COURSE INFORMATION

Text:

Mark Allen Weiss (1998) *Data Structures and Problem Solving Using JAVA*,
Addison-Wesley ISBN 0-201-54991-3

Text is *recommended* but not required. Class lectures will generally cover material not in the text. The text will be used to supplement the lectures. The following Chapters will *not* be covered in this class: Chs 9, 10, 12, 18, 20-23

Evaluation:

Available grades:

non-completion: Incomplete, Withdraw, etc.
completion: A A- B+ B B- C

A:	reserved for superior performance
A- or B+:	expected grade for conscientious performance
B:	adequate work
B-:	barely adequate
C:	equivalent to failing

Grading Options:

1. Performance Quality: attendance, participation, assigned exercises
2. Grading Contract: specify a set of behaviors and an associated grade.
3. Self-determined: negotiate with instructor

Discussion:

If you prefer a clearly defined agenda, if you do well with concrete task assignments,
or if you need a schedule of activities for motivation, then **Option 1** is a good idea.
If you already understand the field, if you plan to excel in a particular area,
or if you need clear performance goals for motivation, then **Option 2** is a good idea.
If you are not concerned about grades, if you intend to do what you choose anyway,
or if you are self-motivated, then **Option 3** is a good idea.
I will notify any student who is not on a trajectory for personal success.

Languages and Style:

The course will emphasize how to think about, design, and select data structures and algorithms for particular applications. It is expected that students will copy algorithms and data structures from reliable sources rather than design their own. Students may use the OS and programming languages of the choice for programming exercises. Most assignments will have an implementation component.

Course Syllabus

Week 1: Text: Part I
Overview: mathematical domains, modeling, computational tools, theory of representation, hardware and software architectures.
Overview of mathematical structures: sets, logic, algebras, numbers, groups, graphs.
Curriculum planning.
Assignment 1: Outline your knowledge of data structures and algorithms.

Week 2: Text: Part I, Ch 6.
Overview of Java
Overview of data structures: stacks, arrays, lists, trees, graphs.
Assignment 1 due.

Week 3: Text: Chs 15, 16, 17
Abstract data structures, the String, Rational Numbers. Case study: the square.
Assignment 2: Build a type hierarchy for data structures

Week 4: Text: Ch 7
Overview of algorithms: sequential and parallel processes, recursion, tractability.
Programming paradigms.
Assignment 3: Implement an ADS for Strings
Assignment 2 due

Week 5: Text: Ch 5
Algorithm analysis.
Assignment 3 due.
Assignment 4: Implement an ADS for Squares

Week 6: Text: Chs 8, 23
Sets and Sorting. Case study: the set
Assignment 4 due.

Week 7: Text: Chs 11, 19
Hashing and parsing. Search algorithms.
Strings, sequences and streams. Pattern matching. Case study: Mathematica
Assignment 5 (major): Abstract control structures

Week 8: Text: Ch 14
Graphs. Graph processing.
Case study: satisfiability of boolean forms

Week 9: Text: Ch 13
NP-completeness, algorithms for intractable problems.
Interpreted languages.
Assignment 5 due

Week 10:
Closure.

Context and Hierarchy

Essential Concepts of the Course:

Complementarity:
the intimate relationship between data structure, algorithm and computational architecture.
Abstraction hierarchies:
from conceptualization through mathematics to implementation.
Programming paradigms:
the languages of design, modeling and implementation
Implementation hierarchies:
trading off between design and implementation efficiencies

Abstraction Hierarchy

conceptualization	(real world specific)
mathematical model	(symbolic)
implementation model	(software specific)
process model	(hardware specific)

Implementation Hierarchy

conceptualization/design (quasi)-language
very-high-level (task specific) programming tool
high-level programming language
low-level programming language
opcodes and machine language
high-level synthesis
low-level synthesis

Naming Domains

data types
constants/grounds
operators (functions, predicates)
program execution types (memory location, signal transitions)
resources (memory, operator circuits, i/o devices)
constraints (equations)

Data Structures

bit	array (eg byte, word)
string	queue
stream	linked list
struct	object

Programming Paradigms

procedural	C, Pascal, COBOL
functional	LISP, ML
recursive	LISP, Prolog
logical/declarative	Prolog
constraint-based	Prolog III
object-oriented	Smalltalk, Java, C++
rule-based	OPS5
mathematical	Mathematica

Models of Computation

table lookup
register manipulation
predicate calculus
lambda calculus, combinators
recursive function theory
term-rewriting
graph-rewriting
matrix algebra
relational database
cellular automata

Mathematical Structures

propositional calculus (boolean algebra)
- truth symbols
- propositional symbols (binary variables)
- connectives (and, or, not)
- interpretations

predicate calculus
- truth symbols
- constant symbols
- variable symbols
- function symbols
- predicate symbols (relations)
- quantifiers

equality and orderings
non-negative integers
sets, bags (multi-sets)
strings, trees, lists
tuples (structs)
graphs

Mathematical Abstractions

Relations

base
atom
compound
structure

reflexive	all x | (x,x) inR
symmetric	if (x,y) inR, then (y,x) inR
transitive	if (x,y) inR and (y,z) inR, then (x,z) inR
antisymmetric	if (x,y) inR and (y,x) inR, then x = y
trichotomy	(x,y) inR xor (y,x) inR xor x=y
irreflexive	not reflexive
asymmetric	not symmetric

Functions (binary relations with existence and uniqueness)

base
compound
structure

identity	Id op A = A op Id = A
inverse	A op iA = iA op A = Id
associative	(A op B) op C = A op (B op C)
commutative	A op B = B op A
distributive	A op1 (B op2 C) = (A op1 B) op2 (A op1 C)
idempotent	A op A = A

Equations (equivalence relations)

theorems (proved)
axioms (assumed)
generate
    base, atom, compound
unique
    base, compound

Wonderful Computer Science Books

Every subfield of Computer Science has several journals, dozens of text books, and hundreds of technical books. The web makes this situation much worse by providing course notes and technical discussion from hundreds or thousands of practitioners. Below is the single best book or two (IMHO) in most of areas covered in this class.

Data Structures, Algorithms and Programming

H. Abelson and G.J. Sussman (1996)
Structure and Interpretation of Computer Programs, Second Edition
McGraw-Hill. ISBN 0-07-000484-6

Comprehensive Reference on Algorithms

T.H. Cormen, C.E. Leiserson, and R.L. Rivest (1990)
Introduction to Algorithms
MIT Press. ISBN 0-07-013143-0

Very High-level Programming

S. Wolfram (1996)
The Mathematica Book, Third Edition
Wolfram Media, Cambridge U. Press. ISBN 0-521-58889-8

Mathematical Models of Data Structures

Z. Manna and R. Waldinger (1985)
The Logical Basis for Computer Programming, Volume 1
Addison-Wesley. ISBN 0-201-18260-2

Old-style Procedural Algorithms

R. Sedgewick (1988)
Algorithms in C++, Third Edition
Addison-Wesley. ISBN 0-201-35088-2

Theory of Computation Complexity

J.E. Savage (1998)
Models of Computation
Addison-Wesley. ISBN 0-201-89539-0

Programming Theory

N.D. Jones (1997)
Computability and Complexity from a Programming Perspective
MIT Press. ISBN 0-262-10064-9

Philosophy of Computation

B.C. Smith (1996)
On the Origin of Objects
MIT Press. ISBN 0-262-69209-0

Programming Languages

B.J. MacLennan (1999)
Principles of Programming Languages, Third Edition
Oxford. ISBN 0-19-511306-3

M.L. Scott (2000)
Programming Language Pragmatics
Morgan-Kauffman. ISBN 1-55860-442-1

Computer Architecture

J.L. Hennessy and D.A. Patterson (1996)
Computer Architecture: A Quantitative Approach, Second Edition
Morgan-Kaufmann. ISBN 1-55860-329-8

Compilers

S.S. Muchnick (1997)
Advanced Compiler Design and Implementation
Morgan-Kaufmann. ISBN 1-55860-320-4

Discrete Mathematics

W.K. Grassmann and J. Tremblay (1996)
Logic and Discrete Mathematics: A Computer Science Perspective
Prentice-Hall. ISBN 0-13-501206-6

Understanding Computing in Simple Language

R. P. Feynman (A.J.G. Hey and R.W. Allen, eds) (1996)
Feynman Lectures on Computation
Addison-Wesley. ISBN 0-201-48991-0

Computer Science Culture

ACM Turing Award Lectures: The First Twenty Years (1987)
ACM Press, Addison-Wesley. ISBN 0-201-07794-9

Seminal Algorithms Text (recently updated!)

D.E. Knuth (1997, 1997, 1998)
Fundamental Algorithms, Third Edition, volume 1 of *The Art of Computer Programming*
Seminumerical Algorithms, Third Edition, v. 2 of *The Art of Computer Programming*
Sorting and Searching, Third Edition, v. 3 of *The Art of Computer Programming*
Addison-Wesley, ISBNS 0-201-89683-4, 0-201-89684-2, 0-201-89685-0

Undergraduate Introduction to Data Structures and Algorithms

F.M. Carrano, P. Helman and R. Veroff (1998)
Data Abstraction and Problem Solving with C++
Addison-Wesley. ISBN 0-201-87402-4

Mark Allen Weiss (1998)
Data Structures and Problem Solving Using JAVA
Addison-Wesley ISBN 0-201-54991-3

Historical First Textbooks

Nicklaus Wirth (1976)
Algorithms + Data Structures = Programs
Prentice-Hall ISBN 0-13-022418-9

A.V. AHo, J.E. Hopcroft and J.D. Ullman (1983)
Data Structures and Algorithms
Addison-Wesley ISBN 0-201-00023-7

Computer Art

A.M. Spalter (1999)
The Computer in the Visual Arts
Addison-Wesley. ISBN 0-201-38600-3

Assignment I

Mapping Your Knowledge of Data Structures and Algorithms

One or two pages to be handed in to the instructor at classtime.
Time allocation: two hours thinking, two hours writing

Construct an outline of what you know about data structures and algorithms.

A **data abstraction** is a way to organize computational information and consequently computer memory. It usually consists of a storage representation and a set of operations to construct, access, modify, delete, deconstruct, test for membership, and/or display stored instances.

An **algorithm** is the structured computational process which converts data into a solution to a particular set of problems. Useful algorithms usually apply to large classes of problems.

These concepts can be understood at **different levels of the design hierarchy**. Data structures and algorithms can be seen

- as ways of implementing low level computational processes,
- as ways of structuring an implementation in a programming language,
- as ways of organizing pseudo-code in preparation for implementation,
- as ways of constructing the mathematical model of a problem, and
- as ways of thinking about and coming to understand a problem space.

How to outline your knowledge:

There are many different ways to collect and organize what you know about a particular topic. The most important thing to recognize is that each person is unique and has a unique understanding of the world. Therefore you should use an outlining technique which feels most comfortable to you. Some choices include

- list major topics and minor topics, similar to the chapter organization of a textbook
- collect words which you can define, and how they relate to each other, similar to a semantic network of object nodes and relational links
- form clusters of similar ideas
- rank the topics covered in the textbook in order of your confidence of understanding
- write down all the things that you have heard about but do not understand
- copy someone else's organization of the topic, marking what you understand and don't
- draw a picture of what you see when you visualize the topic

Remember: Outlines are short. This assignment is not asking you to demonstrate what you know, only to indicate strong and weak areas of knowledge. You do not need to include any form of justification, rationale, or documentation.

Final suggestion: It is often very useful to indicate the degree of confidence you may have for your understanding of a particular topic, as well as the degree of understanding itself. Be sure to test your understanding by asking things like:

- what is the definition? how is this used?
- have I ever actually used this in an implementation? was it successful?
- do I know when not to use this? do I know how to select between alternatives?

Assignment 2: Data Structure Hierarchy

You will not be turning in this assignment.
Time allocation (max): thinking, 2 hour; mapping, 3 hours

Build a type hierarchy for the common data structures.

Many data structures depend on other data structures for their definition. For example, rational numbers depend upon integers, since rational numbers are composed of two integers.

Here is *a listing of most common data types*. Can you construct an inheritance hierarchy which defines their dependencies? You can claim that some entries are not data structures. It will also help if you divide the task into subgroups such as elementary data structures, efficient storage structures, mathematical structures, exotic structures, implementation hacks, etc.

bit
byte
bit stream
character
string
stream

truth value
boolean (proposition)
propositional sentence
function
relation
equality relation
partial ordering relation
total ordering relation

set
bag (multiset)
infinite set
ordering (ranking)
non-negative integer
integer
rational number
real number (specific precision)
complex number
random number

object
class
pattern
persistent object

array
sequence
list
linked list
doubly linked list
association list (bucket)
circular list

stack
queue
priority queue
vector
matrix
table
dictionary

tree
binary search tree
balanced binary tree
red-black tree
B-tree
splay tree
binomial heap
fibonacci heap
graph
directed graph
directed acyclic graph
inheritance graph

tuple
hash table

font
screen image
point
2D graphic
3D graphic
sound bite
video image
video stream
color

hyperlink
URL
socket
button
checkbox
panel
window
scrollbar
menu

equation
procedure
buffer
error (exception)
file
directory
continuation
namespace
package
script
pointer

Challenge problem: include *all* of the data structures above in the inheritance hierarchy, noting the arbitrary design decisions (ie some forms support a choice of subcomponents).

