must include a missionary)
  2. At any time, on either bank, the number of missionaries must not be less than the number of cannibals.