

# GRE CS EXAM STUDY GUIDE

## DISCLAIMER:

I wrote these notes for myself as I was preparing for the GRE in '04. I firmly believe that anyone else should do the same and not rely on mine. It was during the organizing of these notes that I learned and reviewed the material. If you don't go through that process for yourself, you are probably lulling yourself into a false sense of security. Several other things can go wrong:

1. I got the lowest possible passing score. That means you may be smarter than me and picked the wrong source for notes.
2. I organized these notes based on what I needed to learn. I stressed the areas that I needed work on and downplayed or ignored other areas. That may be just the opposite for you.
3. There may be errors.

If you are still reading after the above disclaimer, you will probably read the rest. I sincerely hope it helps you. Best of luck,  
Tom Peterka

## 0. Difficulties with GRE exam of 11/13/04

### 0.1 Strassen's method for fast matrix multiplication

2 problems with determining recurrence step and time complexity (full page description of algorithm) passed on the problem  
could do it with some background on how Strassen's method works to save having to decipher the description (not enough time for that)

### 0.2 trouble with one cache problem – need to learn to do cache problems better (CS466 cache chapter, examples and chapter problems)

### 0.3 wasted much time w/ one breadth first search problem moving chips from one side to another – don't know why it didn't compute

### 0.4 attribute grammars – what are they?

### 0.5 review amount of video ram required for certain resolution and number of colors, eg 1024x2048x64K colors

## 1. Software systems (40%)

### 1.1 Programming

- tracing program flow, determining final output values, # of loops, etc.
- passing values by value vs. by reference
- Loop invariants

a condition that is true every time the guard of a loop is evaluated, including the iteration when the guard evaluates to false and the loop is not taken

eg:

```
x = 0;
i = 0;
while (i < n){
    x = x + 2i;
    i = i + 1;
}
```

a loop invariant for the while loop is  $x = 2^i - 1$  and  $0 \leq i \leq n$  (note that  $i \leq n$ , not  $i < n$ )

### 1.2 Software engineering

#### 1.2.1 Software life cycle model

- Requirements specification - the hardest thing to get right and to fix later is determining what the software system is to do
- Design
- Implementation
- Testing
- Maintenance – maintenance cost is 50% to 65% of the total life cycle cost (for moderately sized packages of 100,000 lines) . This is the costliest and most time consuming part of the software life cycle. Using reusable software as often as possible is a good way to reduce maintenance overhead.

#### 1.2.2 historical perspective

In the 50's and 60's a good program is one that was fast, because CPUs were slow and CPU time was expensive and thus the critical resource.

In the 80's and 90's, the primary qualities of a good working program are that it is readable and maintainable, since maintenance is the costliest part of the software life cycle.

#### 1.2.3 Design patterns

##### Singleton design pattern

Guarantees that only one instance of a class is instantiated.

- A static member function provides its instance, traditionally called Instance(), that is called by the user to construct the class
- A private constructor that is called from Instance() only upon the first invocation

- Otherwise all other features exist, such as subclassing the Singleton class from another class

#### 1.2.4 Function preconditions and postconditions

A set of specifications for a function's operation. If the precondition is met, the function is guaranteed to meet the postcondition. It is the caller's responsibility to meet the precondition; the function assumes it has been satisfied. It is desirable to write the *weakest* precondition and *strongest* postcondition for a function.

#### 1.2.5 Cohesion and coupling

Functions should have strong cohesion (within a function) and loose coupling (between functions).

## **1.3 Operating systems**

### **1.3.1 Introduction**

#### **Buffering (DMA)**

The reading of an entire block of data to/from memory on one interrupt. The CPU initiates the transfer, but the memory controller actually does it, freeing the CPU. The memory bus is “borrowed” from the CPU as needed to perform the transfer. The i/o of one process overlaps its own computation.

#### **Spooling**

The writing of input to a file on disk before outputting (eg printing it) rather than tying up the cpu to go direct. In this way, the i/o of one process can overlap the computation of another. Spooling is most beneficial in a multiprogramming environment when processes are evenly divided as i/o bound and cpu bound.

#### **Multiprogramming**

Maintaining several programs in memory and alternating between them whenever one needs to wait for i/o or based on some other scheduling scheme such as round robin. The goal is to maximize cpu utilization and minimize idle time.

#### **Time-sharing**

Extending multiprogramming to programs from several users, giving each user the impression that he is the only one using the machine.

#### **Protection**

Protecting the o.s. from invalid access by application programs.

### 1.3.2 File systems

#### Blocking

The restriction on file size to be an integral number of some block size. Larger block size leads to more efficient i/o but also leads to *internal fragmentation*. (the wasted space in the last block). Note that these are logical blocks. Physical blocks, also called *record blocks*, are a function of the hardware storage medium. (eg disk sectors)

#### Free space management

- Bit vector or bit map

A string of bits, one bit for each block, that indicates whether it is free or not

- Free list

Free blocks are linked through a pointer at the end of the block that points to the next one

#### Allocation methods

- Contiguous

A file must occupy contiguous blocks on the disk, and enough contiguous space must be found in order to accommodate it.

Dynamic storage allocation: First fit, best fit, worst fit. Dynamic storage algorithms under contiguous allocation suffer from *external fragmentation*, ie, unused blocks not large enough to fit a file contiguously.

- Linked

All blocks of a file are linked together. Solves external fragmentation problem, but does not allow random access to a block in a file, only sequential traversal of all blocks until desired is reached.

- Indexed

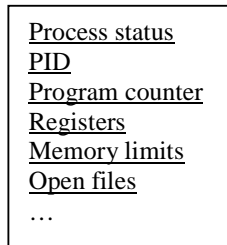
A block serves as a table of indices for all blocks of all files. Files need not be contiguous, and random access is possible. If one block is not large enough to hold all indices, blocks can either be linked or hierarchically organized.

The *activity* of a file refers to the percentage of existing records updated during a run. The *volatility* of a file refers to the number of records added or deleted from a file compared to the original.

Unpaged or read-ahead cache associates cache domains with start address of the read and continues for a specific length. The major disadvantage of unpaged cache is it allows cache domains to contain redundant data.

### 1.3.3 CPU Scheduling

#### 1.3.3.1 Process control block PCB (6 items minimum)



The ready Q for the cpu is a linked list of processes waiting for cpu time

#### 1.3.3.2 scheduling algorithms

choosing a process from the ready Q

- first come first served FCFS  
non optimal average turnaround time
- shortest job first SJF  
provably optimal  
hard to predict cpu length of next burst  
can be used for long term job scheduling but not for cpu short term scheduling

2-part SJF:

To find an optimal non-preemptive schedule of several jobs each consisting of 2 parts that must be performed in order on separate processors, sort all parts together from shortest to longest. If shortest part is a part 1, schedule job from the front and remove both parts of that job from the list. If shortest is a part 2, schedule that job from the back. Therefore, at the start of the run the second processor will be idle for the shortest time (waiting for the first to finish) and at the end of the run the first processor will be idle for the shortest time (waiting for the second to finish).

- Priority

SJF is an example of priority based on cpu requirements. Other factors can be used to determine a different priority ordering

Starvation is a problem solved by **aging**, where the priority is gradually increased at fixed time intervals until the priority is high enough to be executed

- round robin

too short time slice: much overhead for context switches  
too long time slice: degenerates to FCFS

### 1.3.4 Memory, virtual memory, cache memory

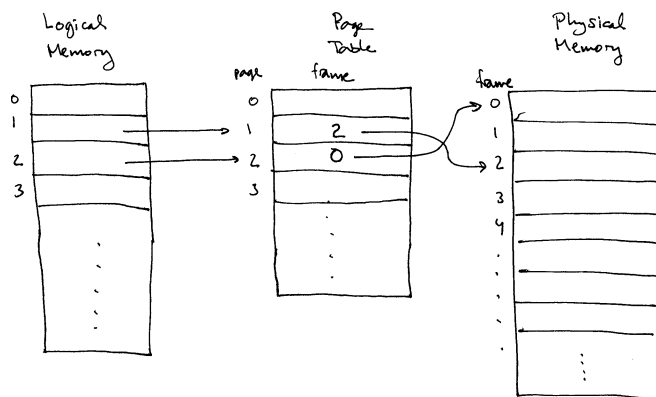
#### Multiple processes in memory

- Contiguous

☞ Suffers from external and internal fragmentation (external = wasted space between regions; internal = wasted space within a region)

- Paging

Solves external frag. problem but has internal frag. due to fixed size pages



Page# = logical address / page size

Offset = logical address % page size

In binary, if page size =  $2^n$ , this is done w/o dividing

☞ Part of page table kept in dedicated cache called transfer look-aside buffer TLB. Since every single memory access must go through the page table lookup, it must be very fast. TLB uses typical cache replacement algorithms like FIFO, LRU, etc. It is a set associative cache, so it is searched in parallel.

If desired page is found, access time = cache access + main memory access

☞ If desired page not found, access time = cache access + 2 \* main memory access

Hit ratio H and average access time is computed as:

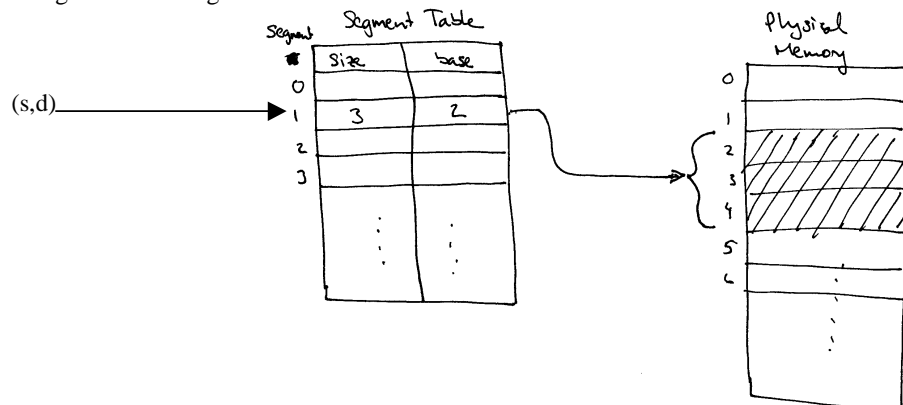
avg access time =  $H(\text{cache access} + \text{memory access}) + (1-H)(\text{cache access} + 2 * \text{memory access})$

- Segmentation

Segments are variable size “pages” based on size of code objects (structures, classes, functions, etc.)

Logical requests are divided into segment# (s) and offset(d)

Segmentation solves internal frag. but suffers from external frag. if there are no regions of contiguous memory large enough to hold a segment.



- Combinations of paging and segmentation

Segmented paging and paged segmentation are possible

- Virtual memory

Increases space (makes memory seem larger than it really is)

It is the extension of memory to allow pages or segments to reside on disk and be brought into memory as needed. Replacement algorithms are used to determine which page or segment is chosen for replacement.

Virtual memory can be paged, segmented, or a combination can be used as described above.

- Cache memory

Increases speed (makes memory seem faster than it really is). Caching good for small loops (entire loop fits into cache)

- Memory-cache mapping
- Direct mapping - Memory addresses map directly to cache blocks (1-way)
- Associative mapping - Memory addresses can map anywhere in the cache (n-way)
- Set associative mapping - Memory addresses can map anywhere within subsets of the cache(eg 4-way)

eg:

given a 4KB cache with 128B block size, number of bits for tag, index, block offset for direct mapped, fully associative, 2-way set associative, and 4-way set associative mappings

block offset bits = 7 because  $2^7 = 128$  (block size) (this is constant for all arrangements)

$4\text{ K} / 128 = 2^{12} / 2^7 = 2^5$  or 32 slots where the block can go

Those slots can be partitioned into 1 way (direct), 32 ways (full), 2 ways, 4 ways, etc.

number of bits: (assume 32 bit address)

ways	tag bits	index bits	block offset bits
1-way(direct)	25	0	7
n-way(full)	20	5	7
2-way	21	4	7
4-way	22	3	7

each cache entry also has a valid bit V to indicate that there is something there (initially V=0)

Replacement algorithms

LRU algorithm most common. Also FIFO and random (not very commonly used)

Write back policy

Write through (every write updates both cache and ram)

Write back or copy back (ram updated when block is evicted)

Write miss policy

Write-allocate – update the block and read it into cache for next time

Write-no allocate – just update memory and don't read into cache

Often write-through accompanies write-no allocate and write-back accompanies write-allocate, but not always




Average effective access time =  $H(\text{cache}) + (1 - H) * (\text{cache} + \text{memory})$

Note: sometimes the cache access time is ignored for a cache miss, which can affect the final result, depending on whether a cache miss is considered to take the same time as a cache hit.