Data Abstraction

The elementary data structures in conventional processors are *memory cells* and *addresses of memory cells*. Compound data structures can be constructed using collections of memory cells. It is almost always a poor idea to conceptualize data at the level of hardware architecture. **Data abstraction** allows us to think about data forms conceptually, using representations which map closely onto the problem being addressed and the intuitive way we think about that problem.

The dominant error in data structure design is to confuse levels of modeling, to design for hardware or software when the problem is in the real world. A similar error is to think that hardware and software data structures model a real problem. Rather than asking: what data structure models the problem?, we should ask: what data structure should I construct to implement the mathematical model of the problem?

Data structures should be designed abstractly. This means that the data structure is *conceptually independent* of the implementation strategy. This is achieved by **abstraction barriers**, functions which isolate the implementation details from the data abstraction. When the storage format, or the implementation approach, or the internal representation, or the algorithm changes, the data abstraction itself does not.

Abstract data structures often closely model mathematical data structures. A **mathematical structure** consists of

- a **representation** of the elementary unit or constants, the **base** of the structure
- **recognizer** predicates which identify the type of the structure
- a **constructor** function which builds compound structures from simple units
- an **accessor** function which gets parts of a compound structure
- a collection of **invariants,** or equations, which define the structure's behavior
- possibly, a collection of **functions** which compute properties of specific structures
- an **induction principle** which specifies how a form is constructed and decomposed

An implemented data structure should have each of the above functionalities, and no others. Most modern languages permit construction of these functionalities, however very few provide the actual tools which would make implementation easy. Accessors, constructors, and recognizers are best expressed in a pattern-matching language; invariants are best implemented in a declarative language; and induction requires special features which are usually included only in theorem provers.

Several examples of data types expressed as abstract data structures follow. The example of *Natural Numbers* illustrates a simple ADS. The example of *Trees* is more complete, including the mathematical axioms and some recursive function definitions, The example of *Strings* illustrates an **abstract domain theory**, and includes specialized functions, an induction principle, and an example of symbolic proof by induction.

Abstract Data Structure: NATURAL NUMBERS

Base	`0`
Recognizer	`numberp[n]`
Constructor	`+1[n]`
Accessor	`-1[n]`

Some invariants	`numberp[n] or not[numberp[n]]`
	`numberp[+1[n]]`
	`numberp[0]`
	`not[+1[n] = 0]`
	
	`(numberp[n] and not[n=0]) implies (+1[-1[n]] = n)`
	`numberp[n] implies (-1[+1[n]] = n)`

Induction	`if F[0] and (F[n] implies F[+1[n]]) then F[n]`

Abstract Data Structure: BINARY TREES

Predicates	`atom[x]` `tree[x]`
Constructor	`+[x,y]`
Uniqueness	`not[atom[+[x,y]]]` `if (+[x1,x2] = +[y1,y2]) then (x1=y1 and x2=y2)`
Left and Right	`left[+[x,y]] = x` `right[+[x,y]] = y`
Decomposition	`if not[atom[x]] then x = +[left[x],right[x]]`

Induction	`if F[atom] and`
	`  (if F[x1] and F[x2] then F[+[x1,x2]])`
	`then F[x]`

Some recursive *binary tree functions*

```
size[x] =def=     size[atom[x]] = 1;
                  size[+[x,y]] = size[x] + size[y] + 1

leaves[x] =def=   leaves[atom[x]] = 1;
                  leaves[+[x,y]] = leaves[x] + leaves[y]

depth[x] =def=    depth[atom[x]] = 1;
                  depth[+[x,y]] = max[depth[x],depth[y]] + 1
```

Pseudocode:

```
leaves[x] =def=   if empty[x] then 0
                    else if atom[x] then 1
                    else leaves[left[x]] + leaves[right[x]]

leaves-acc[x,res] =def=
      if empty[x] then res
        else if atom[x] then leaves-acc[(), res + 1]
        else leaves-acc[right[x], res + leaves-acc[left[x]]]
```

Abstract Domain Theory: STRINGS

Here is the **Theory of Strings** as a complete example. Note that the **Theory of Sequences** and the **Theory of Non-Embedded Lists** are almost identical.

Constants: `{E}` the Empty string

Variables (typed):

`{u,v,...}`	characters
`{x,y,z,...}`	strings

Functions: `{·, head, tail, *, rev, rev-acc, butlast, last}`

· is prefix, attach a character to the front of a string
* is concatenate, attach a string to the front of another string
[the rest are defined below as special functions]

Relations: `{isString, isChar, isEmpty, =}`

`isEmpty[x]`	test for the empty string
`isChar[x]`	test for valid character
`isString[x]`	test for valid string

Generator Facts:

```
isString[E]
isString[u]
isString[u·x]
```

Uniqueness:

```
not(u·x = E)
if (u·x = v·y) then u=v and x=y
```

Special char axiom:

```
u·E = u
E·u = u
```

Decomposition:

```
if not(x=E) then (x = u·y)
head[u·x] = u
tail[u·x] = x
if not(x=E) then (x = head[x]·tail[x])
```

Decompose equality:

```
if (u=v) then (u·x = v·x)
if (x=y) then (u·x = u·y)
```

Mapping:

```
F[u·x] = F[u]·F[x]
```

*The String **Induction** Principle:*

```
if F[E] and
      forall x:   if not[x=E],
                  then if F[tail[x]] then F[x]
then  forall x:   F[x]
```

Recursion, mapping:

```
F[E]                                        base
F[u·x] = F[u]·F[x]                          general1
F[x] = F[head[x]]·F[tail[x]]                general2
```

Pseudo-code for testing *string equality*, using the Induction and Recursion templates for binary relations

```
if =[E,E] and
   forall x,y:
      if (not[x=E] and not[y=E]),
        then if (=[head[x],head[y]] and =[tail[x],tail[y]])
               then =[x,y]
  then forall x,y:      =[x,y]

=[E,E]                                                      base
=[x,y] = =[head[x],head[y]] and =[tail[x],tail[y]]          general1

=[a,b]  =def=
      (a=E and b=E)
   or (=[head[a],head[b]] and =[tail[a],tail[b])
```

Some *axioms and theorems* for specialized functions

Concatenate, `*`, for joining strings together:

```
E*x = x,    x*E = x                         base definition

(u·x)*y  =  u·(x*y)                         recursive definition

isString[x*y]                               type

u*x = u·x                                   character special

x*(y*z) = (x*y)*z                           associativity

if x*y = E, then x=E and y=E                empty string
```

`if not(x=E) then head[x*y] = head[x]`	head
`if not(x=E) then tail[x*y] = tail[x]*y`	tail

Reverse, rev, for turning strings around:

`rev[E] = E`	base definition
`rev[u·x] = rev[x]*u`	recursive definition
`isString[rev[x]]`	type
`rev[u] = u`	character special
`rev[x*y] = rev[y]*rev[x]`	concatenation
`rev[rev[x]] = x`	double reverse
`rev[x*u] = u·rev[x]`	suffix

Reverse-accumulate, reverse the tail and prefix the head onto the accumulator:

`rev-acc[x,E] = rev[x]`	identicality
`rev-acc[E,x] = x`	base definition
`rev-acc[u·x,y] = rev-acc[x,u·y]`	recursive definition

Last and *Butlast*, for symmetrical processing of the end of a string:

`butlast[x*u] = x`	definition
`last[x*u] = u`	definition
`if not(x=E) then isString[butlast[x]]`	type
`if not(x=E) then char[last[x]]`	type
`if not(x=E) then x = butlast[x]*last[x]`	decomposition
`if not(x=E) then butlast[x] = rev[tail[rev[x]]]`	tail reverse
`if not(x=E) then last[x] = head[rev[x]]`	head reverse

Here is a function which mixes two domains, Strings and Integers:

Length, for counting the number of characters in a string

```
length[E] = 0

length[u·x] = length[x] + 1

length[x*y] = length[x] + length[y]
```

A symbolic proof by induction

To prove: `rev[rev[x]] = x` x is of type STRING

Base case: Rule applied:

`rev[rev[E]] =?= E`	1. problem
`rev[E] =?= E`	2. rev[E] = E
`E =?= E`	3. rev[E] = E, identity QED

Inductive case:

`rev[rev[x]] =?= x`	1. problem
`rev[rev[u·x]] = u·x`	2. assume by induction rule
`rev[rev[x]*u] = u·x`	3. rev[a•b] = rev[b]*a
`rev[u]*rev[rev[x]] = u·x`	4. rev[a*b] = rev[b]*rev[a]
`u*rev[rev[x]] = u·x`	5. rev[a] = a a is a char
`u·rev[rev[x]] = u·x`	6. lemma a*b=a•b a is a char
`rev[rev[x]] = x`	7. a•b = a•c iff b=c QED

Lemma:

`u*x =?= u·x`	1. problem
`(u·x)*y = u·(x*y)`	2. prefix/concatenate distribution
`(u·E)*y = u·(E*y)`	3. let x=E
`u*y = u·(E*y)`	4. a•E = a
`u*y = u·y`	5. E*a = a QED

ADS for Rational Numbers

An application program which uses rational numbers needs to perform operations on these numbers without any mention of how they are implemented. Here is a set of functions which implement rational numbers. Note that the *Constructors* and *Accessors* provide the abstraction barrier. *Recognizers*, *Functions* and *Invariants* define the structure.

REPRESENTATION

Let rational numbers be represented by the set of labels `{r,r1,r2,...}`.

A rational number is defined by the division of two integers. For any rational number **`ri`**, let the numerator be represented as **`ni`** and the denominator be **`di`**.

We will assume that the class *integer* is defined.

RECOGNIZERS

```
is-ratnum[x]  = True          iff    x is a rational number
is-denom[x]   = True          iff    x is an integer number
is-numer[x]   = True          iff    x is an integer number

is-zero[r]    = True          iff    get-numer[r] = 0
is-one[r]     = True          iff    get-numer[r] = get-denom[r]
is-illegal[r] = True          iff    get-denom[r] = 0
```

Note that `is-denom` and `is-numer` are typing predicates. They return True for any integer `x`. We arbitrarily elect to restrict numerators and denominators to positive integers, associating the sign of a rational number with the rational number itself. To express these facts:

```
is-denom[x] = True      iff
            (is-integer[x] and not[is-negative[x]] and not[x=0])

is-numer[x] = True      iff    (is-integer[x] and not[is-negative[x]])
```

Specific denominators and numerators can be recognized by the following relations:

```
is-denom-of[x,r] = True       iff    x is the denominator of rational number r
is-numer-of[x,r] = True       iff    x is the numerator of rational number r
```

Example:

```
is-ratnum[r]  =def=
  let n = get-numer[r]
      d = get-denom[r]
    if (is-denom[d] and is-numer[n])
      then return True
      else return False
```

CONSTRUCTORS

`make-ratnum[sign,numer,denom]`	returns the rational number `r = numer/denom`
`destroy-ratnum[r]`	frees memory associated with ratnum `r`

These functions transfer between the abstraction and the implementation choices. See the implementation section.

ACCESSORS

`get-numer[r]`	returns the numerator of `r`
`get-denom[r]`	returns the denominator of `r`
`get-sign[r]`	returns the sign of the ratnum `r`

These functions also transfer between abstraction and implementation.

FUNCTIONS

`equal-ratnum[r1,r2] = True`	iff `r1 = r2`
`add-ratnum[r1,r2]`	returns `r3 = r1+r2`
`sub-ratnum[r1,r2]`	returns `r3 = r1-r2`
`mult-ratnum[r1,r2]`	returns `r3 = r1*r2`
`div-ratnum[r1,r2]`	returns `r3 = r1/r2`
`reduce-ratnum[r]`	returns a reduced ratnum
`print-ratnum[r]`	returns <the graphical representation of a ratnum>
`ratnum-to-decimal[r]`	returns the decimal value of `r`

The code for these functions does not rely on the implementation of ratnums. Some pseudocode examples:

```
equal-ratnum[r1,r2]  =def=
  let n1 = get-numer[r1]
      n2 = get-numer[r2]
      d1 = get-denom[r1]
      d2 = get-denom[r2]
      res = ((n1*d2) = (n2*d1))          ;here * is integer multiply
    return res                           ;res is a Boolean equality test
```

The following code assumes that the two ratnums are positive, it would be a little more complex if signed ratnums were being added.

```
add-ratnum[r1,r2]  =def=
  let n1 = get-numer[r1]
      n2 = get-numer[r2]
      d1 = get-denom[r1]
      d2 = get-denom[r2]
      n3 = ((n1*d2) + (n2*d1))
      d3 = (d1*d2)
    return make-ratnum[n3,d3]
```

INVARIANTS

Invariants provide pre and post tests for ratnum computations. The appropriate invariants should be asserted as error checks before and after programs which use any rational numbers.

```
is-ratnum[r] or not[is-ratnum[r]] = True

is-integer[get-denom[r]]   = True
is-integer[get-numer[r]]   = True
not[equal[get-denom[r],0]] = True
<many other possible invariant equations>
```

Example of an embedded pre-test:

```
greater-than-one-ratnum[r]  =def=
  if is-ratnum[r]
    let n = get-numer[r]
        d = get-denom[r]
        comp = (n>d)
      return comp
    else return error-signal[r,"not-ratnum"]
```

AXIOMS

Axioms are another kind of invariant, defining the numerical behavior of ratnums. An infinite number of theorems can be derived from a set of axioms. Only a very few will be of utility. Finding essential and important theorems is part of the task of domain modeling.