- Replacement algorithms

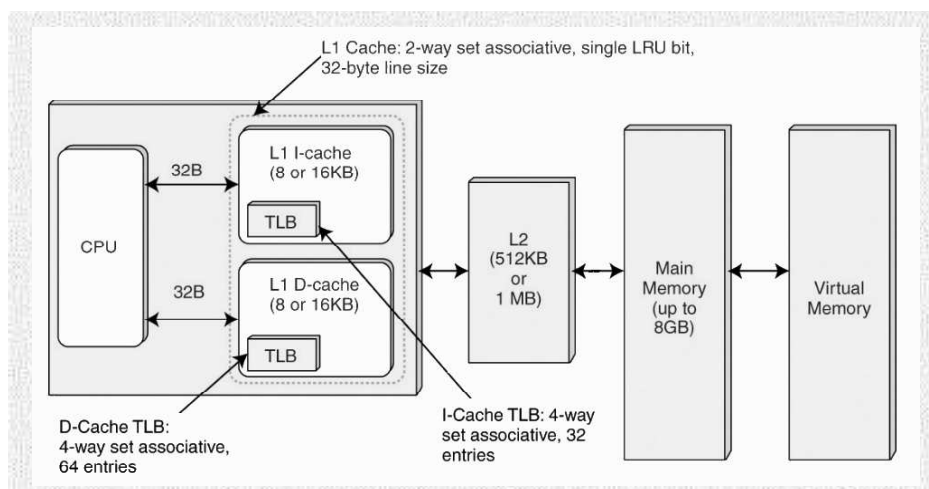
LRU most common; Others are NRU, LFU, MFU

LRU is difficult to implement however, so often a page is tagged with a reference bit indicating if it has been accessed or not. The reference bit does not indicate how long ago it was referenced, so this becomes and NRU (not recently used ) algorithm, which is a rough approximation to LRU.

 Note: if the page references are random and a large number occur, LRU and FIFO will result in the same number of page faults. Ie, for an even distribution of references, no locality occurs and LRU has no effect.

- Storage hierarchy

Cache, memory, and disk combine to form a storage hierarchy as illustrated in the following pentium architecture:



(ref: <http://kr.mnsu.edu/~lct/www/320Ch6.ppt>)

- Memory interleaving

Memory is divided into banks and an address within a bank (similar to page number and offset, but this is a physical division of memory into physical banks)

- Garbage collection

Automatically reclaims allocated memory objects whose content cannot affect future computations. Such objects are identified by determining they cannot be reached from a root set. A root set includes:

- Actual parameters of active procedures
- Global variables of the program
- Local variables on the call stack
- Machine register values
- *Not in the root set: dynamically allocated objects on the heap*

One method of automatically collecting garbage is reference counting, but it does not work for cyclic structures.

- Shared pages

A page or pages can be shared by multiple processes if they are reentrant (read-only). Variations are read/write access where processes have unlimited access, and copy-on-write where the shared copy is used until some process wants to do a write. At that point all processes get their own copy.

Eg: a debugger is a separate process that needs read/write access to the binary pages in order to place breakpoints.

### 1.3.5 Deadlock

4 conditions necessary for deadlock (all 4 must occur)

- Mutual exclusion – resources cannot be shared by processes
- Partial assignment of resources – a process does not have to wait for all required resources to become available before a resource is assigned to the process (hold and wait)
- Non-preemption – once resources have been assigned to a process they cannot be taken away until the process releases them
- Circular wait queue – deadlock can be prevented by imposing a linear ordering on resources because a circular wait queue cannot occur.

Preventions / cures:

- process must request all resources before beginning
- numbering resources and having process request them in increasing order
- having processes time out after an interval of waiting, or being killed and restarted by OS
- *not giving priorities to processes and ordering wait queues by priority*



### 1.3.6 Disk storage

#### 1.3.6.1 Scheduling algorithms

First come first served FCFS

SSTF: shortest seek time first

SCAN: (elevator algorithm) start at one end and move to the other, servicing requests along the way. Then reverse

C-SCAN: circular scan instead of reversing at the end, return to start and always scan in the same direction

LOOK: like scan, but return or reverse when no more requests are beyond the current position (stop short of end whenever possible) CLOOK also possible

#### 1.3.6.2 RAID structure

Redundant array of independent disks serves to increase throughput and reliability.

## 1.4 Compilers, linkers, run-time systems

### 4.1 Comparison of direct and indirect representation of objects

Direct: object's name bound at run time to stack storage

Indirect: object's name bound at run time to stack pointer referencing heap storage

Compilation time not affected by either choice.

Time to access object shorter for direct rep.

When storage size for some private components of an object changes, indirect rep. results in fewer recompilations of source modules.

#### 1.4.2 Activation record frame

For a stack-based language, activation records are kept on a stack. (Most languages such as C and Pascal are stack-based) The alternative is a heap.

The following is a sample activation record:

- return value (may be in a register)
- actual parameters
- saved status (registers and program counter)
- temporaries
- local variables
- access link -- to "nonlocal" data (optional)
- control link -- to previous activation record (optional)

As frames are allocated (procedures activated), the stack grows. When procedures return, the stack shrinks and the space is available for the next call. Stack-based allocation doesn't allow for saving local variables after a function return (except for `static` variables). The storage for these variables is reused at the next procedure call. Heap-based allocation is a more general mechanism that uses chunks of dynamically allocated memory for activation records. In this case, if records need to persist beyond a function's return deallocation can be postponed without prohibiting the next function call. Recursive functions need to be stored on a stack because recursion is naturally modeled by a stack.

## 1.5 Distributed systems

### 1.5.1 Semaphores

```
P(S) { // Dutch proberen "to probe or test"  
    while (S <= 0)  
        ; // wait  
    S--;  
}
```

```
V(S) { //Dutch verhogen "to increment"  
    S++;  
}
```

must be executed atomically

sample usage:

```
P(S);  
// critical section  
V(S);
```

### 1.5.2 Reentrant programming

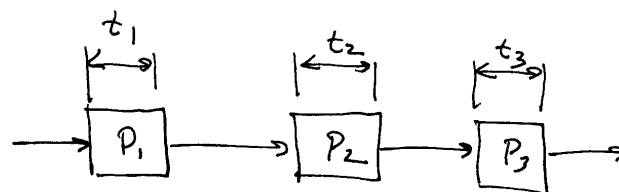
In a multiprogrammed system where the same program is shared by several users, the program must be reentrant. A purely reentrant program is read-only; ie, it does not modify the values of any of its values. A reentrant program is one that several threads can execute concurrently. If values must be modified, their access must be protected in a critical section through semaphores or interlocks.

### 1.5.1 Indempotent functions or operations

A function or operation that has the same effect whether it is executed once or many times.

### 1.5.3 Processor allocation

Suppose a task is performed as 3 sequential processes where the input of one depends on the output of the previous (a pipeline) as shown below:



Each process  $p_i$  takes time  $t_i$ . If a processor is statically allocated to each process, the time between issuing new tasks (repeat / initiation interval) is  $\max(t_i)$ . However, if processors can be dynamically allocated and moved on the fly s.t. any processor can be assigned to any process, then the repeat / initiation interval is  $\text{average}(t_i)$ .

## 2. Theory (40%)

### 2.1 Complexity

$g(n) = O(f(n))$  denotes “ $g(n)$  has order at most  $f(n)$ ” and means that there exist positive constants  $C$  and  $N$  such that  $|g(n)| \leq Cf(n)$  for all  $n > N$ .

$g(n) = \Omega(f(n))$  denotes “ $g(n)$  has order at least  $f(n)$ ” and for this test means that there exist positive constants  $C$  and  $N$  such that  $g(n) \geq Cf(n)$  for all  $n > N$ .

$g(n) = \Theta(f(n))$  denotes “ $g(n)$  has the same order as  $f(n)$ ” and means that there exist positive constants  $C_1$ ,  $C_2$ , and  $N$  such that  $C_1f(n) \leq g(n) \leq C_2f(n)$  for all  $n > N$ .

Common complexities, in order of least to greatest:

Fun.	Name
-----	-----
c	constant
logN	log
log <sup>2</sup> N	log squared
N <sup>1/2</sup>	square root
N	linear
NlogN	nlogn
N <sup>2</sup>	quadratic
N <sup>3</sup>	cubic
N <sup>4</sup>	quartic
2 <sup>N</sup>	exponential

Note that exponential time  $k^N$  dominates only for  $k > 1$ . For  $k < 1$ ,  $k^N$  goes to 0.

In complexity theory, lower bounds are generally shown by proving that a property restricts efficient algorithm design while upper bounds are shown by constructing an algorithm. That is, lower bounds prove that a problem can't be solved in less time; constructing an algorithm cannot show that a faster algorithm cannot be constructed. Rather, some property of the problem is used to show that there is some intrinsic difficulty. On the other hand, upper bounds are generally shown by constructing an algorithm that gives the best upper bound so far. A faster algorithm will lower that upper bound.

Complexities of some classic problems:

Linear programming: polynomial

Graph coloring, traveling salesman, hamiltonian circuits, clique finding, etc. NP-complete (exponential)

Presburger arithmetic: doubly exponential

```
eg: for (i = 0; i < n; ++i){
    for (j = i; j < n ++j){
        ...
    }
}
```

the above fragment is still  $O(n^2)$ , even though the inner loop starts at  $i$  instead of 0  
from a few small samples, it can be seen that the program executes the ... the following number of times:  
 $n + n-1 + n-2 + n-3 + n-4 + \dots + n-n+1$

which is  $n^2 - \sum_{i=0}^{n-1} i = n^2 - (n^2 - n)/2 = n^2/2 + n/2 = O(n^2)$  (high-order terms dominate)

## 2.2 Abstract data types

### 2.2.1 List, stack, Q

Usual examples: infix – postfix conversion, double linked lists, circular lists, circular q's

Binary search on a sorted array requires  $O(\log n)$ . (dictionary lookup). This is only for an array implementation, which allows random access. In a linked list implementation, which only allows sequential access, even if the list is sorted, the time is  $O(n/2)$  on average.

### 2.2.2 Tree

Preliminary defs:

The length of a path from one node to another is the number of edges traversed (not the number of nodes, which is the path length + 1)

The height, depth, or number of levels of a node or complete tree is the path length from the root to the node or to the bottom of the tree. Ie, the root is at level 0 and height, depth, or level is the number of edges

A k-ary tree of height h has at most  $k^h$  leaf nodes.

A k-ary tree of height h has at most  $(k^{h+1} - 1)/(k - 1)$  total nodes (complete tree)

#### 2.2.2.1 Binary search tree

$O(\log n)$  average search time, but worst case is  $O(n)$  (unbalanced tree)

Traversals: preorder, inorder, postorder

#### 2.2.2.2 AVL tree

For any node, the depth difference of the two subtrees is at most 1

Accomplished through a series of single and double rotations

smallest AVL tree of height h has  $S(h)$  nodes as follows:

$S(h) = S(h-1) + S(h-2) + 1$  (similar to Fibonacci #'s)

$S(0) = 1$

$S(1) = 2$

#### 2.2.2.3 Splay tree

On each access of a node, that node is moved up to the root through a series of single and double rotations

Amortized (average) operation is  $O(\log n)$

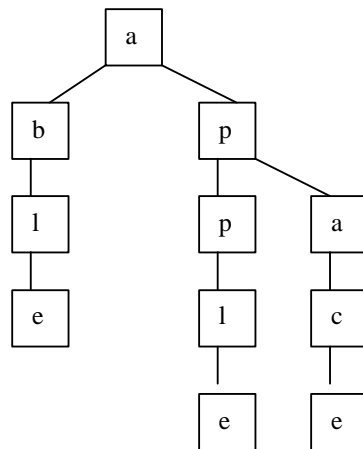
No balance information to maintain as in AVL trees

After a node is accessed several times, successive accesses are cheap because it is at the root or near it.

Splaying not only moves nodes to the root, but also roughly halves the depth of most nodes along the access path.

#### 2.2.2.4 Digital search tree

A tree that stores strings internally so there is no need for extra leaf nodes to store the strings. Also called a trie or directed acyclic word graph (DAWG). Any parent of a node is a common prefix to that node and all its children. The number of accesses to locate a node is  $O(\text{word length})$ . Eg: {able, apple, apace} stored in a DST:

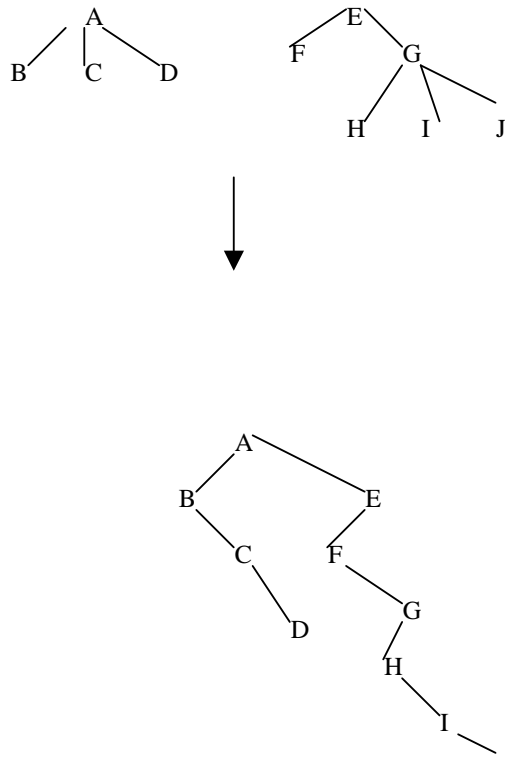


#### 2.2.2.5 Forest of general trees

Any forest of general trees can be converted into a binary tree as follows:

1. Root of binary tree = root of first general tree in forest
2. Left node in binary tree = child of parent in general tree forest
3. Right node in binary tree = brother of left node in general tree forest

eg:





### 2.2.3 Hashing

Table size must be prime

Choice of hash function critical

Load factor = # used entries / table size

Collision resolution

Separate chaining

Each bucket is a linked list

Open addressing

In case of a collision, an alternative bucket is attempted, continuing until an empty one is found

Linear probing – try buckets in succession w/ wrap-around

Quadratic probing – try buckets in quadratic steps w/ wrap-around

Double hashing – apply a second hash function to the collided bucket

In all open addressing schemes, the load factor should be  $\leq .5$

Good uses of hash tables:

- Counting distinct values
- Dynamic dictionary
- Symbol table lookup
- Finding intersections of keys

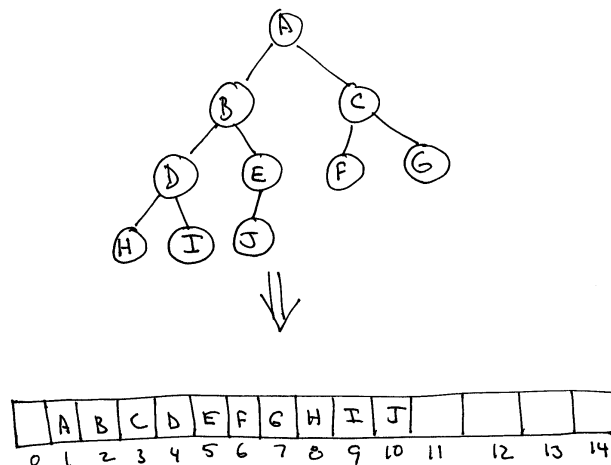


*Not range searching (hashing functions place consecutive items in non-consecutive locations)*

### 2.2.4 Heap (priority Q)

Binary tree that is completely filled except for bottom level

Because of this restricted structure, an array implementation is used:



For any array position  $i$ , the left child is in position  $2i$ , the right child is in the next position  $(2i+1)$ , and parent is in  $\text{floor}(i/2)$

Only 2 operations are allowed: insert and delete min. (root is the min.)