`r1+r2 = r2+r1`	commutativity of addition
`r1+(r2+r3) = (r1+r2)+r3`	associativity of addition
`r1+0 = r1`	additive identity
`r1*r2 = r2*r1`	commutativity of multiplication
`r1*(r2*r3) = (r1*r2)*r3`	associativity of multiplication
`r1*1 = r1`	multiplicative identity
`if d=1 then r=n`	1 denominator
`if n=0 then r=0`	0 numerator
`x+(n/d) = ((x*d)+n)/d`	adding an integer
`(n1/d1)+(n2/d2) = ((n1*d2)+(n2*d1))/(d1*d2)`	adding two ratnums
`x*(n/d) = (x*n)/d`	multiplying by an integer
`(n1/d1)*(n2/d2) = (n1*n2)/(d1*d2)`	multiplying two ratnums

IMPLEMENTATION

Let's implement ratnums as 3-tuples, that is as triples consisting of a sign bit and two integers. We first elect to implement 3-tuples as a list of three integers, encoding the sign as `0=-` and `1=+`:

```
make-ratnum[s,n,d] =def=
  return List[s,n,d]

get-sign[r] =def=
  return First[r]

get-numer[r] =def=
  return Second[r]

get-denom[r] =def=
  return Third[r]
```

In the above, `get-sign[r]` is in error, since it will return 0 or 1, not the “sign” of the ratnum. An apparent fix is:

```
get-sign[r] =def=
  if First[r] = 0
    then return “+”
    else if First[r] = 1
      then return “-“
      else return Error
```

The problem now is that the return of a function name (+ or -) requires either a string encoding, thus changing the intended type of the sign data structure, or it requires language support for returning functions. Further, getting the sign of a ratnum is not intended to *apply* the unary function that the sign represents. The basic issue is that *signed integers* are very often implemented as part of a language definition; thus the programmer will not necessarily know how the operating system is treating the signs for integers. Of course, the application problem may also never require negative rationals. At a sufficiently low level of data description, abstraction is not possible due to language and operating system implementation decisions.

The make-function should include all type checking invariants. In the example below, we force an explicit identification of sign (no default permitted), and check that the numerator and denominator are positive integers.

```
make-ratnum[s,n,d] =def=
  if not[member[s,{+, -}]
    then return error-signal[s,"improper-ratnum-sign"]
    else if (not[is-integer[n]] or is-negative[n])
      then return error-signal[n,"improper-ratnum-numerator"]
      else if (not[is-integer[d]] or is-negative[d] or d=0)
        then return error-signal[d,"improper-ratnum-denominator"]
        else return List[s,n,d]
```

Should we later elect to change the implementation of ratnums, the above four functions are all that need to be changed. Say we decide to use 3 element arrays:

```
make-ratnum[s,n,d] =def=
  let a = Make-array[3] of integers
      a[0] := s
      a[1] := n
      a[2] := d
    return a

get-sign[r] =def=
  return r[0]

get-numer[r] =def=
  return r[1]

get-denom[r] =def=
  return r[2]
```

Should we later decide to include the sign of the rational number as a signed numerator, we still would change only these functions.

```
make-ratnum[n,d] =def=
  let a = Make-array[2] of integers
      a[0] := n
      a[1] := d
    return a

get-sign[r] =def=
  return sign-of[r[0]]

get-numer[r] =def=
  return r[0]

get-denom[r] =def=
  return r[1]
```

Finally, in languages which provide first-class functions, we can define the generic get-functions for an ADS. An example:

```
get-numer =def=
  function[parameters[r], body[r[0]]]
```

Challenge problem: The above model assumes that an integer can be of any precision. We know, however, that computational integers have an associated bit-length (e.g. 32bit, 64bit). Modify the above conceptualization and implementation to include specific implementation constraints such as binary precision, storage and bandwidth precision, and binary encoding.

Assignment 3: ADS for SUBSTRINGS

You will not be turning in this assignment.
Time allocation (max): thinking, 1 hour; pseudocode, 3 hours

Design an abstract data structure for the data type SUBSTRING.

1. Select a set of *tokens* to represent constants and variables in the domain.

2. Identify the *component parts* (the subtypes) of the data structure, and the functions used to recognize those parts (eg empty-substring, character, substring).

3. Identify the *decomposition axiom* which specifies how to construct and take apart SUBSTRING objects. This axiom will include the definition of *accessor functions* which access parts of a SUBSTRING.

4. Identify a *constructor function* (again defined in the decomposition axiom) which permits building compound substrings out of simple substrings. Consider the difference between a proper substring (A proper subcomponent is a component which is always smaller than its container.)

5. Identify some rules, or *invariants*, which hold between component parts of the data structure. These define the "methods" of the object type.

6. Name some *facts* which are true of this data structure. These define the type hierarchy and the other constraints on the object.

7. Identify the *induction principle* for SUBSTRINGS.

8. Using the induction principle, write pseudocode for some simple recursive functions/methods which implement the invariants of the structure.

9. Finally, suggest some computational data structures which would be appropriate for implementing the SUBSTRING ADS.

Here are some axioms you may need:

The definition of a substring:

```
x sub y  =def=  z1*x*z2 = y
```

The empty string is a substring of every string

```
E sub y
```

No string is a substring of the empty string

```
not(y sub E)
```

Prefixing a character to a string maintains the substring relation

```
if (x sub y) then (x sub u·y)
```

The following three properties of the substring relation establish that *substring is an ordering relation.*

transitivity	if `s1` is a substring of `s2`, and `s2` is a substring of `s3`, then `s1` is a substring of `s3`
antisymmetry	if two strings are substrings of each other, they are equal
reflexivity	a string is a substring of itself

Prove or define the above relations. Then prove:

- A string is a substring of itself when a character is prefixed.
- A string is a substring of the empty string when it is the empty string.
- Substring implies all the characters in the substring are in the string.
- The length of a substring is equal to or less than the length of the string.

Extend the results:

The definition of a **proper** substring:

```
x proper-sub y  =def=  not(z1=E and z2=E) and z1*x*z2 = y
```

Prove the properties of *proper* substrings (transitivity, irreflexivity, asymmetry)

Abstract Domain Theory: STRINGS

Functions

is-empty[x]	test for the empty string
is-char[x]	test for valid character
is-string[x]	test for valid string
prefix[u, x]	attach character to front of string

Decomposition

if not[x = empty-string], then x = prefix[head[x], tail[x]]

Induction

if F[empty-string] and
forall x: if not[x = empty-string], then if F[tail[x]] then F[x]
then forall x: F[x]

Recursion

base: F[empty-string]
general: F[prefix[u,x]] = prefix[F[u], F[x] }

Facts

is-string[empty-string]
is-string[u] for all characters u
is-string[prefix[u, x]]

Rules

not[prefix[u, x]] = empty-string
prefix[u, x] = prefix[v, y] iff u=v and x=y
prefix[u, empty-string] = u

if x=y then prefix[x,z] = prefix[y,z]
if x=y then prefix[u,x] = prefix[u,y]

head[prefix[u, x] = u
tail[prefix[u, x] = x

concatenate[empty-string, y] = y
concatenate[prefix[u,x], y] = prefix[u, concatenate[x,y]]
is-string[concatenate[x,y]]
concatenate[concatenate[x,y], z] = concatenate[x, concatenate[y,z]]
reverse[empty-string] = empty-string
reverse[prefix[u,x]] = concatenate[reverse[x], u]
reverse[concatenate[x,y]] = concatenate[reverse[y], reverse[x]]
reverse[reverse[x]] = x

reverse-accum[empty-string, res] = res
reverse-accum[prefix[u,x], res] = reverse-accum[x, prefix[u,res]]

length[empty-string] = 0
length[prefix[u,x]] = length[x] + 1
length[concatenate[x,y]] = length[x] + length[y]

A symbolic proof by induction

To prove: rev[rev[x]] = x x is of type STRING

Base case:	Rule applied:
rev[rev[empty-str]] =?= empty-str	1. problem
rev[empty-str] =?= empty-str	2. rev[empty-str] = empty-str
empty-str =?= empty-str	3. rev[empty-str] = empty-str
QED	4. identity

Inductive case:	
rev[rev[x]] =?= x	1. problem
rev[rev[u • x]] = u • x	2. assume by induction rule
rev[rev[x] * u] = u • x	3. rev[a•b] = rev[b]*a
rev[u] * rev[rev[x]] = u • x	4. rev[a*b] = rev[b]*rev[a]
u * rev[rev[x]] = u • x	5. rev[a] = a a is a char
u • rev[rev[x]] = u • x	6. lemma a*b=a•b a is a char
rev[rev[x]] = x	7. a•b = a•c iff b=c
QED	

Lemma:	
u * x =?= u • x	1. problem
(u • x) * y = u • (x * y)	2. prefix/concatenate distribution
(u • empty-str) * y = u • (empty-str * y)	3. let x=empty-str
u * y = u • (empty-str * y)	4. a • empty-str = a
u * y = u • y	5. empty-str * a = a
QED	

Assignment 4: ADS for a Square

Turn in the assignment at the beginning of class.
Time allocation (max): thinking, 1 hour; design, 3 hours; implementation, 5 hours

Implement an abstract data structure for the object "square".

CONCEPTUALIZATION

Define the object and the invariant relations between its parts.
Select a mathematical formalism which suits your conceptualization.
Separate internal and external transformations.
An external transformation refers to an external coordinate system or origin.

MODELING

Identify:
Domain
the component parts of the object
the appropriate recognizers, accessors, and constructors
Properties
the uniqueness relation which defines equality tests
the containment relationships between the components
the relevant relations between components at the same level
Functions
all relevant functions between components
some interesting functions between the object
and an external coordinate system

IMPLEMENTATION

Select:
a data structure for the object and its components
data structure transformations between components
implementation language or algorithm strategy for functions
Write:
a make function which builds accessors automatically
a get function for locating each part of the object
methods/functions and predicates defined above
use the Induction Principle to write the recognizer `isa-square`

EFFICIENCY

Translate your implementation into bit manipulations.

CHALLENGE ASSIGNMENT

Implement the object "cube",
or more generally still, the object "N-dimensional cube".

ADS for a Square

An **abstract data structure** is a collection of data structures, functions, and relationships which characterize a mathematical object. It is the same as an object and its methods in object-oriented programming. Abstract data structures are used to isolate the implementation of a mathematical structure from its purely mathematical definition.

The structure *square* has a collection of functions, predicates, variables, and axioms which define it as a *Square*. But it can be implemented in a dozen ways, as an array, a list, an ordered list, etc. Implementing an abstract data structure consists of defining the collection of mathematical properties and then choosing a particular implementation approach. An **abstraction barrier** is an implementation technique which allows an abstract object to be called by an application program, while keeping the implementation details completely separate.

CONCEPTUALIZATION

This phase is figuring out what the structure in question is. The exact characteristics of an abstract structure depend upon how that structure is to be used. For example, if the diagonal of a square is never used in an application, then a characteristic such as the length of the diagonal does not need to be included in the abstraction. It is often appropriate to include as many characteristics as possible in the implementation of an abstract structure, but most structures have thousands of possible characteristics. Thus, the abstraction technique makes modular extension of an abstraction easy.

In the conceptualization of a *square*, we identify what we think a square is.

- a visual object with four corner points and four sides on a plane
- the length of each side is variable
- definition: fours sides are of equal length and perpendicular
- alternative definition: interior angles are 90 degrees, and adjacent sides are equal

The idea is to identify the minimal attributes which completely define the object in question.

Note that a conceptualization often needs other concepts to make sense. We might assume, for example, that we need to actually see the square, so we identify a drawing capability to generate the picture of the square. This drawing capability is *not* part of the square. The role of the conceptualization phase is to sort out what is object and what is context. There are no right or wrong assumptions, only useful and clumsy ones.

The first attachment to this handout illustrates a type hierarchy of two-dimensional geometric figures. It shows that in an object-oriented approach, the square has multiple supertypes. Implementation of upbranching type hierarchies is particularly difficult. Designers of the language JAVA, for example, decided against including multiple inheritance. This is an example of the language restricting the type of modeling available. To include multiple inheritance in a model, a programmer would have to explicitly choose a language which includes it, since implementing multiple inheritance oneself, as a customized extension, is too difficult an undertaking.

There are always alternative ways to conceptualize the square, focussing on different sets of component objects.

Let's assume we have a way to draw a line, e.g.: a draw function,

```
DrawLine[point1,point2]
```

This function takes points as input, so that it somewhat predetermines how we think about the square itself: the square must contain points that permit it to be drawn. Drawing a square requires four lines to be drawn. Alternatively we may have a multiple point draw function:

```
DrawPolyline[point1,point2,point3,point4,point5]
```

We also need to assume a place to stand to observe the square. This is the idea of a coordinate system and an underlying space which can hold an object such as a square. E.g.: assume an integer Cartesian grid with origin at (0,0). This raises a central question: where is the origin with regard to the square?

The distinction between *coordinate-free* and *embedded-in-a-coordinate-system* is critical for abstraction. A bare square does not have a particular corner point which is its origin. An embedded square is in a space which has a special point called the origin of the space. Likewise, we can assume that the square is embedded in at least two dimensions (by definition), but a square can still exist in higher dimensions. Both **dimensionality** and **coordinate-system** are external to the square abstraction.

So *one conceptualization of a square* is:

> A compound object in 2 dimensions, consisting of four lines which intersect at four points, such that the line segments between the points are of equal length and the adjacent lines are perpendicular.

Thus we need to find a mathematical model which gives us a way to describe length in 2D and a notion of orthogonality.

Technique: It is almost always a good idea to construct a *canonical* representation, here the standard reference square. Since size is a parameter, it can be abstracted away (standardized) for the canonical square. Another way of thinking about this is that the canonical form is the private part of an object definition, while the parameter is the public part. Canonical forms are usually sorted, or arranged in order, for ease of processing. Global parameters are abstracted, and base values are set to either 1 or 0.

```
Object SQUARE consists of
      2D-points:        tl tr bl br       (tl=top-left, br=bottom-right)
      lines:            t l b r           (top, left, bottom, right)
      area:             a                 real number * real number
      size:             s                 real number

    Base:
      unit-square:      u
```

```
Predicates:
  is-a-square[p1,p2,p3,p4]
  perpendicular[l1,l2]
  parallel[l1,l2]
  on[p,l]

Functions:
  line-from-points[p1,p2]       => line
  points-from-line[l]           => p1, p2
  area-of[s]                    => square units
  hamming-distance[p1,p2]       => integer
```

The SQUARE uses other objects such as points and lines. These would have their own definitions, providing their own recognizers and accessors. For example:

```
Object 2D-POINT consists of
      locations:  x1 x2                  real number
    Base:
      origin:  (0,0)
    Predicates:
      is-a-2D-point[p]
      is-the-origin[p]
    Functions:
      quadrant-of[p]                     =>  quadrant I, II, III, or IV
```

Technique: Define what you want, not what you do not what.

Note that the question of dimensionality (is the square planar?) is completely finessed by building in only with 2D points. Similarly, things like acute angles are simply disallowed by not providing for them.

Technique: Numbers are often a base type. Mathematical structures usually assume real numbers, but computers can handle at best only rational numbers. Choose the type of number that you use for modeling very carefully, using the minimal structure to achieve the purposes of the transformation.

MODEL

Technique: Choose a model that minimizes complexity.

The standard Cartesian coordinate system provides both an orthogonality and a distance concept for the square's definition.

Elect to put the origin at one of the square's corner points. [This is not the only good choice. Putting the origin in the center of the square makes rotation around the center easier. The point is that the choice of a simple model is deeply connected with the permitted transformations of that model.]

Elect to orient the square so that one side is along each coordinate axis. The canonical square would have sides of length 1, so that using orthogonal unit vectors, the four corner points have locations

```
bl = (0,0)        br = (1,0)        tr = (1,1)        tl = (0,1).
```

I've elected to name the points also. Names are free, so some semantic reminders can be built into the names. I've used bottom-left, bottom-right, top-left, top-right.

Unit-vector model:

The Cartesian grid has orthogonality built-in as a primitive, but this does not tell you how to compute with it. From calculus, we might recall that the *unit vector* is the math structure underneath the concept of orthogonal axes in the Cartesian plane.

A 2D-vector is a set of two values specifying a distance and a direction. In polar coordinates, a point at the end of the vector is represented by (distance, angle). Alternatively, the location of a point can be expressed in Cartesian coordinates, as the composition of two orthogonal basis vectors (i, j), which are perpendicular by construction.

```
x  =  (1i,0j)
y  =  (0i,1j)
```

The definition of perpendicular is:

```
i·j = 0            (dot-product)
i·i = j·j = 1
```

Vectors are convenient to process using matrix algebra. Unit vectors provide a math model which makes conceptualization easy. Now we switch math models to make computation easy. The square is now a matrix (rows are corner points, columns are unit vectors):

```
                [0 0]                                          [0 0]
unit-square = [1 0]          parameterized-square = size · [1 0]
                [1 1]                                          [1 1]
                [0 1]                                          [0 1]
```

If we want to move the square (without rotation), for example so that point (0,0) is now at point (a,b), then it suffices to add the new location to the old for each point before scaling:

```
[0 0]                    [ a   b ]
[1 0] + [a b]    =       [a+1  b ]
[1 1]                    [a+1 b+1]
[0 1]                    [ a  b+1]
```

Vector calculus is not necessary here, since it is just one model of how a Cartesian plane works. We could just as well use high school geometry or algebra.

Size:

The canonical square has sides of unit 1. The size parameter can be factored out of the canonical form as a multiplier of the entire unit square:

```
size · {(0,0),(1,0),(1,1),(0,1)}
```

Thus, the square is the set of points:

```
{(0,0),(size,0),(size,size),(0,size)}
```

We say that a function is *listable* when it can be applied to or abstracted from the elements of a nested list. The example here is multiplying a nested list which represents the square by a scalar which represents the size parameter.

IMPLEMENTATION

Now we must implement the functions and predicates in the conceptualization. If the math model is selected carefully, it will provide the operators for the implementation (but it will not provide an efficient selection of data structure).

Example: `perpendicular[l1,l2]`

Our lines are defined by two points each. One method would be to use the formula for the slope of lines (in terms of two points defining the line) and the relationship that perpendicular lines have slopes of the form

```
m1 * m2 = -1
```

This is too complicated since it tests lines in any orientation, for any angle of intersection. Since all lines in the square are either perpendicular or parallel, we could just test for parallel lines, with

```
perpendicular[l1,l2] = not[parallel[l1,l2]]
```

Parallel lines are easier to determine. Two sides of a square are parallel when their slopes are equal, `m1 = m2`.

However, the easiest approach is recall that we have only squares in our model. So all lines are sides of a square. Sides are perpendicular if the have a corner point in common. Sides are parallel if they share no corner points.
In pseudo-code:

```
perpendicular[(p1,p2),(p3,p4)] =def=
      p1=p3 or p1=p4 or p2=p3 or p2=p4
```

If the points in the line data structure are always ordered, then this test becomes even easier:

```
      p1=p4 or p2=p3
```

TRANSFORMATION:

The selection of a model and an implementation depends greatly on how an object is used. Let's assume that we want to do three things with the square:

```
Translate[sq,distance]
Rotate[sq,angle]
Scale[sq,size]
```

If the representation of the square is the unit-square, with a parameterized scaling factor (i.e. size)

```
sq = size · {(0,0),(1,0),(1,1),(0,1)}
```

then these transformations are easy to model:

```
Translate[sq,d]  = d + sq
Scale[sq,s]      = s * sq
Rotate[sq,theta] =
  origin-at-center = Translate[sq,(-1/2,-1/2)];
  radius = SquareRoot[x^2 + y^2];          ;any point is ok by symmetry
  for all points in origin-at-center:
    new-x = radius * cos[theta]
    new-y = radius * sin[theta]
```

Rotation becomes complex because

1) we adopted Cartesian coordinates, when polar coordinates make rotation simpler, and
2) we can rotate around any arbitrary point.

The default above is to rotate around the origin point, but the intuitive default would be to rotate around the center of the square. Thus the origin needs to be translated before the rotation.

TRANSFORMATION DEFINING NEW PROPERTIES

Rotation of the square lets us observe some of its symmetry properties. For example, rotation by 90 degrees generates a unit-square with the same set of corner points, in a different order. But since the points are a Set, order doesn't matter. That is, rotation by 90 degrees is equivalent just to renaming the corners. Since the choice of names is free, rotation by 90 degrees is the same as no change at all to the square. An implementation could efficiently use:

```
Rotate[sq,90] = sq
```

This observation is the beginning of exploring the group structure of the square. We have moved from graphics to algebra to vectors to group theory, different mathematical models of the conceptualization of a square.

CODE DECISIONS

Without concern for optimization, we can see that almost any transformation of the square involves adding or multiplying the set of points which define the particular square. After the pseudo-code for all functions and predicates has been constructed, we will have a very good idea of the types and frequencies of mathematical operations (just count the number of times a data structure is accessed, multiplied, etc.). The implementation data structure should make these frequent operations computationally easy. But also fundamental to an abstract data type, the interface to the implementation must be mathematical, not computational. That is, the mathematical functionality must be partitioned from the implementation choice of data structure.

I'd put each square into an array consisting of the name and the eight rational numbers defining corner points:

```
make-square[name,sq] =
  sq-name := make-array[9];
  sq-name[0] := name;
  sq-name[1] := blx-of-sq;                ;blx is bottom-left-x value
  ...
  sq-name[8] := tly-of-sq;
```

An example function:

```
Translate[sq,(dx,dy)] =
  for i= 1 to 8, do
    when odd[i]  sq-name[i] := sq-name[i] + dx;      ;x values in array
    when even[i] sq-name[i] := sq-name[i] + dy;      ;y values in array
```

The next level of implementation optimization is to examine the encoding of the data structure to determine if it is efficient for the operations performed on it. For example, above Translation is better expressed in Cartesian coordinates, while Rotation is better expressed in polar coordinates. The programmer needs to know what the usage of the data structure will be. If, for instance, 95% of the operations on the square will be rotations, then the canonical square is best formulated in polar coordinates.

Another example from above is the choice of the Perpendicular algorithm. The encoding can encourage calculation of line slopes by providing them in ready form, or it can encourage comparison of points by providing the points directly.

AN OPTIMIZED EXAMPLE

The attached three pages contain a description of the development of a minimal data structure for a cube (the principles apply to a "cube" of any dimension, including a 2D square). The mathematical model is *unit vectors*. All properties are defined as operations on unit vectors, using only multiplication in the ternary domain `{0,_,1}`. Special multiplication operators are defined for this purpose. The ADS keeps a tight association with the visual properties of a cube.

Abstract Data Structure: SETS

Sets (unordered collections of unique objects) are a fundamental mathematical concept. Pure set operations are impossible to implement on a serial processor. Note that the model of sets is isomorphic with the model of *propositional calculus*, with the membership operator added.

A **set implementation** with the functions Insert, Delete, and Member is called a *dictionary*.

Mathematical model:

```
S = {x| <statement about x>}   extensional, defined by common property
S = {a,b,c,...}                intensional, defined by naming the members
```

empty set:	`not (x in S)  forall x`
membership:	`x in S  =def=  x=s1 or x=s2 or x=s3 or ...`
subset:	`if (x in S1) then (x in S2)`
union:	`(x in S1) or (x in S2)`
intersection:	`(x in S1) and (x in S2)`
difference:	`(x in S1) and not(x in S2)`

recursive set membership:

```
x in S  =def=
      not[x=empty-set] and (x = get-one[S] or (x in rest[S]))
```

Implementation functions:

```
Make-empty-set
Make-set[elements]
Insert[element,set]
Delete[element,set]
Equal[set1,set2]
```

`Cardinality[set]` = count of members

Characteristic function F:

```
(F[x] = 1 iff x in S)  and  (F[x] = 0 iff not(x in S))
```

Implementation using Enumeration: the named members of the set can be stored as any of the following: array, list, linked list, queue, stack, bit-array, hash table, balanced tree, binary search tree, etc.

Implementation using Predicates: the common property of set members can be implemented as a characteristic function, accessor function, or regular function.

Algebraic Specification of Sets

This algebraic specification is also a functional implementation (i.e. code) in a programming language designed for formal verification. The language (called ASL for Algebraic Specification Language) is programmed by specifying the abstract domain theory for a datatype. All implementation decisions are made by the *compiler*. The engine which operationalizes the code is very much like Prolog, that is, it is a logic pattern-matching engine with patterns compiled from the input specifications.

New functions and capabilities are added by *extending* a theory, that is, by building another theory which uses the base theory definitions and axioms. Application code consists of mathematical formulas to be evaluated.

```
theory TRIVIAL is

      sorts Elt

endtheory TRIVIAL


module BASICSET [ELT :: TRIVIAL] is

      sorts       Set

      functions
            Phi, Universe :         Set

            {_}:                    Elt -> Set

            _ symmetric-diff _ :    Set, Set -> Set
                                    (assoc comm ident: 0)
            _ intersect _ :         Set, Set -> Set
                                    (assoc comm idem ident: Universe)

      variables
            S,S',S'':               Set
            Elt,Elt':               Elt

      axioms
            (S sym-diff S) = Phi

            {Elt} intersect {Elt'} = Phi  :-  not(Elt = Elt')

            S intersect Phi = Phi

            S intersect (S' sym-diff S'')
                  = (S intersect S') sym-diff (S intersect S'')

endmodule BASICSET
```

```
module SET [X :: TRIVIAL] using NAT, BASICSET[X] is

      functions
            _ union _ :             Set, Set -> Set

            _ - _ :                 Set, Set -> Set

            #_ :                    Set -> Nat

      predicates
            _ member _ :            Elt, Set

            _ subset _ :            Set, Set

            empty :                 Set

            _ not-member _ :        Elt, Set

      variables
            X:                      Elt
            S,S',S'':               Set

      axioms
            S union S' = ((S intersect S') sym-diff S) sym-diff S'

            S - S' = S intersect (S sym-diff S')

            empty S :- S = Phi

            X member S :- {X} union S = S

            X not-member S :- {X} intersect S = Phi

            S subset S' :- S union S' = S'

            # Phi = 0

            #({X} sym-diff S) = #(S) - 1 :- X member S

            #({X} sym-diff S) = #(S) + 1 :- X not-member S


endmodule SET
```

Data Structures for Sets

The appropriate data structure to implement a mathematical object such as sets depends upon *resources*, *task,* and *context*. Things that effect the performance of set data structures include:

- size of elements
- complexity of elements (i.e. nested and hierarchical forms)
- size of the Universe of Discourse
- typical size of sets during an application
- maximum size of sets during an application
- frequency of use of each type of transformation
- available resources (processing power and memory size)

The choice of a data structures depends upon an *analysis of the task* it is to be used for. To implement a common mathematical object (such as sets, integers, strings, trees and logic), we build a model of the task which tells us the frequency of transforms and the types of elementary objects. We then match these characteristics against the known performance of the several implementation choices. When the task is unknown or likely to change (almost always the case), *defensive programming* can lower maintenance and modification costs through the use of more generic data structures. Naturally, data types that are built into a language have had their implementation decisions made independent of the task.