Both are  $O(\log n)$

## 2.2.5 Sorting

### 2.2.5.1 Insertion sort, bubble sort, selection sort

$O(n^2)$  average for entire class of sorting algorithms that perform exchanges of adjacent elements.  
Outperforms other algorithms on small arrays ( $\leq 20$  elements)  
Most sensitive to input variations (sorted input  $O(n)$ ; reverse input  $O(n^2)$ )

### 2.2.5.2 Shellsort (named after Donald Shell)

Diminishing increment sort: elements positioned at diminishing increments are sorted on each pass.  
 $O(n^2)$  worst case, but average performance is better (sub-quadratic)  
In an array that is  $k$ -ordered, items spaced  $k$  locations apart are ordered.  
In a 2-ordered array of  $2N$  items, an item can be at most  $N$  locations away from its final position.  
In a both 2-ordered and 3-ordered array of  $N$  items, an item can be at most 1 location away from final.

### 2.2.5.3 Heapsort

Construct a heap and output elements in order  
 $O(n \log n)$

### 2.2.5.4 Mergesort

Recursively merge the first half of the list w/ the second half (both sublists must already be sorted – apply mergesort recursively until lists are single elements)  
 $O(n \log n)$  always – most consistent performance, independent of input

### 2.2.5.5 Quicksort

Quickest sort in practice and method of choice for internal sorting; tight inner loop produces small proportionality constant for time complexity.  
Let  $S$  be the set to sort  
If  $|S| == 0$  or 1, return  
Pick pivot  $v$  in  $S$   
Partition  $S$  in 2 disjoint subsets, those elements smaller than  $v$  and those larger  
Recursively quicksort those subsets  
 $O(n \log n)$  average  
 $O(n^2)$  worst case, but the likelihood of that can be made very small by choice of pivot and careful handling of special cases.  
Performs poorly on small arrays ( $\leq 20$  elements); use insertion sort instead

### 2.2.5.6 Special case algorithms

If size and range of input is finite, known, & manageable size, build a histogram of buckets and then output the list in order from the histogram.  
 $O(n)$  beats  $O(n \log n)$  because not doing sequential comparisons. Useful in many cases such as small integers.

### 2.2.5.7 Summary

All sorting algorithms based on comparisons require  $\Omega(n \log n)$  time in the worst case (lower bound on worst case performance; cannot do better)

Algorithm least affected by input variations is mergesort,  $O(n \log n)$  always, but not used for internal sorting due to space and copy requirements

Algorithm most affected by input is insertion sort, bubble sort, selection sort,  $O(n)$  for sorted input,  $O(n^2)$  for reversed input.

Shellsort highly dependent on choice of increments

Quicksort best internal sort because  $O(n \log n)$  on average, and worst case of  $O(n^2)$  can be made highly unlikely

Time complexity independent of data structure. eg: mergesort is  $O(n \log n)$  for single, double linked list same as for array.

### 2.2.6 Disjoint set

Used for representing equivalence relations (reflexive, symmetric, transitive), ie, partitions of a set

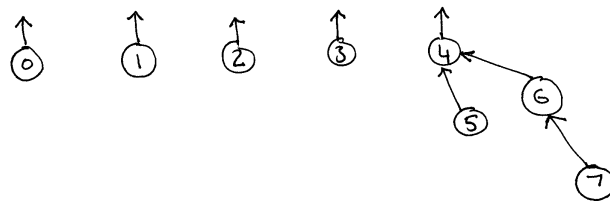
2 operations: union and find

find: finds the equivalence class that an element belongs to,  $O(n)$ , linear time

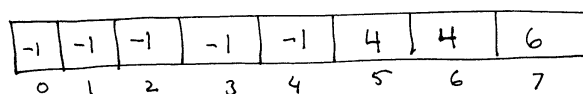
union: merge two equivalence classes into one,  $O(1)$ , constant time, assuming the two roots of the individual trees have already been found

Explicit (theoretical) and implicit(actual) representations:

explicit: forest



implicit: array



### 2.2.7 Graph

#### Defns:

If it is possible to establish a path from any vertex to any other vertex of a graph, the graph is said to be **connected** (different than **fully connected** or **complete**, where a path exists from every vertex to every other) A graph that is not connected may still be partitionable into connected components. Eg: cities and roads connecting them on Hawaii islands from Aho et al, Foundations of CS, p. 457) Connectedness is typically applied to undirected graphs. Connected components partition a graph into equivalence classes.

A **planar graph** is one which can be drawn in the plane without any two edges intersecting.

A **Eulerian path** in a (undirected) graph is a path that uses each edge precisely once (don't end at starting point). If such a path exists, the graph is called *traversable*. An **Eulerian cycle** (end at starting point) is a cycle which uses each edge precisely once. For a Eulerian **cycle(not just path)** to exist, the graph must be connected and all vertices must have even degree, because for any vertex it must be possible to come in and go out.

A **Hamiltonian path** in a graph is a path that visits each *vertex* once and only once; and a **Hamiltonian cycle** is a cycle which visits each vertex once and only once.

An **independent set** in a graph is a set of pairwise nonadjacent vertices.

A **clique** in a graph is a set of pairwise adjacent vertices.

A **bipartite graph** is any graph whose vertices can be divided into two sets, such that there are no edges between vertices of the same set. A graph can be proved bipartite if there do not exist any circuits of odd length.

A **k-partite graph** or **k-colorable graph** is a graph whose vertices can be partitioned into  $k$  disjoint subsets such that there are no edges between vertices in the same subset. A 2-partite graph is the same as a bipartite graph.

[Notation: in a bipartite graph  $K(V)$ ,  $V$  is the number of vertices in the graph, and  $V/2$  are in each partition. In  $K(V_1, V_2)$ ,  $V_1$  is the number of vertices in one partition of the bipartite graph and  $V_2$  is the number of vertices in the other partition. – TP ]

**Maximal independent set:** largest subset of graph vertices s.t. no edges connect them

**Maximal clique:** largest subset of graph vertices s.t. all are completely connected to each other.

**Graph coloring:** color each maximal independent subset a different color (same problem as finding the maximal independent subsets)

**Path length** is the number of edges, not vertices (as with trees)

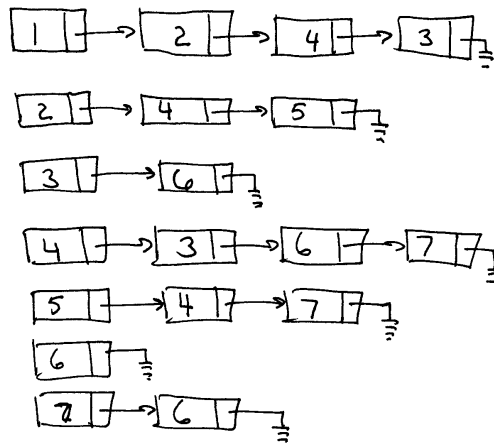
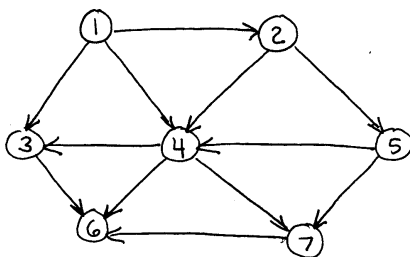
#### Various characteristics

- In a finite directed acyclic graph with at least one edge, there must exist at least one vertex with no incoming edge (in-degree 0) and at least one vertex with not outgoing edge. (out-degree 0)
- In a connected, undirected graph, the sum of the degrees of all vertices is even, and  $|E| \geq |V| - 1$  (try drawing connected graphs of some small numbers of vertices, eg 2, 3, 4)

#### Representations

Adjacency matrix(dense graphs) space complexity  $O(V^2)$ ,  $V$ =#vertices

Adjacency list (sparse graphs) space complexity  $O(E + V)$ ,  $E$ =#edges,  $V$ =#vertices



### Adjacency list (sparse matrix)

A sparse matrix can be stored as a doubly linked list (vertical and horizontally linked) according to rows and columns. A head link exists for each row and for each column, and 2 links exist for each nonzero entry (vertical and horizontal link).

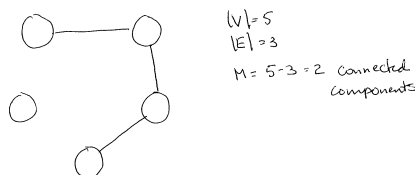
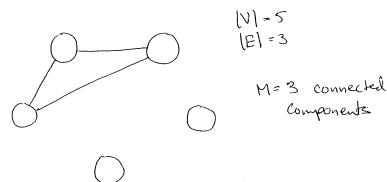
Total number of links required:

$$\# \text{ rows} + \# \text{ columns} + 2 * \text{ number nonzero entries}$$

### Connected components

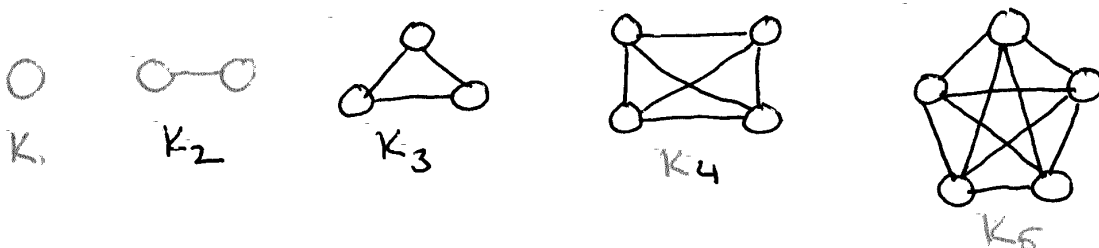
Eg: in a graph with 5 nodes and 3 edges, what is the max and min number of connected components that can be found, assuming no double edges and no self-loops?

For max. number, connect the fewest nodes possible with all edges, leaving all remaining as single components. For min. number, connect in series  $|E| + 1$  nodes, forming that component and any remaining single-node components ( $m =$



### Complete graph

Every node is connected to every other directly. The first five undirected complete graphs are shown below. Note that the even/odd degree of the vertices is opposite that of the number of vertices. (Eulerian cycles cannot be formed in  $K_2$ ,  $K_4$ ,  $K_6$ ,... because the vertex degrees are odd).



### Shortest path (British museum search)

Find all possible paths using either depth first or breadth first search and select the best from all of them. Do not stop until all have been found. This is the most expensive, combinatorially explosive algorithm, but guaranteed to be optimal.

### Depth-first search

For directed and undirected graphs

Construct a depth first search tree (actually a forest of them) from the graph as follows:

1. Pick a node and start a tree with the node as its root
2. Recursively visit its thus far unvisited successors, marking nodes as visited so as to not visit them twice and end up in an infinite loop.
3. In the tree, make visited successors children of the previous node, in left to right order.
4. Once no more progress can be made, if any graph nodes are still unvisited, start another tree for those, thus resulting in a forest.

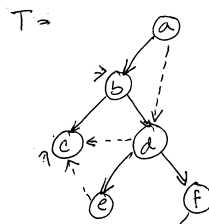
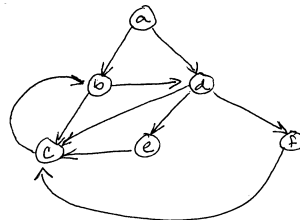
Complexity  $O(m)$ , where  $m$  is the larger of the number of graph nodes and graph edges.

Examples of directed and undirected graphs and their corresponding search trees are below. Dotted arcs in the search trees are original arcs in the graphs that were not used in the depth first search.