Implementing Sets

We need a representation which may require adding and deleting members, identifying membership, and performing basic set operations such as union and intersection.

doubly-linked lists
good for small sets
O(NlnN) operations

balanced trees
good for sorted lists with lots of look-up and for testing membership
O(NlnN) operations, membership in O(lnN)

hash tables
must be invertable for set enumeration
converts set to range of integers, easy for lookup
good for adding and deleting members

bit vectors
union and intersection are bit level
poor for enumerating elements
good for transforming sparse sets, bad for very large sets
O(n) operations

Hybrid representations, such as a linked segment list, can be constructed which optimize particular performance characteristics.

Algorithm Complexity

Classification of Worst Case Algorithm Complexity

Complexity refers to the growth in resources needed by an algorithm when the size of the input increases. *Resources* may be computing time or storage space or both.

Big-Oh notation, **O(f[n])**, identifies an asymptotic upper bound to the computational complexity of an algorithm.

O(1)	instructions executed once (constant time)
O(logN)	as input size N increases, running time increases by logN (logarithmic)
O(N)	instructions executed N times, proportional to size of input (linear)
O(N logN)	time proportional to input size N times logN (slightly more than linear)
O(N^2)	time proportional to size squared (quadratic)
O(N^3)	time proportional to cube of input size (cubic)
O(2^N)	exponential time, non-polynomial (NP)

In the above the function **f[n]** is represented by the argument to the function **O**.

Caution: Algorithm complexity is for the *worst case*; it does not indicate average performance. Very often the worst case is extremely rare for practical problems. Also initialization and non-loop running times may be large regardless of big-oh time.

Little-Oh notation, **o(f[n])**, identifies complexity that is strictly less than **f[n]**. Big-Oh allows the possibility that complexity is exactly equal to f[n], where Little-Oh does not. Little-Oh is rarely used.

Big-Oh Notation as an Approximation Technique

A polynomial has the form:

```
Ax^n + Bx^(n-1) + ... + Yx + Z.
```

For example, a third degree polynomial has the form

```
Ax^3 + Bx^2 + Cx + D
```

Big-oh notation is intended to tell you approximately how much effort an algorithm might require, given the worst case. Because the terms to the right contribute much less than accompanying terms on the left, big-oh notation drops all terms on the right. Further, because the multiplication factor in front of each term contributes a constant amount, it too is dropped. Thus, the complexity of a general polynomial is

```
O(x^n)
```

This approximation is used for three reasons:

1. We are usually concerned about only a rough approximation of how much effort an algorithm takes.
2. We usually change an algorithm only when the choice is significantly much better.
3. Our analysis tools aren't strong enough to tell us much more anyway.

Caution: Depending on the input size, the deleted coefficient, **A**, may have more of an influence than the determining function **f[n]**.

Classification of Lower Bounds of Complexity

Big-Omega notation, **Omega(f[n])**, identifies a lower bound to worst case complexity. Knowing that all inputs will take a minimum amount of time is useful, but is usually too difficult to determine. As well, we do not know how to design silicon circuits to take advantage of lower bounds.

Big-Theta notation, **Theta(f[n])**, means that complexity increases exactly the same as **f[n]**. This is fairly rare, but useful for analysis of families of algorithms.

Complexity Reduction

When a problem can be subdivided into parallel processes, or subdivided into multiple recursive processes, the complexity of the algorithm decreases, most usually from a function of **x^n** to a function of **x^(n-1) * log n.**

When input is sorted, as opposed to random, complexity usually decreases from a function of **x^n** to a function of **x^(n-1)**. However, the sorting process takes at best O(N log N), so this gain is meaningful only when the algorithm using the input is more complex than O(N log N).

Algorithm analysis yields theoretical results, which are meaningful only when the implementation and compilation are maximally efficient. However, algorithm analysis does produce results which cannot be improved upon by any means (implementation can only make the situation worse).

Satisfiability

The Satisfiability Problem for propositional calculus (i.e. Boolean algebra, combinational circuitry) asks: Is there at least one binding for the variables in the statement of the problem which makes the entire problem True?

Satisfiability is the basis for almost all algorithm analysis which distinguishes between polynomial (i.e. tractable) and non-polynomial (i.e. intractable) problems. The complexity of the algorithm which answers the satisfiability problem is currently not known, even though it is about the most simple non-trivial mathematical problem possible. If the best case complexity of that algorithm is non-polynomial (i.e. exponential), which almost everyone believes is the case, then most interesting algorithms in Computer Science are intractable.

Complexity Workshop

Here is a small example of how to think about and work with algorithm complexity.

Data Structure Efficiencies

The elementary unit of analysis is "***memory accesses***", which may include storing an item (constructors), locating an item (recognizers), and retrieving an item (accessors).

Arrays:	`A[i]`	given index `i`, go straight to `A[i]`	`O[1]`
Trees:	`((a b)(c d))`	a series of branching decisions locates item `i`	`O[ln n]`
Lists:	`(a b c)`	look through all items for `i`	`O[n]`

The Memory Hierarchy (1999 technology)

Type	*typical access time (^10 ns)*	*typical capacity (bytes) (^2 bits)*	
registers	2-5 ns	0	64-512 9-12
primary cache	4-10 ns	1	8K-256K 16-21
secondary cache	20-100 ns	2	512K-4M 22-25
main memory	50-1000 ns	3	8M-4G 26-35
disk	5-15 ms	7	500M-1T 32-41
tape	1-50 s	10	unlimited

1 ns = 10^-9 s 1 ms = 10^-3 s

Pragmatics of Nested Loops

Nested loops effectively do a brute-force search over all items. Consider searching a three dimensional matrix, indexed by `(i,j,k)`:

```
for each i do
  for each j do
    for each k do
      <process>
```

If every item must be processed (e.g.: a pixel-based graphics display process), then the loops are unavoidable, and the best case is also the worst case.

However, if one item must be found out of n, then we can avoid some effort. The two effort avoidance techniques are

data structure organization and ***smart, knowledgeable search***

When is Organization better than Knowledge?

Search is an exponential process. Ignoring effort other than direct search, and assuming the data can be structured as a binary decision process, then

`Search-effort = 2^n` `n` is number of items in search pool

How deeply can loops be nested to have the same efficiency as search? Ignoring polynomial coefficients:

`Loop-effort = n^k` `k` is the number of nested loops

Point of equal effort:

```
2^n = n^k
```

Solving for `k`:

```
n/ln n = k
```

Table of `n/ln n`:

ln n	*n*	*n/ln n*
1	2	1
2	4	2
3	8	2.7
4	16	4
5	32	6.4
6	64	10
7	128	18
8	256	32
9	512	57
10	1024	102
11	2048	186
12	4096	341

Conclusion: worst case search through even small sets is worse than many nested loops.

Pragmatics

In fact, worst case search requires every decision to be incorrect. This is very difficult to achieve, since it requires perfect anti-knowledge.

Let `P` be the fraction of times that a incorrect decision is made. For example, in a sorted binary tree, the correct decision is made every time, since the structure of sorting gives the needed contextual information. In the perfectly sorted case, with no search errors:

`n` decisions requires `ln n` steps, `2^n` decisions requires `n` steps.

Conclusion: Sorting turns search into looping

In general, making P correct decisions:

```
2^(nP) = n^k

Pn/ln n = k
```

Here is how knowledge effects search effort:

ln n	*n*	*all wrong P=1 n/ln n*	*half/half P=2^-1*	*1 in 10 P=2^-3*	*1 in 100 wrong P=2^-7*
1	2	1			
2	4	2	1		
3	8	2.7	1.4		
4	16	4	2		
5	32	6.4	3.2		
6	64	10	5	1.2	
7	128	18	9	2	
8	256	32	16	4	
9	512	57	29	7	
10	1024	102	51	13	
11	2048	186	93	23	1.5
12	4096	341	170	42	2.7

When we guess correctly half of the time, the cross-over point between search and brute-force looping is at about 30 items for k=3. As errors decrease to 1 in 100, the number of items increases to over 4000. Thus, partial knowledge about the location of an item greatly increases the number of items that can be searched before a search strategy becomes less desirable than brute-force looping.

What is an Algorithm? Versions of Factorial

What is an algorithm? The text offers this: "An *algorithm* is a clearly specified set of instructions the computer will follow to solve a problem." This is essentially the same definition as given by one of the first books on algorithms: Aho, Hopcroft and Ullman, *Data Structures and Algorithms* (1983). The essential ingredients of an algorithm:

1. A sequence of steps
2. An unambiguous specification for each step
3. The steps can be carried out mechanically, ie by a machine.
4. Finite and terminating.

The above definition is bias toward a procedural model of computation. As well, it is blind to compiler and machine code transformations. What, for instance, does "clearly" mean? An algorithm fits the illusion that the user has about the computational process? An algorithm has a unique sequence of machine instructions? An algorithm is a mathematical construct divorced from implementation details?

Why a sequence of steps? Are parallel, distributed, and concurrent algorithms not really algorithms? Recall that *all* silicon processes occur in parallel.

Why unambiguous? A proof, for example, can take many different forms, using completely different steps and theories, but still arriving at the same result. Are non-deterministic, learning, chaotic, quantum, and other modern computational techniques not using algorithms?

Why mechanical? This one is easy: because Computer Science studies machines.

Why finite? When you write a control loop that waits for input, it is an algorithm?

Are algorithms different than data structures? Consider what some founders of Computer Science (Wirth, Djkstra, Hoare) said thirty years ago: "Decisions about structuring data cannot be made without knowledge of the algorithms applied to the data and that, vice versa, the structure and choice of algorithms often strongly depend on the structure of the underlying data. In short, the subjects of program composition and data structures are inseperably intertwined." (in Wirth, Algorithms + Data Structures = Programs (1976), pxii)

Why then do object-oriented, logical, functional, and mathematical text books not mention algorithms at all? How does *algorithm analysis* work if it does not include notions of data structure or underlying engine or application context?

Algorithm analysis assumes that algorithms are mathematical procedures which have clearly definable upper limits on resource usage. The upper limits on algorithmic complexity provide little useful information about processes; algorithm analysis primarily tells us whether or not an encoding is tractable or intractable.

Tractable algorithms do not require exponentially increasing resources as problem size increases. Finding out whether or not an arbitrary Boolean expression has any solutions (the *satisifability* problem) is thought to be an intractable problem. And this is the simplest of all non-trivial mathematical questions in the simplest of all non-trivial mathematical domains. Recall that Boolean computation is all that occurs at the silicon level of computation. Why can algorithm analysis not even tell us if satisfiability is tractable or intractable?

Examine each of these algorithms for computing the Factorial function. Which is best? Which is most efficient (this is a trick question, why?)? Which are algorithms?

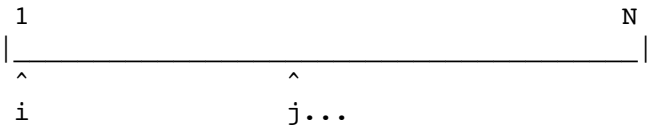
```
proceduralFactorial[n] :=
      if ( Integer[n] and Positive[n] )
            then
                  Block[ {iterator = n,
                          result = 1  },
                    While[ iterator != 1,
                        result := result * iterator;
                        iterator := iterator - 1 ];
                    return result]
            else  Error


sugaredProceduralFactorial[n] :=
      Block[  {result = 1},
          Do[  result = result * i, {i, 1, n} ];
          result]


loopFactorial[n] :=
      { For[ i=1 to n, i++, result := i*result ];
        result }


assignmentFactorial[n] :=
      { product := 1;
        counter := 1;
        return assignmentFactorialCall[n, product, counter] }


assignmentFactorialCall[n, product, counter] :=
      if[ (counter > n)
            then
                  return product
            else
                  { product := (counter * product);       /error if these are
                    counter := (counter + 1);             /in reverse order
                    return assignmentFactorialCall[n, product, counter] } ]


guardedFactorial[n, result] :=
      Precondition:     Integer[n] and Positive[n]        /also end condition
      Invariant:        factorial[n] = n * factorial[n - 1]
      Body:             guardedFactorial[ (n - 1), (n * result) ]
      PostCondition:    result = Integer[result] and Positive[result]
                                 and (result >= n)


recursiveFactorial[n] :=
      if[  n == 1, 1,  n*recursiveFactorial[n - 1] ]
```

```
rulebasedFactorial[1] = 1;
rulebasedFactorial[n] := n * rulebasedFactorial[n - 1]


accumulatingFactorial[n, result] :=
      if[ (n = 0)
            then
                  return result
            else
                  return accumulatingFactorial[ (n - 1), (n * result) ]


upwardAccumulatingFactorial[product counter max] :=
      if[ (counter > max)
            then
                  return product
            else
                  return upwardAccumulatingFactorial[ (counter * product)
                                                      (counter + 1)
                                                      max ] ]


mathematicalFactorial[n] =
      Apply[ Times, Range[n] ]


generatorFactorial[n]
      Times[ i, Generator[i, 1, n] ]


combinatorFactorial :=
      Y f< n< COND (=0 n) 1 (* n (f (-1 n))) >>


sugaredCombinatorFactorial =
      S (CP COND =0 1) (S * (B FAC -1)))


integralFactorial[n] =  Gamma[ n + 1 ]  :=
      integral[ 0 to Infinity, (t^n * e^(1 - n)), dt ]


streamOfFactorials  =
    streamAttach[ 1 streamTimes[streamOfFactorials streamOfPositiveIntegers] ]
streamOfPositiveIntegers =
    streamAttach[ 1 streamBuild[ Add1 CurrentStreamValue ] ]


From Abelson and Sussman, Structure and Interpretation of Computer Porgrams

abstractMachineFactorial =  <p385>

registerMachineFactorial =  <p511>

compiledFactorial = <p596-7>
```

Analysis of Sorting Algorithms

Selection Sort

Find the nth smallest element and exchange it with the element in the nth position; recur.