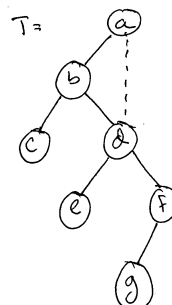
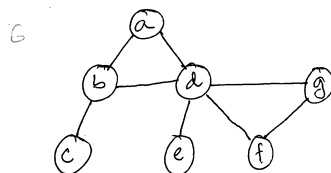
Once the search tree is constructed, various properties of it can be examined. Eg, in the undirected case: preorder and postorder traversals are: Pre: a,b,c,d,e,f,g      Post: c,e,g,f,d,b,a

If  $(u,v)$  is an arc in  $G$  but not in  $T$  s.t.  $\text{pre}(u) < \text{pre}(v)$ , for example  $(d,g)$ , then

- $u$  is an ancestor of  $v$  in  $T$
- $\text{post}(u) > \text{post}(v)$  (opposite  $\text{pre}(u) < \text{pre}(v)$ )



Undirected



### Shortest path (Dijkstra's algorithm)

For both directed and undirected graphs, weighted or unweighted. Finds shortest path lengths to all other nodes from a single source.

1. Mark the starting vertex known and path length 0
2. From all known vertices, compute total path length to all adjacent vertices (path lengths for unknown vertices may change from previous iteration)
3. Choose the smallest and mark it as known, and set its total path length permanently.
4. Repeat from 3 until all vertices are known.

$O(mn \log n)$ , where  $n$  is the number of nodes,  $m$  is the max of the number of nodes and number of edges.  $m$  can be up to  $n^2$  in the worst case.

### All shortest paths (Floyd-Warshall algorithm)

Set up an adjacency matrix of the graph with edge weights being the matrix entries. Use 0 for same vertex and infinity if not directly connected. Then iterate over pivot nodes (intermediate nodes) and reduce entries to minimum values whenever a path through a pivot node is less than current cost.

$O(n^3)$  (no better than repeated iterations of Dijkstra, but adjacency matrix data structure is more compact)

### Maxflow

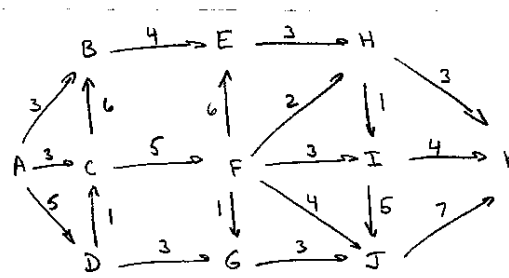


Greedy algorithm does not produce optimal solution

1. Maintain 2 graphs besides the original: flow graph and residual graph
2. Flow graph initially has all 0 flows, residual graph initially is same as original
3. Choose a path from source to sink non-deterministically and find the minimum capacity along the way in the residual graph
4. Add that amount of flow to all edges along the route in the flow graph and subtract from all edges along the route in the residual graph. Add augmenting edges in the reverse direction along the route in the residual graph with that amount of flow in each (so the algorithm can backtrack if necessary)
5. Repeat from 3 until no path from source to sink exists.
6. The flow graph contains the maximum flow leaving the source and entering the sink.

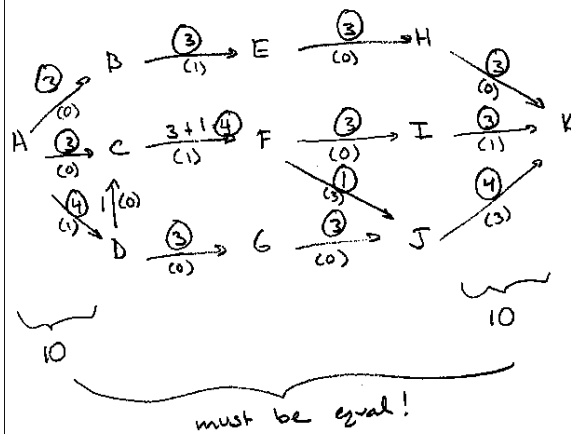
Sometimes an intuitive approach can be done manually (test problems). Examine the flows leaving the source and the flows entering the sink. The minimum total of these provide an upper bound on the max flow, from which some answer may be eliminated. Then look for clear paths that use of most of the flow (usually easy to spot), and direct as much of the flow as simply as possible, keeping track of what remaining capacities edges have (like a residual graph, but without the augmented edges)

Eg: (max flow from A to K)



max leaving A = 11  
max entering K = 14 }  $\max \leq 11$

circled values = flows  
parenthesized values = residuals



### Minimum spanning tree MST

Prim's, Kruskal's algorithms for undirected graphs. Directed graphs are harder. On exam problems, assume the graph is undirected for computing MST, even if the original is directed.

Kruskal's algorithm:

1. Start w/ a forest of all nodes unconnected.
2. Repeatedly add lowest cost edges until a tree results
3. Do not add edges that produce a cycle.

### Time complexity (undirected graph algorithms)

Polynomial time (P):

- Finding the shortest cycle

Not polynomial (NP):

- Finding the longest simple cycle
- Finding **all** spanning trees
- Finding the largest clique
- Finding a node coloring with the minimum number of colors



### 2.2.8 Comparison of data structures

#### Worst-case complexity

ADT	O(insert)	O(delete)	O(find)
Sorted list	n	n	n
Unsorted list	1	n	n
Heap	log(n)	NA	NA
AVL tree	log(n)	log(n)	log(n)
Hash table w/ linked lists	n	n	n

## 2.3 Algorithms

### 2.3.1 Greedy

Perform what appears to be the best option at each stage.

### 2.3.2 Divide and conquer

Divide: smaller problems are solved recursively, must form 2 or more smaller problems at each stage; a single simpler problem does not qualify

Conquer: combine solutions into final solution

#### 2.3.2.1 Recursion

Recursive algorithms that can be converted to iterative ones (using a fixed number of for loops based on the input) are called primitive recursive. A polynomial or exponential time complexity can be computed. Non-primitive recursive algorithms are those that can only be computed recursively (or using a combination of for and while loops) such as Ackermann's function. No known time complexity exists. The Ackermann function is computable, but not primitively recursive. Every computable function can be programmed with for and while loops, but only primitively recursive functions can be programmed with only for loops. Examples of primitive recursive functions are power, greatest common divisor, and nth prime.

Direct recursion: a function calls itself

Indirect recursion: a function calls something else, which may call something else, but eventually the original function is called before returning.

#### 2.3.2.2 Recurrence relations

Recurrence relations are used to compute the run time of recursive functions. Three methods exist for solving a recurrence relation:

- Trying available answers
- Multiple substitution
- Guessing a solution from known forms

Trying available answers:

Sometimes testing for a correct answer from a list of choices can be the fastest method and amounts to setting up a small test upper limit and plugging the numbers in.

$$\text{Eg: } \sum_{k=2}^n (k(k-1))$$

or alternately,  $F(j) = F(j-1) + j(j-1)$  for  $j = 2 \dots n$  with  $F(2) = 2$

can be found to be equal to  $(n-1)n(n+1)/3$  as a given choice and other choices eliminated simply by testing  $n=3$ . If multiple choices were correct, try  $n = 4$ , etc. until all others are eliminated.

Multiple substitutions:

Eg: given,

$$T(1) = O(1)$$

$$T(n) = O(1) + T(n-1) \text{ for } n > 1$$

Rewrite the big-O notation using constants:

$$T(1) = a$$

$$T(n) = b + T(n-1)$$

Then substitute repeatedly until a pattern emerges:

$$T(2) = b + a$$

$$T(3) = b + b + a = 2b + a$$

$$T(4) = b + 2b + a = 3b + a$$

...

$$T(n) = a + b(n-1)$$

Then remove constants and switch back to big-O notation:

$$T(n) = O(n) \quad [\text{linear}]$$

Known forms:

Often a known pattern will occur that can be matched up to the following:

## Summary of Solutions

In the table below, we list the solutions to some of the most common recurrence relations, including some we have not covered in this section. In each case, we assume that the basis equation is  $T(1) = a$  and that  $k \geq 0$ .

INDUCTIVE EQUATION	$T(n)$
$T(n) = T(n-1) + bn^k$	$O(n^{k+1})$
$T(n) = cT(n-1) + bn^k$ , for $c > 1$	$O(c^n)$
$T(n) = cT(n/d) + bn^k$ , for $c > d^k$	$O(n^{\log_d c})$
$T(n) = cT(n/d) + bn^k$ , for $c < d^k$	$O(n^k)$
$T(n) = cT(n/d) + bn^k$ , for $c = d^k$	$O(n^k \log n)$

All the above also hold if  $bn^k$  is replaced by any  $k$ th-degree polynomial.

Eg:

$$T(n) = 8T(n/2) + qn \quad \text{for } n > 1$$

$$T(n) = p \quad \text{for } n = 1$$

The time complexity is  $O(n^3)$  because from the above table, we know it is one of the last 3 forms, depending on the values of the constants. Matching up constants,  $c = 8$ ,  $d = 2$ ,  $k = 1$

Since  $c > d^k$  (ie,  $8 > 2$ ), it is  $O(n^{\log_d c})$  (the third choice in the table)

Since  $\log_d c = \log_2 8 = 3$ , the time complexity is  $O(n^3)$ .

Some common ones to memorize (based on above table)

Form	complexity
$T(n) = T(n-1) + O(1)$	$O(n)$
$T(n) = T(n-1) + O(n)$	$O(n^2)$
$T(n) = T(n/2) + O(1)$	$O(\log n)$
$T(n) = 2T(n/2) + O(n)$	$O(n \log n)$
$T(n) = kT(n-1)$	$O(k^n)$ for $k > 1$ , eg: $k = 2$

Master theorem:

$$\text{let } T(n) = \begin{cases} c & \text{for } n < d \\ aT(n/b) + f(n) & \text{for } n \geq d \end{cases}$$

case 1: if  $f(n)$  is  $O(n^{\log_b a - \epsilon})$  for some  $\epsilon > 0$  then  $T(n)$  is  $\Theta(n^{\log_b a})$

case 2: if  $f(n)$  is  $\Theta(n^{\log_b a} \log^k n)$  for some  $k \geq 0$  then  $T(n)$  is  $\Theta(n^{\log_b a} \log^{k+1} n)$

case 3: if  $f(n)$  is  $\Omega(n^{\log_b a + \epsilon})$  for some  $\epsilon > 0$  then  $T(n)$  is  $\Theta(f(n))$  provided  $\exists$  some  $\delta < 1$  s.t.  
 $af(n/b) \leq \delta f(n)$

### 2.3.3 Branch and bound

Search technique where we keep expanding nodes with least accumulated cost thus far.

### 2.3.4 Dynamic programming

In cases where recursion re-computes previously computed values (eg Fibonacci #'s), those values are stored dynamically in a table and used in a linear program instead.

### 2.3.5 Easy vs hard

#### Reducibility

A problem is often reduced to a simpler problem. Eg, the problem of measuring the area of a rectangle reduces to measuring the height and width. If A is reducible to B, solving A cannot be harder than solving B because B gives a solution to A.

Assume A reduces to B.

B is decidable implies A is decidable.

A is undecidable implies B is undecidable.

If A is polynomial-time reducible to B, then if B can be computed in P time, so can A. It is the same as reducibility, except with efficiency taken into account.

P (polynomial time)

- Finding the inverse of a matrix, including the *exact* inverse of a matrix w/ rational number entries.
- Finding a path in a directed graph between two nodes, shortest path in a graph (Dijkstra) and all shortest paths (Floyd-Warshall)
- Determining if 2 numbers are relatively prime
- Deciding any context free language

NP and NP-complete (not polynomial, actually nondeterministic polynomial) NP is the class of problems that have polynomial verifiers, ie, once a solution is given, it can be verified polynomially

NP-complete is a subset of NP s.t. every problem in NP can be polynomially reduced to an NP-complete problem. If one NP-complete problem can be found to be in P, then all NP problems are in P. NP-complete are the hardest of all NP problems. To prove a problem R is NP or NP-complete, polynomially reduce some other NP-complete problem to it.

- Given a combinational circuit, find a string of inputs that would produce a given output string or determine that it can't (similar to satisfiability) (NP-complete)
- Graph coloring(NP-complete)
- Clique finding (NP-complete)
- Hamiltonian cycle or path finding (NP-complete)
- Traveling salesman (given a complete graph, is there a simple cycle that visits all vertices and has a total cost  $\leq$  some k) (NP-complete)
- Longest path(NP-complete)
- Bin packing(NP-complete)
- Knapsack(NP-complete)
- Tautology problem, satisfiability problem (NP-complete)
- Determining whether a number is composite or prime(NP)
- Finding smallest vertex cover (NP-complete)

## 2.4 Languages and automata

### 2.4.1 Taxonomy

Automata parse languages, grammars generate them

Chomsky hierarchy ([http://www.fact-index.com/c/ch/chomsky\\_hierarchy.html](http://www.fact-index.com/c/ch/chomsky_hierarchy.html))

- Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. The language that is recognized by a Turing machine is defined as all the strings on which it halts. These languages are also known as the recursively enumerable languages, or Turing-recognizable. Note that this is different from the recursive languages which can be *decided* by an always halting Turing machine. **No restrictions on rules**

Egs:

$\{0^{2^n} \mid n \geq 0\}$   
 $\{a^i b^j c^k \mid i, j, k \geq 1 \text{ and } i * j = k\}$   
 $\{x_1 \# x_2 \# x_3 \# \dots \mid \text{all the } x_i \text{'s are unique}\}$   
 $\{ww \mid w \in \{0,1\}^*\}$

- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form  $\alpha A \beta \rightarrow \alpha \gamma \beta$  with  $A$  a nonterminal and  $\alpha, \beta$  and  $\gamma$  strings of terminals and nonterminals. The strings  $\alpha$  and  $\beta$  may be empty, but  $\gamma$  must be nonempty. **For any rule, the left side must be less than or equal to the length of the right side, in terms of number of words.** The rule  $S \rightarrow \epsilon$  is allowed if  $S$  does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic Turing machine whose tape is bounded by a constant times the length of the input.

Eg's:

$0^n 1^n 2^n$  is context sensitive

- Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form  $A \rightarrow \gamma$  with  $A$  a nonterminal and  $\gamma$  a string of terminals and nonterminals. **For any rule, the left side must be a single non-terminal** These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton. Context free languages are the theoretical basis for the syntax of most programming languages.

Eg's:

$0^n 1^n$

$0^n 1^n 2^k 3^k$  (a concatenation of two  $0^n 1^n$  context frees)

$\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

$\{0^i 1^j 0^k \mid i + k = j\}$

$\{0^i 1^j 0^k \mid i + j = k\}$

$\{0^i 1^j \mid i \neq j\}$  (failure state of  $0^i 1^i$ )

$\{0^i 1^j \mid i \leq j \leq 2i\}$

$\{ww^R \mid w \in \{0,1\}^*\}$  (a string concatenated with its reverse)

$a^n b^n c^n$  is not context free

$\{a^i b^j c^k \mid 0 \leq i \leq j \leq k\}$  is not context free

$\{0^i 1^j \mid i = j^2\}$  is not context free



- Type-3 grammars (regular grammars) generate the regular languages. Such a grammar restricts its rules to a single nonterminal on the left-hand side and a right-hand side consisting of a single terminal, possibly followed by a single nonterminal. The rule  $S \rightarrow \epsilon$  is also here allowed if  $S$  does not appear on the right side of any rule. These languages are exactly all languages that can be decided by a finite state automaton. Additionally, this family of formal languages can be obtained by regular expressions. Regular languages are commonly used to define search patterns and the lexical structure of programming languages. **Left side single nonterminal, right side one or more symbols including at most a single non-terminal at extreme right.**



Eg's: The reverse of an RL is still RL (Reverse =  $\{W \mid W \text{ is the reverse of } V, \text{ where } V \in L\}$ )  
The set of prefixes of an RL is RL

Grammar	languages	automaton	production rules
Type 0	recursively enumerable	Turing machine	no restrictions
Type 1	context sensitive	linear bounded non determ. Turing machine	$\alpha A \beta \rightarrow \alpha \gamma \beta$
Type 2	context free	non-determ. Pushdown automaton	$A \rightarrow \gamma$
Type 3	regular	deterministic finite state automaton	$A \rightarrow aB, A \rightarrow a$

$RL \subset CF \subset CS \subset RE(\text{regular, context-free, context-sensitive, recursively enumerable})$

If a language  $L$  is CF, then it is also CS because  $CF \subseteq CS$ .  $\bar{L}$  will be CS, but not CF.

Determining all possible decompositions of sequential machines amounts to finding all closed partitions and covers, which requires exponential time.

A non-deterministic finite automata with  $n$  states can always be converted to a deterministic automata with at most  $2^n$  states.



An interesting note is that if a machine (finite automata, deterministic or non-deterministic, etc. can recognize a language, the same type of machine can recognize the complement of the language by switching accept and reverse states.



Eg: if  $L$  can be recognized by an NDFA, so can  $L'$   
If  $L$  can be recognized in polynomial time, so can  $L'$   
If  $L$  is decidable, so is  $L'$   
...

#### 2.4.2 Regular and context free grammars

A notation for defining CF languages (type 2), as well as many programming languages. Also can be used to define regular languages (type 3), but more often used for CF, since regular expressions are already a compact notation for defining regular languages.

Eg. of a CF grammar for programming languages is operator precedence.

Eg:

$\langle \text{word} \rangle := \langle \text{letter} \rangle \mid \langle \text{letter} \rangle \langle \text{pairlet} \rangle \mid \langle \text{letter} \rangle \langle \text{pairedig} \rangle$   
 $\langle \text{pairlet} \rangle := \langle \text{letter} \rangle \langle \text{letter} \rangle \mid \langle \text{pairlet} \rangle \langle \text{letter} \rangle \langle \text{letter} \rangle$   
 $\langle \text{pairedig} \rangle := \langle \text{digit} \rangle \langle \text{digit} \rangle \mid \langle \text{pairedig} \rangle \langle \text{digit} \rangle \langle \text{digit} \rangle$   
 $\langle \text{letter} \rangle := a|b|\dots|y|z$   
 $\langle \text{digit} \rangle := 0|1|\dots|8|9$

The string "words" can be produced from  $\langle \text{word} \rangle$  as follows:

$\langle \text{word} \rangle \rightarrow \langle \text{letter} \rangle \langle \text{pairlet} \rangle \rightarrow w \langle \text{pairlet} \rangle \langle \text{letter} \rangle \langle \text{letter} \rangle \rightarrow w \langle \text{letter} \rangle \langle \text{letter} \rangle \text{ds} \rightarrow \text{words}$

The string "c22" can be produced from  $\langle \text{word} \rangle$  by:

$\langle \text{word} \rangle \rightarrow \langle \text{letter} \rangle \langle \text{pairdig} \rangle \rightarrow c \langle \text{digit} \rangle \langle \text{digit} \rangle \rightarrow c22$

However, the string "word" cannot be produced from the above grammar:

$\langle \text{word} \rangle \rightarrow \langle \text{letter} \rangle \langle \text{pairlet} \rangle \rightarrow w ?$  (this grammar can produce only odd numbers of letters)

Eg: find the RE corresponding to the language produced by:

$\langle v \rangle := a \langle w \rangle$

$\langle w \rangle := bb \langle w \rangle \mid c$

v

aw

ac

abbw

abbc

abbbbw

abbbbc

abbbbbb

abbbbbb

abbbbbbbw

abbbbbbbc

...

RE:  $a(bb)^*c$

Eg: find the RE corresponding to:

$\langle S \rangle := \langle A \rangle x \mid \langle B \rangle y$  (1)

$\langle A \rangle := \langle B \rangle y \mid \langle C \rangle w$  (2)

$\langle B \rangle := x \mid \langle B \rangle w$  (3)

$\langle C \rangle := y$  (4)

Solution:

$Ax + By$  (from (1), converting to algebraic form)


B can be replaced by  $xw^*$  (realizing how (3) works) (5)

$Ax + xw^*y$

A can be replaced by  $By + Cw$  from (2), or  $xw^*y + yw$  from 5,4

$(xw^*y + yw)x + xw^*y$

$xw^*yx + ywx + xw^*y$  (answer)

 An operator grammar is a grammar that has no adjacent non-terminals in the right hand side and no right and side is  $\epsilon$ .

Eg: convert the following to an operator grammar:

$S \rightarrow SAS \mid a$

$A \rightarrow bSb \mid b$

The SAS in the first rule makes it not an operator grammar as written. However, by substituting for A using the second rule, the result is  $S \rightarrow SbSbS \mid SbS \mid a$

Tricks to constructing grammars:

1.  $0^k 1^k$  is generated from  $\langle S \rangle := 0 \langle S \rangle 1 \mid \epsilon$

2. if a CFG can be broken into a union of simpler CFGs, generate each of the simple ones as

$\langle S1 \rangle := \dots$

$\langle S2 \rangle := \dots$

...

and then combine them all as  $\langle S \rangle := \langle S1 \rangle \mid \langle S2 \rangle \mid \dots$

### 2.4.3 DFA's and regular languages and grammars (ref: Kolman & Busby, 1984, sect. 7.3)

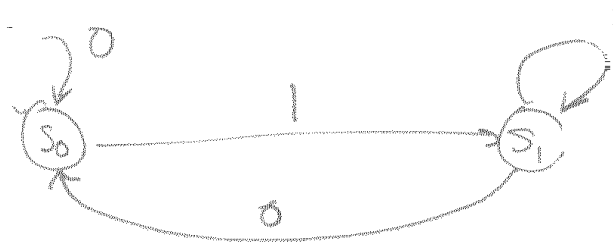
Regular expression operator precedence:

Parentheses, otherwise star first, then concatenation, then union.

Two key operations can be performed:

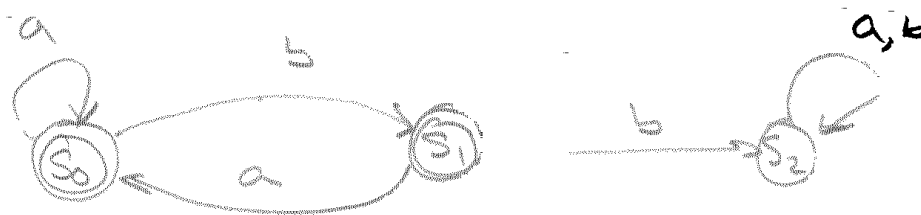
1. given a DFA determine the language (represented by either an English description, a regular expression, production rules, or BNF grammar)
2. given a language requirement, determine the DFA

eg: find the RE corresponding to the language generated by:



RE:  $(0 \vee 1)^*$

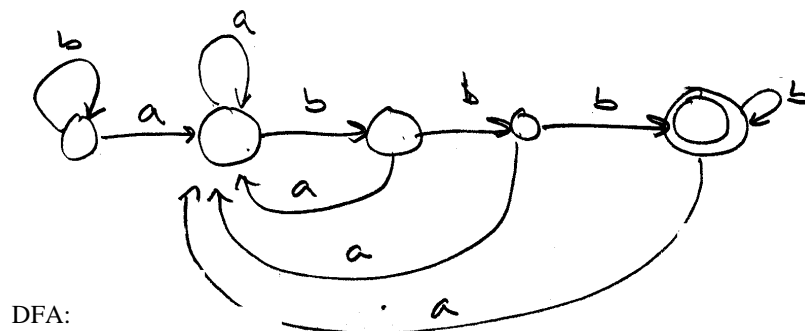
Eg: find the BNF grammar corresponding to the language generated by:



BNF:

$$\begin{aligned} \langle s2 \rangle &:= a \langle s2 \rangle \mid b \langle s2 \rangle \\ \langle s1 \rangle &:= b \langle s2 \rangle \mid a \langle s0 \rangle \mid a \\ \langle s0 \rangle &:= a \mid b \mid a \langle s0 \rangle \mid b \langle s1 \rangle \end{aligned}$$

eg: construct a DFA that accepts a language with inputs  $I = \{a, b\}$  s.t. the accepted strings contain  $ab$  and end  $bbb$ .



DFA:



### 2.4.3 Parsing

A derivation from a grammar can be modeled as a tree called a derivation tree as follows:

(Derivation trees exist only for context free and regular grammars (type 2 and 3))

Eg:

$\langle \text{sentence} \rangle := \langle \text{noun} \rangle \langle \text{verbphrase} \rangle$

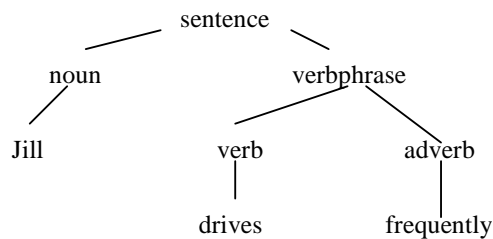
$\langle \text{verbphrase} \rangle := \langle \text{verb} \rangle \langle \text{adverb} \rangle$

$\langle \text{noun} \rangle := \text{John} \mid \text{Jill}$

$\langle \text{verb} \rangle := \text{drives} \mid \text{jogs}$

$\langle \text{adverb} \rangle := \text{carelessly} \mid \text{rapidly} \mid \text{frequently}$

The derivation tree for 'Jill drives frequently' is



The leaves are the final result, from left to right. If the tree is unique, then the derivation is unambiguous. Languages with non-unique derivation trees are ambiguous. A tree may be constructed in non-unique orders of substitutions, but the resulting trees should be unique for a non-ambiguous language.

### Ambiguous grammars

Eg:  $\langle \text{expr} \rangle := \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \langle \text{expr} \rangle * \langle \text{expr} \rangle \mid a$

is ambiguous because  $a+a*a$  can be derived 2 ways (+ first or \* first)

Some ambiguous grammars can be re-written unambiguously, but others cannot, called **inherently ambiguous**.

Parsing can be done bottom-up or top-down. Any grammar that is unambiguous can be parsed either way.

Another, more complex ambiguous example:

77. Which of the following sets of productions determine(s) an ambiguous context-free grammar?

- I.  $\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle$   
 $\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$   
 $\langle \text{stmt} \rangle \rightarrow \text{skip}$   
 $\langle \text{cond} \rangle \rightarrow \text{true} \mid \text{false}$
- II.  $\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle \text{ fi}$   
 $\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \text{ fi}$   
 $\langle \text{stmt} \rangle \rightarrow \text{skip}$   
 $\langle \text{cond} \rangle \rightarrow \text{true} \mid \text{false}$
- III.  $\langle \text{stmt} \rangle \rightarrow \text{if } \langle \text{cond} \rangle \text{ then } \langle \text{stmt} \rangle \langle \text{tail} \rangle$   
 $\langle \text{stmt} \rangle \rightarrow \text{skip}$   
 $\langle \text{cond} \rangle \rightarrow \text{true} \mid \text{false}$   
 $\langle \text{tail} \rangle \rightarrow \epsilon$   
 $\langle \text{tail} \rangle \rightarrow \text{else } \langle \text{stmt} \rangle$

First, note that the <tail> construction in III does nothing, and that III is the same as I. I is ambiguous (and III), but II is not unambiguous. Why?