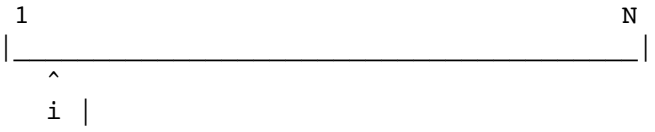


Find the smallest element and put it in place (first) by exchanging it with the first element. Then find the second smallest of those remaining and put it second, etc.

Look at every `i`. For every `i`, look at every `j`. N^2/2 comparisons and N exchanges. O(N^2). Each item is moved only once, so selection sort is good when the item records are very large relative to the sorting key.

Insertion Sort

Take the next available element and put it in its sorted place among those already sorted; recur.

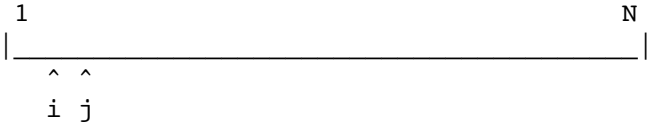


Look at each element in turn and insert it in order on the left. To make room, elements on the right may have to be shifted right.

For every `i`, compare it to every `j` (worst case). N^2/4 comparisons and N^2/8 exchanges on average. Linear for almost sorted lists.

Bubble Sort

Take the next element and pairwise sort it with the previous element; recur on the elements of the list. Then recur on the list until no pairwise changes.

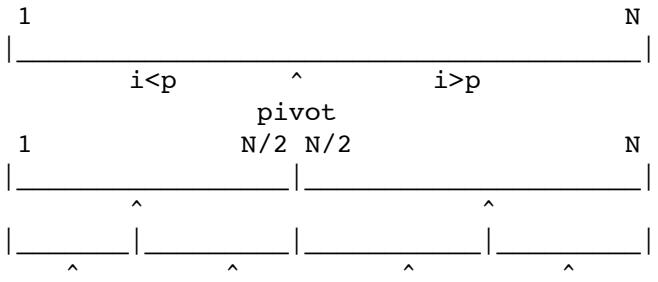


Look at each pair of adjacent elements and swap them if they are not in order. Pass through the entire list until no exchanges.

Look at every pair and sort the two if necessary. Repeat. N^2/2 comparisons and N^2/2 exchanges worst case and average case. Always slower than insertion sort.

Quicksort

Partition the list into two subsections. Sort each recursively.



Select a pivot. Exchange each `i` on the left of the pivot that is greater than the pivot with a `j` on the right of the pivot which is less than the pivot. Recur. 2NlogN comparisons on average

Mergesort

Divide the list in half, sort each half recursively, then merge the results. Merging is combining two sorted files into one sorted file. Merging two sorted files takes O(N) effort. Since the quicksort "front-end" to mergesort takes O(NlogN) steps, the cost of merging is relatively insignificant.

Hashing

The technique is to make arithmetic transformations on the search key, which then corresponds to table addresses. A **hash function** computes the key transform (examples: mod M, sum of digits). The **hash key** is then an index into a table which can be accessed in O(1) time.

Hashing is a time-space tradeoff, providing faster access by using more memory space. The storage part of a hashing algorithm is a type of bin sorting (i.e. sorting objects into bins by some criteria). The retrieval part of hashing is a type of searching (i.e. finding the right bin and then searching for the desired object within that bin).

When the hash function computes the same key for two records, the table entry is not unique. Further search is then required within the table index. This is called **collision detection**.

Radix search is similar to hashing, but the key itself is used rather than an arithmetic function of the key.

Strings and Patterns

String Processing Design Issues

Size of alphabet: the alphabet is the set of characters in the string language
Boolean alphabet = {0,1}
simple English alphabet = {a,b,c,...,x,y z}
Larger alphabets require more effort for matching

Redundancy: strings with high character redundancy require more comparisons
e.g.: "abbaabbacdcdcd" is more redundant than "thisstringistoolong"

Processor/data-structure: Sometimes it is easier not to decrement the pointer index (i.e. not to back up). Some languages do not support intertwined functional recursion.

Brute-force-match

```
pattern[0..3] =   "bcde"
string[0..14] =   "abcdeabcdeabcde"
```

Compare the first character in the pattern to the first character in the string. If it matches, compare the rest of the pattern to the string, one character at a time. When it doesn't match, compare the next character in the string to the beginning of the pattern. Repeat until the string is exhausted.

```
pattern[0..M]
string[0..N]
brute-match[string]  =
  i := 0;  j := 0;
  loop until j=M or i=N
    if pattern[j] = string[i]
      then  i++; j++                    */increment if matching
      else  i := i-j+1; j := 0          */reset if not matching
  if j=M                                */pattern match found
    then return (i-M)                   */location of start of match
    else return i                       */end of string
```

N*M comparisons worst-case, but average case is almost always N+M

KMP-match and Boyer-Moore-match

KMP-match is the same as brute-force-match, except when there is a mismatch, it backs up the string pointer as little as possible by using the knowledge that the examined string characters before the mismatch do not match the pattern.

Boyer-Moore-match is the same as KMP-match, except that it makes the comparisons from right to left, thus assuring that the maximum number of mismatching characters can be skipped. If the front of the pattern does not match, then you can only skip ahead one character in

the string, but if the end of the pattern does not match, then the entire length of the pattern can be skipped. Boyer-Moore requires reverse traversal of an array, which is expensive in some implementations.

Pattern-matching Languages

Expand the concept of a pattern to include different types of matches. These three define a *regular language*:

Concatenation: characters must be adjacent (standard)

Or: a character location in the pattern may have more than one acceptable match.
`A(B+C)D` matches both `ABD` and `ACD`

Closure: a pattern may be repeated any number of times (including zero)
`A* = AAA... (AB)* = ABABAB...`

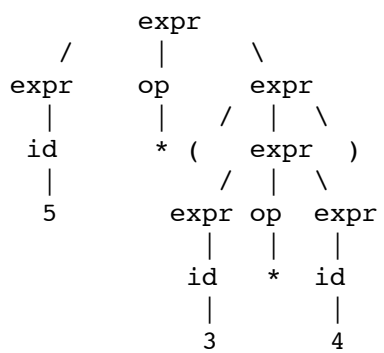
Parsing a context-free grammar

Parsers convert a string input into a tree, with tree-leaves forming the words/characters and internal nodes describing the type of expression.

Example, simple arithmetic expressions:

```
expr  -->   expr  op  expr
expr  -->  ( expr )
expr  -->  - expr
expr  -->  id
op  -->  +
op  -->  -
op  -->  *
op  -->  /
op  -->  ^
```

Parse 5 * (3 + 4):



Example, Backus-Naur form for a regular language:

```
<expression>  ::=  <term>  |  <term>  <expression>
<term>  ::=  <factor>  |  <factor><term>
<factor>  ::=  (<expression>)  |  v  |  (<expression>)*  |  v*
```

Top-down Parsing

The input expression (in the above example, `5*(3+4)`) is processed one character at a time. The parser calls each production rule recursively until either the entire expression is accepted or a syntax error is identified.

Example of a parser/recognizer:

```
input[0..6]          */in the example, N=7 for the seven characters in "5*(3+4)"
i = 0
expression =
      case input[i]
            "("         i++;
                              expression;
                              if input[i] = ")" then i++ else ERROR
            "-"         operator;
                        expression
            id          i++           */if id has more than one character
                                      */then need to process id length here
            otherwise   expression;
                        operator;
                        expression

operator =
      if  member[ input[i], { "+",  "-",  "*", "/", "^" } ]
            then i++
            else ERROR
```

Pattern Variables

Some programming languages allow patterns as variables. Identifiers within the pattern control decomposition and construction.

Record:

```
((first-name last-name) (address-digits street-name city state) (phone))
```

Example:

```
Get-component [((_ _) (_ _ city_ _) (_)),my-record]
      returns city = <my-city>

Change-city [record-pattern, new-city]
      returns ((_ _) (_ _ new-city _) (_))
```

Search Algorithms

Search is an approach to finding desired data in the presence of a collection of other data. The question is which subgroups of data to examine first. The purpose of sorting and other data ordering techniques such as array indexing, sorted lists, binary search trees, hash tables, and priority queues, is to make search more efficient.

Data Structures

dictionary:	a collection of searchable records
record:	information associated with a search key
key:	data field of search

Types of Search

sequential: look through the data array until the search object is found
average cost = N/2 comparisons
can use any of the sorting algorithms to improve this search algorithm

binary: divide and conquer, analogous to quick-sort, assumes ordered keys
maximum cost = lnN+1 comparisons
variant is **interpolation** search, guess location of target as entry rather than middle of dictionary.

binary search tree: build a balanced tree so that comparisons are always lnN in efficiency
maintenance of the balanced tree structure is expensive.
This technique trades search efficiency for data sorting efficiency.
worst case is when files are in order (or reverse order)

Techniques for balancing tree search

2-3-4 trees: have multiple keys associated with each node which separate children into organized groups
worst case = lnN+1 nodes

red-black trees: use an extra bit to encode 2-3-4 nodes into binary nodes
extra key is “red” or “black”, black meaning “smaller than” or “greater than” the node key and red meaning “split 3 nodes into 2”.

Generic Search

When searching an organized data structure such as a list or a tree, there are a variety of strategic approaches, falling into three broad categories:
brute-force, *iterative*, and *heuristic*.

Brute force techniques	depth-first breadth-first
Iterative techniques	iterative deepening iterative broadening
Heuristic techniques	hill-climbing beam search best-first adversarial search

For *depth-first* search, the tree is descended until a leaf node is reached, then the search backs up to the nearest node with has not been explored. The trouble with depth-first search is that some descents may go on for a long time, while most of the rest of tree remains unexplored. One way to manage the exploration is to collect a list of children nodes to be explored, putting the newest ones on the top of a priority queue.

For *breadth-first* search, the tree is descended one level at a time. All nodes at each level are explored before going on to the next deeper layer. The trouble with breadth-first search is that the desired leaf nodes may be the last to be explored. One way to manage breadth-first search is to collect a list of children nodes to be explored, putting the newest ones (for the next level) on the bottom of a priority queue.

Iterative techniques are designed to avoid the problems with pure depth and breadth-first searches. The idea is to elect to go to a particular depth, and then change from depth-first to breadth-first. Similarly, a search could elect to completely explore only those nodes with a limited fanout, or breadth, before converting to depth-first. One way to manage iterative approaches is to collect a list of children nodes to be explored, putting the desired nodes on the top of the priority queue, and the postponed nodes on the bottom of the queue.

Heuristic approaches depend upon an evaluation function computed for each node. Nodes to be explored are placed on a priority queue in order of their value. For *hill-climbing*, the evaluation function estimates the remaining distance to the bottom of the tree, in an attempt to improve depth-first search. Analogously, *beam search* selects the best nodes at the same level, attempting to limit breadth-first search. *Best-first* selects the node with the best evaluation, regardless of depth or breadth.

Finally, *adversarial search* applies to situations in which search turns are taken by two competing opponents (like in checkers and chess). The evaluation function estimates the best strategy, taking into account opposing choices as well as positive moves. One way to think of adversarial search is that a tree is being searched, but every other move is directed by an anti-search, which attempts to make the search fail.

Graph Search

Techniques for dealing with graph data structures generalize those for tree data structures, since trees are a special case of graphs.

Basic Algorithms for Intractable Problems

Recursive Decomposition (Divide and Conquer)

Divide a problem into smaller problems of the exact same type, solving each recursively. Quicksort is a primary example.

Branch-and-Bound

Use this when a collection of decision variables must be examined. Visualize the problem as a tree in which each node represents a decision point, and the edges coming from the node represent the various choices being made. A *branch* is a choice point for one decision variable. At each branch, the *lower bound* of the solution is computed. If that bound is higher than other known solutions, then the branch can be pruned, or not visited.

The strength of this algorithm is in the ability to prune large portions of the decision tree. This depends on the quality of both the *selection function* for which branch to consider next, and the *bounding function* for evaluating that branch. The algorithm is efficient for average cases when the selection function makes an early choice of a branch which is close to optimal, and when the bounding function is sharp (providing exact bounds) and fast.

```
Branch-and-bound =def=
  current-best := anything;
  current-cost := infinite;
  decisions := s0;                    /* s0 is all possible unmade decisions
  while decisions =/= ( ) do
    select a decision  s;
    remove s from decisions;
    make a branch based on s, giving sequences s[i] from 1 to m;
    for i = 1 to m                    /* m=2 is a binary decision tree
      compute the lower bound b[i] of s[i];
      if b[i] >= current-cost
        then kill s[i];
        else if s[i] is a complete solution
          then current-best := s[i];
             current-cost := cost of s[i];
          else add s[i] to decisions.
```

Example: Selecting the shortest path from your house to the store. Each corner provides a decision point.