Consider the case of the dangling else:

```

    if E1 then
      if E2 then
        E3
      else E4

```

or alternatively

```

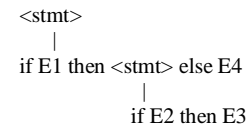
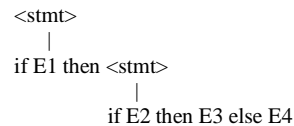
    if E1 then
      if E2 then
        E3
      else E4

```

of course the white space is not parsed, so both are identical to the parser, and the ambiguity results from the 'else' being possible to be matched with either 'if' (that is why programming languages have rules about the else belonging to the closest unmatched if)

The input string without white space is: **if E1 then if E2 then E3 else E4**

The two parse trees are:



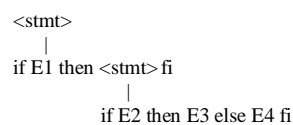
For II, one version is

```

    if E1 then
      if E2 then
        E3
      else
        E4
    fi

```

or without white space: **if E1 then if E2 then E3 else E4 fi fi**  
and the unambiguous parse tree is:



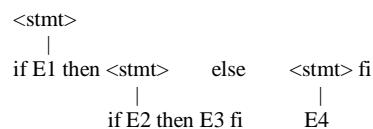
and the 2<sup>nd</sup> version is

```

    if E1 then
      if E2 then
        E3
      fi
    else
      E4
    fi

```

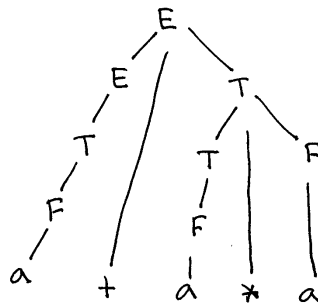
or without white space: **if E1 then if E2 then E3 fi else E4 fi**  
and the unambiguous parse tree is:



#### 2.4.4.1 Shift-reduce (bottom-up) parser

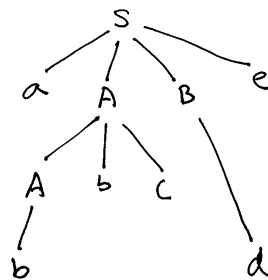
Parsing occurs starting at the first symbol (left side) of the expression and attempts to match the lowest rules first. The parse tree is constructed from the leaves and works upwards towards the root. We think of the process as reducing a substring of the remaining input with some left hand side of some rule. 3 examples follow. The parse tree is shown to help track the progress and it is recommended to draw a parse tree first. Also, the production rules are numbered and the number of the rule used is given with each reduction. Evaluation order: expand each path from bottom up as far as it will go before starting new path. Choose paths from left to right.

- 1)  $E \rightarrow E + T$
- 2)  $E \rightarrow T$
- 3)  $T \rightarrow T * F$
- 4)  $T \rightarrow F$
- 5)  $F \rightarrow (E)$
- 6)  $F \rightarrow a$



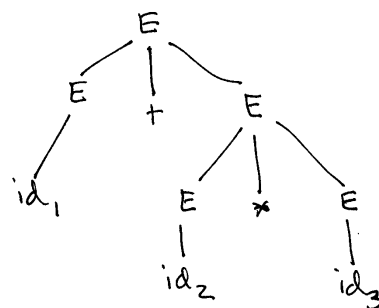
$a + a * a$   
 $F + a * a$  6)  
 $T + a * a$  4)  
 $E + a * a$  2)  
 $E + F * a$  6)  
 $E + T * a$  4)  
 $E + T * F$  6)  
 $E + T$  3)  
 $E$  1)

- 1)  $S \rightarrow aABe$
- 2)  $A \rightarrow Abc$
- 3)  $A \rightarrow b$
- 4)  $B \rightarrow d$



$abbcde$   
 $aAbcde$  3)  
 $aAde$  2)  
 $aABe$  4)  
 $S$  1)

- 1)  $E \rightarrow E + E$
- 2)  $E \rightarrow E * E$
- 3)  $E \rightarrow (E)$
- 4)  $E \rightarrow id$



$id_1 + id_2 * id_3$   
 $E + id_2 * id_3$  4)  
 $E + E * id_3$  4)  
 $E + E * E$  4)  
 $E + E$  2)  
 $E$  1)

### Stack implementation of shift-reduce parser

2 operations:

shift = put next input symbol on the stack

reduce = replace a string on the stack with a left side of a production

ambiguities in grammars can be handled as special cases with rules in the algorithm

For the following grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow (E)$

$E \rightarrow id$

the state of the stack, input, and parser action are shown below:

STACK	INPUT	ACTION
(1) \$	$id_1 + id_2 * id_3 \$$	shift
(2) $\$id_1$	$+ id_2 * id_3 \$$	reduce by $E \rightarrow id$
(3) $\$E$	$+ id_2 * id_3 \$$	shift
(4) $\$E +$	$id_2 * id_3 \$$	shift
(5) $\$E + id_2$	$* id_3 \$$	reduce by $E \rightarrow id$
(6) $\$E + E$	$* id_3 \$$	shift
(7) $\$E + E *$	$id_3 \$$	shift
(8) $\$E + E * id_3$	$\$$	reduce by $E \rightarrow id$
(9) $\$E + E * E$	$\$$	reduce by $E \rightarrow E * E$
(10) $\$E + E$	$\$$	reduce by $E \rightarrow E + E$
(11) $\$E$	$\$$	accept

### 2.4.4.2 Recursive descent parser (top-down)

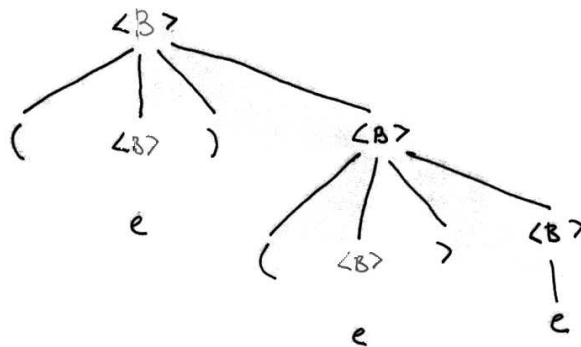
Recurses whenever a non-terminal is recursively expanded.

Eg: balanced parentheses

$\langle B \rangle := e$

$\langle B \rangle := (\langle B \rangle) \langle B \rangle$

The following parse tree would be produced when parsing the following input:  $((()))$



The tree and grammar are unambiguous. Execution order: expand paths from top down as far as they will go before moving to next path. Choose paths from left to right.

#### 2.4.4.3 Table-driven parsing (top-down)

Given a grammar such as:

- (1)  $\langle S \rangle := w \ c \ \langle S \rangle$
- (2)  $\langle S \rangle := \{ \ \langle T \rangle$
- (3)  $\langle S \rangle := s \ ;$
- (4)  $\langle T \rangle := \langle S \rangle \ \langle T \rangle$
- (5)  $\langle T \rangle := \}$

construct a parsing table:

the first column are the non-terminals; the top row are the terminals, and the entries are the production numbers used for a given non-terminal with a given terminal as the next input. A blank entry indicates it cannot be done and the parse has failed.

	w	c	{	}	s	;
$\langle S \rangle$	1		2		3	
$\langle T \rangle$	4		4	5	4	

Maintain a stack that grows from right to left (left end is the top of the stack) which is initialized with the start symbol  $\langle S \rangle$

Algorithm:

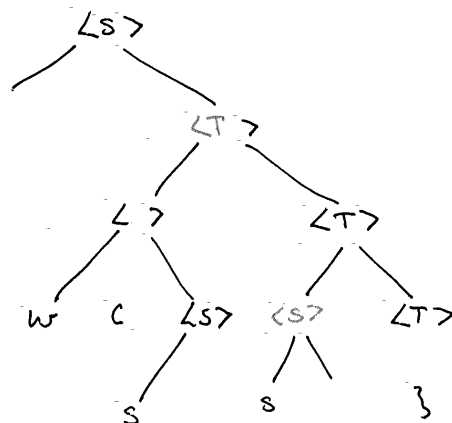
1. If the top of the stack is a terminal and the next input is the same terminal, pop it and advance the input.
2. If the top of the stack is a non-terminal, consult the parse table for the entry of that non-terminal with the corresponding next input, and replace the non-terminal with the grammar production of that rule.
3. Continue until the stack and inputs are empty (valid input) or until a failure occurs (invalid input)

Eg: for the above grammar and parse table, confirm that  $\{ w \ c \ s \ ; \ s \ ; \}$  is a valid input.

The stack looked like the following at various stages growing from right to left, although symbols were continuously being pushed and popped and it never contained more than a few of these symbols at any one time. Symbols were crossed off as they disappeared.

$\} \ s \ ; \ \langle S \rangle \ \langle T \rangle \ s \ ; \ w \ c \ \langle S \rangle \ \langle S \rangle \ \langle T \rangle \ \{ \ \langle T \rangle \ \langle S \rangle$

A parse tree is easy to build along the way. Start with  $\langle S \rangle$  as the root. Each time a non-terminal in the stack gets replaced by some items according to a grammar production, make those items the children of that node from left to right. The parse tree for the above example is shown below:



#### 2.4.5 Left-recursive grammars and left factoring

A grammar cannot have the same non-terminal both on the left side and as the first symbol of the right side because it will not be parsable.

The way to remove left recursion is to introduce an additional non-terminal:

$A \rightarrow A\alpha \mid \beta$   $L(A) = \beta\alpha^*$ , but the grammar is left recursive and therefore not parsable  
remove left recursion:

$A \rightarrow \beta R$

$R \rightarrow \alpha R \mid \epsilon$  same language, but not left recursive and therefore amenable to parsing  
to help remember the order of symbols, think of R as “rest or tail”, ie, it goes last

another eg:

$\text{expr} \rightarrow \text{expr} + \text{term} \mid \text{expr} - \text{term} \mid \text{term}$

can be converted to

$\text{expr} \rightarrow \text{term } R$

$R \rightarrow + \text{term } R \mid - \text{term } R \mid \epsilon$

Left recursion can occur in more than one production rule

eg:  $A \rightarrow A\alpha \mid A\beta \mid \gamma$

remove similarly as above

$A \rightarrow \gamma R$

$R \rightarrow \alpha R \mid \beta R \mid \epsilon$

Sometimes several rules have the same left side and the right side starts with the same non terminal so that the parser doesn't know which to choose. The problem is solved by left-factoring and is similar to the solution to left recursion.

eg:  $S \rightarrow A\alpha \mid A\beta$

left factor as:

$S \rightarrow AR$

$R \rightarrow \alpha \mid \beta$

Eg:  $\langle \text{number} \rangle := \langle \text{digit} \rangle \langle \text{number} \rangle \mid \langle \text{digit} \rangle$

In this case, add an extra non-terminal,  $\langle \text{tail} \rangle$  as follows:

$\langle \text{number} \rangle := \langle \text{digit} \rangle \langle \text{tail} \rangle$

$\langle \text{tail} \rangle := \langle \text{number} \rangle \mid \epsilon$

#### 2.4.6 Converting a regular expression to a grammar

Eg:  $a \mid bc^*$

1. set up nonterminals for all terminals

$\langle A \rangle := a$

$\langle B \rangle := b$

$\langle C \rangle := c$

2. Using correct operator precedence for the RE, convert  $\mid$  to  $\mid$ , concatenation to concatenation, and stars as  $\langle S \rangle := \langle S1 \rangle \langle S \rangle \mid e$  where  $\langle S \rangle$  is a new symbol and  $\langle S1 \rangle$  is the symbol for the  $*$ 'd item in the RE

Continuing with the above eg., the highest precedence is the  $*$ , so for  $c^*$ , add

$\langle D \rangle := \langle C \rangle \langle D \rangle \mid e$

Next, for the concatenation,

$\langle E \rangle := \langle B \rangle \langle D \rangle$

Finally, for the union,

$\langle F \rangle := \langle A \rangle \mid \langle E \rangle$

so the resulting grammar is

$\langle F \rangle := \langle A \rangle \mid \langle E \rangle$

$\langle E \rangle := \langle B \rangle \langle D \rangle$

$\langle D \rangle := \langle C \rangle \langle D \rangle \mid e$

$\langle A \rangle := a$

$\langle B \rangle := b$

$\langle C \rangle := c$

where  $\langle F \rangle$  is the start symbol

#### 2.4.7 Closure of languages

Regular languages:

- Union
- Concatenation
- Kleene star
- Intersection
- Complement
- *Not subset*

Context free:

- Union
- Concatenation
- Kleene star
- *Not intersection or complement or subset*

Recursively enumerable:

- Union
- Concatenation
- Kleene star
- Intersection
- *Not complement or subset*

Eg: consider languages  $L$  and  $L_1$  both over  $\{a,b\}$  s.t.  $L_1 = \{w \mid w \text{ contains some word in } L \text{ as a substring}\}$

If  $L$  is regular, so is  $L_1$

If  $L$  is CF, so is  $L_1$

If  $L$  is recursively enumerable, so is  $L_1$

Why?  $L_1$  is actually the Kleene star of  $L$ , and all of the language types are closed under it.

#### 2.4.8 Turing machines

Operation:

- Infinite tape
- Can both read from the tape and write onto it
- Read-write head can move both left and right
- Rejecting and accepting states take immediate effect

#### Church's Thesis (or Church-Turing Thesis)

Computable function or algorithm  $\Leftrightarrow$  function computable by a Turing machine

It is a thesis, not a mathematical formulation, ie, it can never be proven. Perhaps it will be shown to be false at some time, but it has not happened yet and is unlikely.

The writing of a universal Turing machine is solvable.

However, the following are unsolvable:

- Determining if an arbitrary Turing machine is a universal Turing machine
- Determining if a universal Turing machine can be written in fewer than k instructions for some k
- Determining if a universal Turing machine will halt (halting problem)

For a single tape deterministic Turing machine, determining that a given computation will last for n steps and will scan at least n tape squares is a decidable problem. (For some n) Determining the output after some given n steps is undecidable.

A Turing machine can be simulated with a pushdown automata with 2 stacks and the added ability to be able to pop from one stack and push onto the other without having to read any input. The two stacks simulate a Turing machine tape because left head tape movements correspond to popping from one stack and pushing onto the other, while right head tape movements are simulated by popping/pushing the opposite stacks.

#### 2.4.9 Post's correspondence problem

An undecidable problem (therefore a good test question!)

Given a set of ordered pairs of strings, a correspondence (or a match or sequence) is some combination of the pairs s.t. all the 'abscissas' concatenated together = all the 'ordinates' concatenated. Recall that any combination of the original pairs is allowed (not all, or several repeated, etc.)

Eg:

{ (ab,abb), (ba,aaa), (aa,a) } has a viable sequence:

ab	ba	aa	aa	=	abbaaaaa
abb	aaa	a	a	=	abbaaaaa
1 <sup>st</sup>	2 <sup>nd</sup>	3 <sup>rd</sup>	3 <sup>rd</sup> again		

eg:

{ (ab,aba), (baa,aa), (aba,baa) } does not have viable sequence



#### 2.4.10 de Bruijn sequence

Let  $\Sigma$  = alphabet of  $\sigma$  symbols. Let  $n \geq |\Sigma|$ . A de Bruijn sequence is a word of length  $\sigma^n$  s.t. all combinations of elements of  $\Sigma$  of length  $n$  appear in the de Bruijn sequence exactly once in a circular fashion.

Eg:  $\Sigma = \{0,1\}$ ,  $\sigma = 2$ ,  $n=3$

The combinations of length 3 are the usual  $2^3$  binary numbers 000, 010, ... 110, 111

01101010 is not a deBruijn sequence because 101 is repeated

10010110 is also not because 101 is repeated (circular)

00011101 is a deBruijn sequence

## 2.5 Other discrete structures

### 2.5.1 Logic

Tautology = statement that is always true

Unsatisfiable formula = statement that is always false (contradiction or absurdity)

Satisfiable formula = true or false depending on values of inputs (contingency)

Well formed formula = statement using correct syntax (says nothing about truth values)

A set of operators that is sufficient to represent all Boolean expressions is said to be complete. The following are complete

- and, not (eg,  $p$  or  $q$  is equivalent to  $\text{not}(\text{not } p \text{ and not } q)$ )
- or, not (eg.,  $p$  and  $q$  is equivalent to  $\text{not}(\text{not } p \text{ or not } q)$ )
- nand (eg, not  $p$  is equivalent to  $p$  nand  $p$ )
- nor (eq. not  $p$  is equivalent to  $p$  nor  $p$ )
- *and, or is not complete (nothing is equivalent to not)*

### 2.5.2 Set theory

$$A \cup B$$

$$A \cap B$$

$$A - B \Leftrightarrow A \cap \overline{B}$$

$$A \cap (B \cup C) \Leftrightarrow (A \cap B) \cup (A \cap C)$$

$$A \cup (B \cap C) \Leftrightarrow (A \cup B) \cap (A \cup C)$$

$$(A \cup B)' \Leftrightarrow A' \cap B'$$

$$(A \cap B)' \Leftrightarrow A' \cup B'$$

#### 2.5.2.1 Countability (not to be confused with finiteness)

A set is countable if there is a one-to-one correspondence between its elements and the natural numbers. It does not have to be finite to be countable, but it does have to be discrete rather than a continuum.

Eg's of countable sets are:

- Natural numbers
- Rational numbers (just a quotient of natural numbers)
- Integers
- Pairs of natural numbers such as some relations in  $N \times N$ 
  - functions from  $\{0,1\}$  to  $N$  are countable ( $N^2$ )
  - *functions from  $N$  to  $\{0,1\}$  are not ( $2^N$  is too big)*
- Subsets of  $N$ , largest subset of  $N$  etc.
- *Real numbers are not countable (there are more real numbers between 0, 1 than all the natural numbers) because of infinite precision of decimals (a continuum rather than a discrete set)*

### 2.5.3 Binary operations on sets

Group a set together with a binary operation that is closed under the operation, along with an associative property, an identity element, and an inverse for each element.

Semigroup a group w/o identity or inverses

Abelian group a group w/ the commutative property

Monoid a semi group that contains an identity element but no inverses

	closure	associative	identity	inverse	commutative
Semigroup	X	X			
Monoid	X	X	X		
Group	X	X	X	X	
Abelian group	X	X	X	X	X

#### 2.5.4 Functions

A relation  $f$ : defined on  $A \times B$  is a function iff for all  $a \in A$ ,  $\exists$  exactly one  $b \in B$  s.t.  $(a,b)$  is in  $f$

One-to-one, onto, bijection

Defs: A function  $f: A \rightarrow B$  is one-to-one if for all  $a, a'$  in  $A$ ,  $a \neq a'$  implies  $f(a) \neq f(a')$

A function  $f: A \rightarrow B$  is onto if  $\text{range}(f) = B$

A function that is both one-to-one and onto is a bijection, or one-to-one correspondence

Invertibility

A function  $f$  is invertible ( $f^{-1}$  is also a function) iff it is a bijection.

#### 2.5.5 Lattices

A partial order is a binary relation that is:

Reflexive:  $a \leq a$

Not symmetric:  $a \leq b$  and  $b \leq a$  implies  $a = b$

Transitive:  $a \leq b$  and  $b \leq c$  implies  $a \leq c$

A poset (partially ordered set) is a set with a partial order defined on it  $(P, \leq)$

Eg: subset of, less than or equal to, greater than or equal to, divisible by are all partial orders.

Note,  $\leq$  is used as a common partial order relation but usually does not mean the familiar less than or equal to relation.

$a, b \in A$  are comparable if  $a \leq b$  or  $b \leq a$ . Not all elements of a partial order need be comparable. If they all are, then the partial order is called a linear order or chain.

Hasse diagram (lattice diagram)

Since a lattice is a relation, it can be drawn with the usual digraph. However, because of the restrictions imposed by a partial order, some simplifications can be made to the digraph:

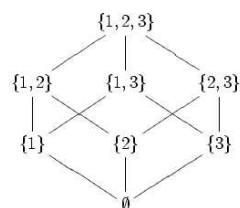
1. remove reflexive cycles (understood)
2. do not draw edges due to transitivity (understood)
3. use dots instead of circles for vertices
4. draw all edges pointing upward and eliminate arrows

A graphical method for drawing the lattice using two rules:

1. If  $x < y$  in the poset, then the point corresponding to  $x$  appears lower in the drawing than the point corresponding to  $y$ .
2. The line segment between the points corresponding to any two elements  $x$  and  $y$  of the poset is included in the drawing iff  $x$  covers  $y$  or  $y$  covers  $x$ .

ref: <http://planetmath.org/encyclopedia/HasseDiagram.html>

Example: If  $A = \mathcal{P}(\{1, 2, 3\})$ , the power set of  $\{1, 2, 3\}$ , and  $\leq$  is the subset relation  $\subseteq$ , then Hasse diagram is



Even though  $\{3\} < \{1, 2, 3\}$  (since  $\{3\} \subset \{1, 2, 3\}$ ), there is no edge directly between them because there are inbetween elements:  $\{2, 3\}$  and  $\{1, 3\}$ . However, there still remains an indirect path from  $\{3\}$  to  $\{1, 2, 3\}$ .

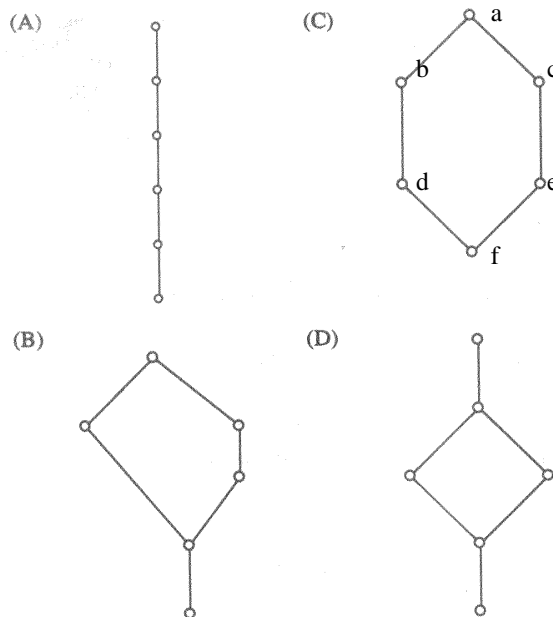
Given a poset  $(A, \leq)$  and a subset  $B$  of  $A$ , the set of all upper bounds of  $B$  and lower bounds of  $B$  are all  $a \in B$  s.t.  $a \geq \text{all } b \in B$ ,  $a \leq \text{all } b \in B$ , respectively. Note that  $\geq$  and  $\leq$  can only be determined for those elements explicitly connected by a sequence of edges, so some elements cannot be determined whether they belong to a lower or upper bound set. Indeed, a set of lower or upper bounds may be empty because no determinations could be made at all. The least upper bound LUB is the smallest of the upper bounds, and the greatest lower bound GLB is the largest of the lower bounds. Again, one or both may not exist because of the same logic.

### Lattice (finally)

A lattice is a poset  $(L, \leq)$  s.t. every subset  $\{a, b\}$  consisting of 2 elements has an LUB and a GLB. We call the LUB the join of  $a, b$  and denote it  $a \vee b$ . We call the GLB the meet of  $a, b$  and denote it  $a \wedge b$ .

Eg:  $(S, \subseteq)$  is a lattice, and if  $A, B$  are two subsets of  $S$ , then the join of  $A, B$  is  $A \cup B$  and the meet of  $A, B$  is  $A \cap B$ .

Below,  $A, D$  are valid lattices but  $C$  is not because a GLB does not exist for  $\{b, c\}$ . ie, we know  $d$  and  $e$  are both  $\geq f$ , (so  $f$  cannot be the GLB), but we don't know how  $d$  and  $e$  are related because they are not comparable.  $B$  is also not a lattice.



The following properties are true for lattices:

Idempotent:  $\exists a$  s.t.  $a \wedge a = a$ ,  $a \vee a = a$

Commutative, associative: usual defs

Absorption:  $a \wedge (a \vee b) = a \vee (a \wedge b) = a$

A lattice  $L$  is said to be bounded if it has a greatest element  $I$  and a least element  $0$ .

A lattice  $L$  is called distributive if for any elements, the distributive property holds (both ways,  $\wedge$  over  $\vee$  and vice versa)

Let lattice  $L$  be a bounded lattice with greatest element  $I$  and least element  $0$ . An element  $a'$  of  $L$  is called a complement of  $a$  if  $a \vee a' = I$  and  $a \wedge a' = 0$ . A lattice is called complemented if it is bounded and if every element has a complement. (the complements need not be unique)

A distributive lattice, if it is complemented, has unique complements.

### Boolean lattice

Boolean algebra = a boolean lattice = a complemented distributive lattice with at least 2 elements.

A distributive lattice in which each element has a complement s.t.

$$x \wedge x' = 0$$

$$x \vee x' = I \text{ (or } 1\text{)}$$

$$(x')' = x$$

$$(x \wedge y)' = x' \vee y'$$

$$(x \vee y)' = x' \wedge y'$$

Any boolean lattice must have  $2^k$  elements.

Eg: the power set under subset is a boolean lattice and therefore a boolean algebra. Logic is of course the most familiar boolean algebra, and in this case the join ( $\vee$ ) and meet ( $\wedge$ ) operations correspond to the familiar or, and operations.

### 2.5.6 Logic diagrams, truth tables, Karnaugh maps

Karnaugh map (K-map)

A 2-d grid representation of a logic function, with the special ordering of rows and columns s.t. only one bit changes at a time (unlike ordinary binary ordering of values)

The largest possible rectangles of  $2^n$  elements correspond to the simplest terms to describe the function. So, using the K-map, a truth table can be converted into a function in simplest form.

Eg: 4-variables,  $x, y, z, w$

$$f = xy + zx'$$

K-map:

	xy	00	01	11	10
zw	-----				
00				1	
01				1	
11		1	1	1	
10		1	1	1	

Race hazards: In the K-map above, two disjoint, adjacent rectangles represent a race hazard. Depending on timing, changes in input can cause glitches in output. To eliminate the hazard, form a rectangle that overlaps the boundary in question, and add the corresponding term to the function. The term is redundant, so it will not change the value of the function, but will eliminate the race hazard. The dotted rectangle in the K-map below corresponds to the term  $yz$ . The function w/o hazards is  $f = xy + zx' + yz$ .

	xy	00	01	11	10
zw					
00				1	
01				1	
11		1	1	1	
10		1	1	1	

A K-map can also be used to find a product of maxterms, or to convert a sum of minterms into a product of maxterms. In this case, rectangles are formed where 0's exist and the maxterms are those terms are identified by the variables that are 0-valued

Eg: convert  $x'yz' + x'y'z + xy + xz$  into a product of maxterms

	xy	00	01	11	10
z					
0		0	1	1	0
1		1	0	1	1

The 0's correspond to  $(y+z)$  and  $(x + y' + z')$  (taking the complements of the variables at the 0 positions)  
So the product of maxterms is  $(y+z)(x + y' + z')$

The same can be done with a truth table. Take the rows where the function is 0. For those rows, take the variables and complement them if they are 1, leave uncomplemented if 0. Form a product of maxterms from those rows.