Dynamic Programming

Use this when the problem can be subdivided into subproblems, each of which allows a local optimal solution. Thus the best decision for each subproblem sums to the best decision for the entire problem. The key idea is to avoid solving the same problem multiple times by saving the solution to each subproblem. This is a bottom up approach. The Fibonacci recursion is a primary example: `F[n] = F[n-1] + F[n-2]`

Here is the dynamic programming algorithm applied to tree covering:

```
Tree-cover[T(V,E)] =def=
  Initialize cost of the internal vertices to -1;
  Initialize the cost of leaf vertices to 0;
  while some vertex has a negative weight do          /* bottom-up
    select a vertex v whose children have all nonnegative costs;
    M := set of all matching pattern trees at vertex v;
    L := set of leaves in T matching the leaves of pattern tree M;
    cost[v] := min[cost[M[i]] + sum-over-j[cost[L[j]]]].
```

Greedy Algorithm

This is a top-down approach. Use this when

1) the problem can be decomposed into local optimal decisions, and
2) making a particular locally best decision reduces the problem size.

Here is the greedy algorithm applied to task scheduling. We are given a set of tasks T, with each task having

a duration	`length[T[i]]`,
a release time when the task can begin	`release[T[i]]`, and
a deadline for completion	`deadline[T[i]]`

Find a schedule (an ordering of the tasks) which meets the release times and deadlines. The optimization problem is to find the minimal time schedule.

```
Greedy-scheduling[T] =def=
  i = 1;
  repeat
    while (Q := unscheduled tasks with release time < i)==0 do i++;
    if there is an unscheduled task p such that
      (i + length[p] > deadline[p])
      return FALSE;
    select Q[j] with smallest deadline;          /* greedy decision
    schedule Q[j] at time i;
    i := i + length[Q[j]];
  until all tasks are scheduled;
  return TRUE.
```

The greedy decision is to take the most constrained task when a task is selected. This may result in missing a possible solution, since selected tasks are not rescheduled by the algorithm (i.e. no search for the best solution).

E.g.: consider three tasks T = {a,b,c}, with length[1,2,3], release[1,1,3], deadline[4,inf,6].

Another example: what are the minimal number of coins that add to 63 cents? The greedy approach automatically uses the largest denomination coins first.

Backtracking

Sometimes there are no hints about the best choice. We simply must guess. Backtracking keeps track of that guess, and revokes it if necessary to solve the problem. Example is a complex game like chess, played by a non-expert.

There is a single generic search algorithm for tree and graph search:

```
Generic-search[current-state,search-list] =def=
  if current-state=nil or current-state=goal
    then done
    else search-list =
      special-merge[get-children[current-state],search-list];
      generic-search[first[search-list],rest[search-list]]
```

When the current-state is not the goal, generic-search puts the children of the current-state on the search-list and recurs. The `special-merge` function determines the type of search:

depth-first	add the new nodes on the front of the search-list
breadth-first	add the new nodes on the end of the search-list
iterative-deepening	like depth-first, but add new nodes to the front of the search-list only when their depth is not greater than the current cut-off, otherwise add new nodes to the end of the search-list
iterative-broadening	like breadth-first, but add new nodes to the front of the search-list only when their breadth is not greater than the current cut-off, otherwise add new nodes to the end of the search-list
hill-climbing	sort the new nodes by their estimated distance from the goal and add them to the front of the search-list
best-first	add the new nodes to the search-list, sort the entire list by estimated distance to the goal

Graph Algorithms

Graph algorithms are pervasive in Computer Science. The graph is such a general data structure that almost all computational problems can be formulated using one of the primary graph processing algorithms. Lists and trees are subsets of graphs. The major problems in hardware synthesis, operating system scheduling and sharing, compiler optimization, software design and minimization, network communication and synchronization, and requirements and specification modeling are graph problems. Variations of the fundamental graph algorithms of traversal, covering, and coloring model the problems in these and other application areas.

Types of graphs

A ***graph*** is a collection of *vertices* `V` (also called nodes) and *edges* `E` (also called arcs or links) which connect the vertices. A single edge can be described by the two vertices which it connects: `e = (u,v)`

undirected graph: The edges can be traversed in either direction. Undirected graphs are either trees or cyclic.

cyclic graph: The graph contains loops than permit a node to be visited recurrently.

directed graph: The edges are directional and support only one direction of traversal.

directed acyclic graph: There are no paths which permit visiting a node more than once.

weighted graph: The edges have a numerical value which may represent the cost of traversing that edge. Weights can be both positive and negative.

connected graph: All of the vertices can be visited when starting from any vertex. The list of vertices visited between a starting vertex and an ending vertex is called a *path*.

sparse graph: There are many more vertices than edges, that is `E << V^2`.

dense graph: The number of edges is about the same as the number of vertices squared; `E =~= V^2`

complete graph: Every pair of vertices has an edge connecting them. The graph is thus maximally connected.

clique: A subgraph (that is a subset of nodes and edges in a graph) which is complete.

bipartite graph: A graph with two sets of vertices, such that each edge connects vertices from different sets. No edges connect vertices in the same set.

hypergraph: A graph with edges that connect more than two vertices. All hypergraphs can be converted into regular bipartite graphs.

graph complement: The complement of a graph is a graph with the same vertices, but all edges exchanged (i.e. if an edge is present, then delete it; if an edge is missing, add it).

Representations of graphs

The choice of graph data structures depends on the density of the graph connectivity. The complexity of different algorithms depends both on the number of edges and on the number of vertices. Unlike sets, graphs often have a direct representation in memory. When the data field of a memory cell contains an address of another cell, the cell can be interpreted as a node and the shared address can be interpreted as a link.

adjacency lists

A collection of `V` lists (usually stored as an array), each listing the adjacent vertices of a particular vertex. For directed graphs, the sum of entries in the `V` lists will equal the number of edges. For undirected graphs, it will equal twice the number of edges. Adjacency lists use memory O(V+E). Use this for sparse graphs.

adjacency matrix

A square matrix of `V` Boolean entries, with a `1` recording adjacent vertices, and a `0` recording non-adjacent vertices. Use this for dense graphs.

indexed arrays

For graphs which change during processing, two arrays are used. The first array contains the `V` vertices, and provides an index for each vertex. Each indexed vertex in the vertex-array contains all the information about the vertex, including its processing status (visited, frontier, or not-visited). This array is changed only when the status of the vertex changes, the vertex index is never changed. The second array contains the `E` edges, and provides an index for each edge. The data field of the second array contains the indices of the two vertices connected by that edge. The connectivity of the graph is changed by changing the edge array entries. Graph traversal is managed by a priority queue of edge indices to be visited.

Topological Sort

Sorting lists and trees makes them easier to process, since the vertices have a consistent ordering. Directed acyclic graphs (DAGs) can also be sorted by the partial ordering implicit in the graph connectivity. Cyclic and undirected graphs do not have a consistent sort.

A ***topological sort*** of a DAG is a consistent ordering of the vertices that serves as an ordering for processing vertices. One convenient sorting is by length of shortest paths from the root vertex to the leaf vertices. The complexity of a topological sort is linear, O(V+E).

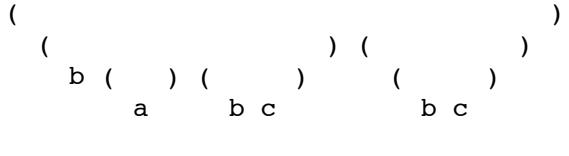
Interestingly, graph sorting is faster than both list and tree sorting, when the tree sort is done by comparing elements. There are linear-time list sorting algorithms (counting, radix and bucket sort). These rely on *a priori* encoding techniques for the input. Graph sort also relies on a particular type of input (i.e., an adjacency list). This structure is not, however, related to the type of input data, since data itself is sorted at each graph node.

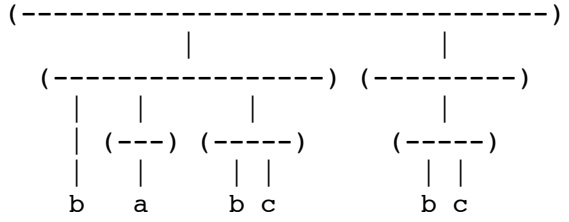
Parenthesis structure

Directed acyclic graphs can be expressed in a linear parenthesis notation which is convenient for textual processing, computation and analysis.

Example

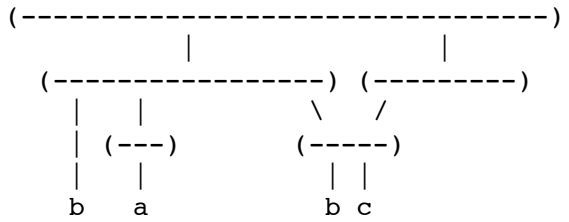
```
V = {1,2,3,4,5,a,b,c}        E = {12,13,24,25,2b,35,4a,5b,5c}

      G = ( ( b ( a ) ( b c ) ) ( ( b c ) ) )
          1 2   4     5          3 5
```





Structure sharing in graphs is represented by labeling:

```
G = ( ( b ( a )    5  ) (      5    ) )
5 = (b c)
```



Since parenthesis structure represents both lists and trees naturally, the parenthesis structure of graphs provides a uniform representation for most common CS data structures (except arrays).

Primary graph algorithms

searching

The search algorithm traverses a graph, usually identifying the shortest (or longest) path between two vertices. When the graph has a source (a root, or beginning, vertex) and a sink (a set of terminal vertices), beginning and ending points are defined by those vertices. A variant of the path problem when there is no preferred start or finish is to find the shortest (longest) path between any pair of vertices.

Spanning trees list paths that connect all vertices. The spanning tree problem is to identify the smallest (largest) number of edges which span the graph. The travelling salesman problem (i.e., what is the shortest route available to visit a collection of sites) is an example of computing a minimal spanning tree.

A ***Hamiltonian path*** is a path which visits each vertex while not traversing the same edge more than once. This is an example of a graph theory problem little application to CS.

covering

A ***vertex cover*** is a subset of vertices of an undirected graph such that every edge has at least one end in the subset. The covering algorithm visits every vertex, identifying the minimal number of vertices in the covering subset. Spanning trees necessary cover a graph, but not minimally. Every alternate vertex in a spanning tree also forms a cover.

Minimization of Boolean expressions (by Karnaugh maps or by algebraic techniques) is an example of graph covering. Each product term in a sum-of-products (SOP) representation is a subset of vertices of a Boolean hypercube. The covering problem is to include each vertex in the SOP using a minimum of sub-cubes of the hypercube.

coloring

A ***vertex coloring*** is a labeling of the vertices of an undirected graph such that no edge has two endpoints with the same label. The coloring algorithm visits every vertex, identifying the minimal number of colors necessary to label every vertex.

An example of graph coloring is scheduling and resource allocation using a ***resource conflict graph***. Vertices in a resource conflict graph represent operations to perform; edges represent pairs of operations which are in conflict because they cannot use the same resource. (A ***resource compatibility graph*** is the complement of the conflict graph.)

Graph searching and shortest paths algorithms

Breadth first search

```
BFS(G) =def=
    for each vertex v excluding the starting-vertex do
        color[v] := white;
        distance[v] := infinite                  /* initialization
    color[starting-vertex] := gray
    distance[starting-vertex] := 0
    queue := starting-vertex
    while not[null queue] do
        v := head[queue]
        for each u in adjacent[v] do
            if color[u] = white
                then color[u] := gray
            distance[u] := distance[v] + 1
            push[queue,u]                  /* always gray in queue
            pop[queue]
            color[v] := black
```

Depth first search

```
DFS(G) =def=
    for each vertex v do
       color[v] := white
    time := 0
    for each vertex v do
       if color[v] = white
        then dfs-visit[v]

DFS-VISIT(v) =def=
    color[v] := gray
    time := time +1
    gray-time[u] := time
        for each u in adjacent[v] do
           if color[u] = white
               then dfs-visit[u]
    color[v] := black
    time := time +1
    done[time] := time
    store-topological-sort[v]
```

"Colors" are used to identify the process state of a graph. Below, white refers to *not-yet-processed*, gray to *currently-on-the-frontier*, and black to *done-processing*.

This algorithm outline can be generalized by assigning each vertex a *visit-time*, which then identifies a depth ordering for the vertices. Depth-first visit-times thus count the path length from the start node to each other node. A topological sort can be stored by recording this timing information, providing an easy data structure for computing the length of all traversal paths.

Depth-first search is O(V+E) for an adjacency list representation and O(V^2) for an adjacency matrix representation.

Shortest path questions can be converted into longest path questions simply by reversing the sign of weights for each edge.

Interestingly, ***binary search trees*** are a way to organize data so that search is O(lnN), i.e. proportional to the depth of the tree. Graphs are thus most costly to search. However, since we are at worst using linear algorithms, this is rarely a problem.

Dijkstra's minimal weighted path algorithm

```
Path[G(V,E,w)] =def=
    s[0] = 0;                  /* s is the path
    for i=1 to n do
        s[i] = w[0,i]          /* initialize path weights
    repeat
        select an unmarked vertex v[q] so that s[q] has a minimal weight;
        mark v[q];
        foreach unmarked vertex v[i] do
            s[i] = min[s[i], s[q]+w[q,i]];
    until all vertices are marked.
```

Relaxation

```
init-min-path[G,s] =def=       /* s is the starting vertex
    for each vertex do
        d[v] = infinity;       /* the estimated value to be successively reduced
        path[v] = nil
    d[s] = 0

edge-relax[u,v,w] =def=
    if d[v] > d[u] + w[u,v]
        then d[v] = d[u] + w[u,v]
            path[v]=u

dag-shortest-path[G,w,s] =def=
    topological sort G;
    init-min-path;
    for each vertex in sorted order do
        for each vertex in adjacent[v] do
            relax[u,v,w].
```

The general technique of *relaxation* consists of testing whether a result can be improved by a particular choice by trying that choice. The choice is accepted if it is an improvement, or revoked if it is not an improvement. The upper bound on a result is thus continually decreased. Consider the shortest path algorithm as an example:

Minimum Spanning Tree

```
MST[G,w] =def=
    A := 0;
    while A does not form a spanning tree do
        find an edge [u,v] that is part of the spanning tree
        A := A union [u,v];
    return A
```

Vertex cover

Graphs may provide multiple paths between vertices. The minimal vertex cover is the smallest number of vertices which include at least one endpoint of each edge. The algorithm below is heuristic, providing a cover but not necessarily the minimal cover. Finding the minimal cover is intractable, O(2^n).

```
Vertex-cover-Vertex[G(V,E)] =def=
    C = 0;                          /*initialize cover
    while E =/= 0 do
        select a vertex v;
        delete v from V;            /* remove edges incident with v also
        C = C union v.

Vertex-cover-Edge[G(V,E)] =def=
    C = 0;
    while E =/= 0 do
        select an edge e[u,v];
        C = C union {u,v};
        delete u and v from V.      /* remove edges incident with u and v also
```

Vertex coloring

A *vertex coloring* of a graph is a labeling of the vertices such that no edge has two vertex endpoints with the same label. The optimization problem is to identify the minimal number of colors for a graph. The algorithm below is heuristic; find the minimal coloring is intractable.

```
Vertex-coloring[G(V,E)] =def=
    for i=1 to V do
        c = 1;                      /* c is the number of colors
        while there exists a vertex adjacent to v[i] with color c do
            c = c + 1;
        label v[i] with color c.
```

Mathematica

The Philosophy

> The programmer's time is more valuable than the processor's time. Thus, the architecture is interpreted (interactive), real-time, and goal-oriented.

"Programs you write in Mathematica may nevertheless end up being faster than those you write in compiled languages" p.506

"The internal code of Mathematica uses polynomial time algorithms whenever they are known." p.63

> Mathematica accepts code in all of the modern programming paradigms.

"All the approaches are in a sense ultimately equivalent, but one of them may be vastly more efficient for a particular problem, or may simply fit better with your way of thinking about the problem." p.487

"As a matter of principle, it is not difficult to prove that *any* Mathematica program can in fact be implemented using transformation rules alone." p.503

The Mathematica Program

a general purpose computational engine for
- numerical calculations (arithmetic)
- symbolic transformations (algebra)
- graphic display (geometry)

a modern programming language with multiple styles
- procedural
- functional
- logical
- object-oriented
- rule-based

an integrated tool
- C, TeX, UNIX, Postscript

Everything is an Expression

"At a fundamental level, there are no data types in Mathematica. Every object you use is an expression, and every function can take any expression as an argument." p.496

`x+y`	`Plus[x,y]`
`120`	`Integer[120]`
`2ab`	`Times[2,a,b]`
`{a,b,c}`	`List[a,b,c]`
`i = 3`	`Set[i,3]`
`x^2+2x+1`	`Plus[Power[x,2],Times[2,x],1]`

An undefined symbol is *itself*, providing functional transparency and WYSIWYG debugging

The Meaning of Expressions

`F[x,y]` `F` is the *head*. `x,y` are the *contents*.

Apply function `F` to arguments `x` and `y`.
Do action `F` to objects `x` and `y`.
The label `F` points to elements `x` and `y`.
The object-type `F` has parts `x` and `y`.
The arguments `x,y` are of data type `F`

The head can both act on its contents (as a function) and maintain the structure of its contents (as an object), depending on context.

Lists

The List container is used for all collections and all database entries:

data record	`{John, 555-1234, j@mma.com}`
matrix	`{{11,12},{21,22}}`
set	`Union[{a,b},{a,c}] ==> {a,b,c}`
graphics spec	`Line[{{0,1},{1,1},{1,0}}]`
stream	`{1,2,3,...}`
structure template	`{_,{_,_},{{_,_},...}}`

The Fundamental Principle of Computation

Take any expression and apply transformation rules until the result no longer changes.

1. Reduce head	
2. Reduce each element	base case arithmetic
3. Standardize	
4. Apply user defined rules	inductive case algebra
5. Apply built-in rules.	
6. Reduce the result.	recursion

Patterns

A *pattern* is a class of expressions with the same structure.

`_`	"underbar" means *any* expression
`x_`	any expression locally named `x`
`x__`	any sequence of expressions (double underbar)
`x___`	any sequence, including none (triple underbar)
`x_h`	any expression with head = `h`.

Examples:

`f[n_]`	the function `f` with a parameter named `n`
`2^n_`	2 raised to any power
`a_ + b_`	the sum of two arbitrary expressions
`{a__}`	a list with at least one element

Object-oriented Organization

```
square/:    perimeter[ square[n_] ] := 4*n
square/:    area[ square[n_] ] := n^2
circle/:    area[ circle[r_] ] := Pi*r^2
```

The outer "function" transforms the inner "argument".
The inner "object" contains a private outer "message handler".
The outer "matrix" is indexed by the inner "accessors".

Final Assignment: Control Structures

A three to five minute presentation to the class on your approach.
Time allocation (max): thinking, 8 hours; designing, 8 hours; implementing 20 hours
HAND IN YOUR WORK (notes, code, comments, results)

Select from and complete as many of the exercises as you can within the time allocation.

I. Sorting

Comprehension exercise: Implement several *sorting algorithms* (possibly by copying algorithms from the book). Design and implement a *record generator* which generates random collections of record indices (ie numbers or key words). Design and implement a simple *test statistics package* which counts the number of sorting steps for each sorting algorithm. Answer these questions:

1. Make a graph of sorting steps (record swaps or relocations) vs size of input (number of records) for each sorting algorithm. Do your algorithms perform as the book predicts?

2. Characterize the essential difference between each algorithm. Why do some algorithms perform better or worse than others?

3. Design several different input orderings to sharpen the difference in performance between your algorithms. That is, build input sets which are almost completely ordered, almost completely out-of-order, in some random ordering, each value duplicated once, all the same value, etc. Vary the size the input sets and their ordering characteristics, and test the efficiency of your algorithms.

4. For all experiments, abstract the performance in terms of asymptotic notation.

5. Characterize the stability of each algorithm. That is, differentiate between sorting which may move elements with the same key, and those that will not move them.

6. ***Hybrid algorithm*** **(challenge):** mix what appear to you to be the best parts of your algorithms, to build a hybrid sorting algorithm with better characteristics than the ones you started with.

II. Algorithm Variety

1. Factorial: Implement many substantively different versions of the factorial algorithm (factorial computes the product of 1..n integers). Compare each for efficiency and for ease of writing and maintenance.

If you look on the web, there are collections of dozens of ways to implement factorial (that is, this problem can be addressed with research as well as with creativity). Some methods require an appropriate engine, which you may or may not have. For example, an object-oriented implementation requires an object-oriented language. Don't implement engines, do use different languages when appropriate.

Can you find the fastest possible algorithm?

2. Fibonacci: Implement many substantively different versions of the Fibonacci algorithm Fibonacci computes the sum of the previous two Fibonacci numbers:

```
Fib[0] = 0
Fib[1] = 1
Fib[n] =  Fib[n-1] + Fib[n-2]
```

Compare for efficiency and ease of use. Identify three classes of algorithm (highly inefficient, efficient, and highly efficient) with an implementation of each.

3. Generalization (challenge): Design and implement an abstract procedure which computes any simple recursive function, given the base and the recursion relation.

III. Tree Searching

Assume that you have a tree data structure and you have to search it for a leaf with a particular value.

1. Searching: Design and implement several search algorithms for visiting the leaves of the tree. Standard techniques include depth-first, breadth-first, best-first, hill-climbing, iterative deepening, and iterative broadening. Which are most efficient and why?

2. Structured search: In what ways can specialized tree data structures (such as balanced search trees) help to improve the efficiency and ease of programming search algorithms? Analyze the tradeoff between complex data structures for search and complex search algorithms.

3. Generic search: Design and implement a single algorithm which takes the type of search as a parameter. Other generalization parameters which may be of interest include

- the goal predicate which tests when the leaf being sought after is found
- a function which returns the children of an interior node in the tree. (This is used for determining the cost of various search strategies.)
- a priority function which determines which node to search next.

4. Smart search (challenge): Design and implement an algorithm which dynamically determines which kind of search is best for the particular context, given the depth, branchiness, and other characteristics of the tree at the node currently being explored.

Final Assignment (option): ADS for Control Structures

Design an abstract data structure for program control structures.

Caution: This exercise is too difficult to assign a legitimate due date or time allocation, because of its exploratory nature. No one has been able to do the task of control structure abstraction well. Thus, this exercise requires a research mentality. Think about how to do it, what the task requires, the tools you have available, and where the hard and easy parts are. Sketch a partial solution and do the parts that you believe are possible. Almost all time should be spent puzzling about what the exercise means. Do not expect to complete any subparts of this assignment.

Definition: Program control structures are those components of a program which determine the sequence and style of program execution. The *semantics*, or meaning, of a program is defined by its behavior, which is guided by control structures. The major program-level control structures are:

- logic (Boolean primitives such as AND, OR, NOT, EQUALS, IF-THEN-ELSE, CASE)
- assignment (assigning names to return values)
- loops (structures which repeat instructions a specific number of times) including iteration, recursion, and {DO, FOR, WHILE, UNTIL}
- sequencing (calls that are executed sequentially as they are written in the program)
- function invocation (calls to specific functions)
- mapping (applying a function to a collection of items)
- catch and throw (jumps from one part of the program to another, usually in exceptional circumstances)

Level of effort: First try to design and implement an ADS which generalizes one of the above control structures. The above structures are elementary, similar to the elementary data structures of bits, arrays and pointers. The idea is to build higher-level control structures from these primitives. The following discussion may help you to choose a difficulty level.

Logic is expressed by propositional calculus or by Boolean algebra. An ADS for logic would identify possible Boolean structures (the propositions in propositional logic can be functions that return a specific value). Most languages extend the semantics of logic so that any return is viewed as True. But how would you specify a logic structure abstractly?

Assignment looks simple, but has been one of the most difficult concepts to understand. Naming is inherently tricky, and naming to a memory location is trickier still. Scoping rules make the duration of assignment tricky also. Several languages do not use assignment, it is not an essential concept. Good programming isolates all assignments. ASSIGNMENT is still very much in use, but it is beginning to look like a bad idea. It is being replaced by LET and by parameter passing in FUNCTION INVOCATION.

Loops are a dominant tool for repetitive actions. They come in many varieties. It is easy translate between FOR, DO, WHILE and UNTIL. Converting between iteration and recursion is significantly harder, but not tricky. Iteration repeats over a data structure (the loop index); recursion repeats over a function invocation. Newer repetitive techniques use an *iterator* data/control structure which generates items as needed. *Streams* and *mappings* can also drive a repetitive call without introducing looping.

Sequencing is fairly easy if it is not mixed with other control structures. Assignment and goto make sequencing particularly difficult.

Function invocation is captured by the rules of lambda calculus, and is not difficult. Since lambda calculus provides two evaluation regimes, there are significant design decisions about what to evaluate when. This has driven the debate between *eager and lazy evaluation*.

Mapping is easy and is widely under utilized in languages. It is very similar to implicit looping.

Catch and throw are dynamic exits into non-local environments. They are principled and easy to model since they essentially throw away intermediate results. But since they cross algorithmic boundaries, they have very limited use (ie for exception handling and occasionally when a loop must be interrupted). GOTO (jumps out of a program without encapsulation) is a control structure which is antiquated and no longer in use.

Algorithms: When the simple control structures above are combined to make a compound program, that program is an algorithm. Your ADS should be able to handle algorithms. The essential utility of the control structure ADS is to convert algorithms which do the same task into different programming approaches or metaphors. This is called a *meta-protocol*. We all know that we can choose between different ways of implementing a specification, iteration vs recursion for example. Your ADS should be able to isolate the choice of a particular control structure from the meaning of the specification.

Examine several SORT algorithms. How do they differ in control structure? How can the concept of sorting be separated from the implementation of sorting? How can the concept of sorting be separated for the various sorting algorithms?

Examine several tree traversal algorithms. How do they differ? How can they be abstracted into one algorithm which steers the search based on a parameter?

What other compound control structures do you commonly use? How would you design a program which could switch between members of a family of algorithms (e.g. the SORT family) depending on the incoming data structure, expectations about the average data structure being processed, and the available computing resources?

Discussion: This assignment is intended to let you think about a hard problem, to use the ADS template on difficult structures in unique ways. Be sure to follow the ADS guidelines. Be sure that you understand the limitations of your language of choice; often language design will preclude your access to manipulating control structures (this is because it is easier to implement languages when control is fixed by a design choice). For example, C++ uses assignment, sequence (or *compound*), logic, and for-loops as elementary control structures.

Stick to a subset of your programming language which avoids the difficult constructs. (Then learn to program like this all the time, never using the problematic ideas of assignment, goto, and possibly loops and variables.) Assume a **pure language** which does not include very tricky concepts like memory allocation and deallocation, destructive operations on memory, dynamic scoping, global variables, and non-local jumps.