Remember that end rows and columns can wrap around to make larger rectangles.

Eg:

$x_2$	$x_1$	$x_0$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

$x_1x_0$	00	01	11	10
$x_2$				
0		1		
1	1	1		1

wrap-around

$\overline{x_1}x_0 + x_2\overline{x_0}$

## 2.5.7 Properties of XOR

Same as mod-2 arithmetic over binary numbers

$$a \oplus a = 0$$

$$a \oplus a' = 1$$

$$a \oplus b = b \oplus a$$

$$a \oplus 0 = a$$

$$a \oplus 1 = a'$$

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

#### 2.5.8 isomorphism, homomorphism

A homomorphism is a mapping between two groups that preserves the same structure of the groups

An isomorphism is a bijection between two groups s.t. the two groups can be substituted for each other

#### 2.5.9 Equivalence relation

Reflexive, symmetric, and transitive

An equivalence relation partitions a set into clusters where all three properties hold.

### 3. Architecture (15%)

#### 3.1 Number representation

##### 3.1.1 Number conversions

Treat integer part (IP) and fractional parts (FP) separately

IP:

Divide by base until quotient becomes 0. Remainders are bits from right to left (starting at decimal point and moving away from it)

FP:

Multiply by base until fraction becomes 0. Integers are bits from left to right (again starting at decimal point and moving away)

Eg:  $41.6875_{10} = ?_2$

IP:

$41/2 = 20$	r1
$20/2 = 10$	r0
$10/2 = 5$	r0
$5/2 = 2$	r1
$2/2 = 1$	r0
$1/2 = 0$	r1

so, 101001

FP:

$.6875 * 2 = 1.375$	1
$.375 * 2 = .75$	0
$.75 * 2 = 1.5$	1
$.5 * 2 = 1.0$	1
$0 * 2 = 0$	stop

so, .1011

therefore,  $41.6875_{10} = 101001.1011_2$

The method works for any base.

Some decimals repeat or cannot be represented in a fixed number of digits and an error results. The size of this error is easy to compute for a given number by converting it to the fixed representation and computing how much the represented number differs from the original.

##### 3.1.2 2's complement

1's complement: invert all bits

2's complement: 1's complement + 1

subtraction and logic ops. are easier under 2's comp.

##### 3.1.3 excess-n biased

n is added to the number prior to storing (representing) it. n must be subtracted from the representation when reading it back. (eg  $-2$  in excess-8 binary representation is  $-2+8 = 6 = 110$ ).

##### 3.1.4 normalized binary form

the most significant digit is a 1. Eg, .0110 in normalized form is  $.110 \times 2^{-1}$  When the number is stored, the leading 1 is omitted to save space, so only 10 is stored in the above eg.



### 3.1.5 binary coded decimal BCD

every 4-bits represent a decimal digit

### 3.1.6 Representing signed integers

Sign bit = most sig. Bit, 0 = (+), 1 = (-)

3 methods:

1. sign-magnitude
2. sign-1's comp
3. sign-2's comp (most often used)

complements are only used for negative values (ie, complement everything, including the sign bit to indicate negative, then add 1)

Perform in reverse order to convert back to its decimal value.

Typically today, 2's complement means signed 2's complement. (don't complement or add/subtract 1 if sign is positive, sign bit 0)

eg., assume a 7-bit word

decimal	+9	-9
sign-mag	0 001001	1 001001
sign-1's comp	0 001001	1 110110
sign-2's comp	0 001001	1 110111

min and max values that can be stored in signed 2's complement:

eg: 4 bits

(+)ve:  $0111 = 7 = 2^3 - 1 = 2^{4-1} - 1$  (in general  $2^{n-1} - 1$ )

(-)ve:

decimal	binary	signed 2's comp
-1	-001	1111
-2	-010	1110
-3	-011	1101
-4	-100	1100
-5	-101	1011
-6	-110	1010
-7	-111	1001
-8	-1000	1000

so,  $-8 = -2^3 = -2^{n-1}$

in general, in n bits,  $[-2^{n-1}, 2^{n-1} - 1]$  (total of  $2^n$  values)

### 3.1.7 Representing floating point numbers

IEEE-754 standard for single precision 32 bit floats:

$$N = (-1)^S * 1.F * 2^{E-127}$$

Where S is the sign bit, F the fractional mantissa, E the excess-127 biased exponent

A float is stored as S:E:F, where S is 1 bit, E is 8 bits, F is 23 bits

Eg: what decimal value is represented by C1E00000?

This is :

1	10000000011	11000000000000000000000
S	E	F

So, the number is negative, the exponent is 4,  $2^4 = 16$ , the mantissa is .75, and  $-.75 * 16 = -12$

### 3.2 Memory mapped i/o

A section of RAM addresses is reserved for i/o devices. The actual data resides in the i/o device (eg, network adapter), but the CPU and i/o device exchange data through the same protocol as for accessing RAM locations. The drivers for the i/o device take care of the indirection, so that the CPU reads/writes to/from a memory location, but the device actually receives/sends the data. The address space for actual memory and devices must be disjoint, and data should be declared as volatile (eg to a compiler) so that it is not cached.

### 3.3 Processor optimizations

#### 3.3.1 RISC vs CISC ISA (instruction set architecture)

CISC (complex instruction set computer)

Higher level instructions requiring several clock cycles to execute

eg `MULT addr1 addr2`

requires `addr1` to be fetched, copied to a register, `addr2` to be fetched, copied to another register, two registers to be multiplied, result copied to `addr1` (for eg.)

RISC (reduced instruction set computer)

Lower level instructions requiring 1 clock cycle

eg `load A, addr1`

`load B, addr2`

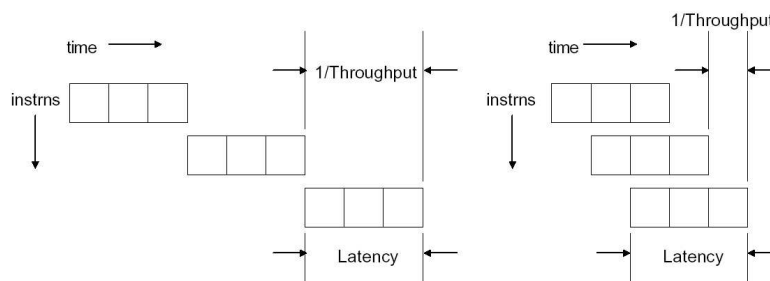
`prod A,B`

`store A, addr1`

Short, simple instruction set that permits various processor optimizations, especially pipelining. A CISC instruction set is often translated to RISC by the processor (Pentium has a CISC ISA to be backward compatible w/ x86, but is deeply pipelined)

#### 3.3.2 Pipelining

Executing stages of instructions overlapping in time. Goal is 1 instruction per clock cycle.



**Ideally,**

$$Time_{pipeline} = \frac{Time_{sequential}}{Pipeline\ Depth}$$

$$Speedup = \frac{Time_{sequential}}{Time_{pipeline}} = Pipeline\ Depth$$

In order for pipelining to be effective the following criteria are required:

- Condition codes not set by every instruction (infrequent)
- Uniform length encoding of instructions
- Instructions require uniform number of cycles to execute
- Instructions that read few arguments from memory

In general, this requires a RISC architecture

Hazards: (structural hazards and data hazards)

- Structural: two instructions attempting to access physical device (eg RAM) in same stage of pipeline
- Data: operands or results of an operation written or read by multiple instructions, possibly out of order

In a pipelined RISC architecture performance can otherwise be improved by increasing clock rate and /or instruction and data cache sizes.

### 3.4 Processor instructions

#### 3.4.1 Macro-code (macro-operation)

Eg: Add(A,B)

#### 3.4.2 Micro-code (micro-operations)

The above macro-op results in the following micor-ops:

1.  $MAR \leftarrow PC$  // copy the contents of the program counter to the memory address register
2.  $IR \leftarrow MDR, PC \leftarrow PC + 1$  // load contents of memory data register into instruction register, increment program counter
3.  $Decoder \leftarrow IR$  // decode the instruction
4.  $ABUS \leftarrow R1, BBUS \leftarrow R2$  // load operands to inputs of ALU
5.  $ALU \leftarrow ABUS + BBUS$  // perform the addition in the arithmetic logic unit
6.  $CBUS \leftarrow ALU$  // put the result at the output of the ALU
7.  $R1 \leftarrow CBUS$  // copy the result to a register

Steps 1-3 are considered the fetch phase of the instruction cycle.

### 3.5 Data communications

#### 3.5.1 Bits and baud

Baud rate = # changes in signal / second

Bit rate (bps) = # bits / second

Baud may equal bps if 1 signal change = 1 bit, as in the simplest case of 2 levels of amplitude modulation (logical 0 and 1) and no other phase modulation. Usually more than 1 bit/baud is possible using other techniques.

Bit rate = baud rate \*  $\log_2$  (# voltage levels)

### 3.5.2 Attenuation in decibels

Attenuation in dB =  $10 \log_{10}(\text{transmitted power} / \text{received power})$

eg: A signal of power 1000 mw is inserted in a transmission line and at some distance the measured power is 10 mw. The loss in signal power in decibels is

$$* \log_{10}(10 / 1000) = 10 \log_{10}(10^{-2}) = 10(-2) = -20 \text{ dB}$$

### 3.5.3 Shannon's Law

(noisy channel)

$$\text{Max bits/sec} = H \log_2(1 + S/N)$$

Where H = frequency (low pass filter of bandwidth H)

S/N = signal to noise ratio (as a rational number, not decibels; ie convert decibels to a fraction)

Eg: The signal to noise ratio for a voice grade line is 30.1 dB. What is the maximum achievable data rate on this line whose spectrum ranges from 300Hz to 3400Hz?

$$S/N = 10^{(30.1/10)} = 1023.$$

$$B = H \log_2(1 + S/N) = (3400-300) \log_2(1024) = 3100(10) = 31,000\text{bps}$$

### 3.5.4 Nyquist's formula

(perfectly noiseless channel)

With V voltage levels, max data rate =  $2H \log_2 V$  bits/sec

Eg: A digital signaling technique is required to operate at 4800bps. Each signal element encodes 8-bit bytes using multi-level phase shift keying. What is the minimum bandwidth required?

$$B = 2H \log_2 V, \text{ where } V = 256, \text{ so } \log_2 V = 8$$

$$4800 = 2H(8)$$

$$H = 300 \text{ Hz}$$

### 3.5.5 RS-232C

An interface standard between data terminal equipment DTE and data circuit terminating equipment DCT. Does not specify mechanical connector or number of pins.

### 3.5.6 CRC cyclic redundancy check

The data (M) is left shifted and then divided by a polynomial (P). The resulting remainder is appended to the data and sent. The reverse process occurs at the receiving side. The data plus remainder are divided by the polynomial and the remainder must be 0, otherwise an error occurred. The checksum is a frame check sequence (FCS) and has a desired bit length n. The divisor polynomial P must be n+1 bits long. The first and last bits of P should be 1. Refer to the 17-bit CRC-16 pattern as a popular choice.

### 3.5.7 fiber optic communication

$$\lambda f = c$$

where  $\lambda$  = wavelength, f = frequency, c = speed of light,  $3 \times 10^8$  m/s

### 3.5.8 Digital lines (T1)

Uses time division multiplexing TDM to combine 24 channels into each frame.  $24 \times 8 + 1$  framing bit = 193 bits /  $125 \mu\text{s} = 1.544 \text{ Mbps}$

### 3.6 Micro-instructions and control signals

eg: In a micro-programmed control unit the microinstructions  $M_j$  and control signals  $a, b, c, \dots$  they generate are:

$M_1$ :  $a, b, e, f$

$M_2$ :  $b, c, d, e$

$M_3$ :  $a, c, e, g$

The maximal compatibility classes MCCs for the control signals are found by:

1. Find all pairs that aren't used in the same instruction:  $\{ad, bg, cf, df, dg, fg\}$
2. Add all such triples to the set so far  $\{ad, bg, cf, df, dg, fg, dfg\}$
3. Remove redundant pairs that are included in triples  $\{ad, bg, cf, dfg\}$
4. Add any singletons that don't appear in the set  $\{e, ad, bg, cf, dfg\}$  (5 final MCCs)

Eg: For control signals  $\{a, b, c, d, e, f, g\}$  and MCCs  $\{adfg, abd, bcd, bdeg\}$ , to remove any redundancies, construct a cover table:

	a	b	c	d	e	f	g
adfg	1			1		1	1
abd	1	1		1			
bcd		1	1				1
bdeg		1		1	1		1

Keep any row where any element is the only one in a column. Remove others where all elements appear in some other row. Therefore,  $abd$  is redundant and is covered by the other MCCs, so it can be removed

Eg: a small microprocessor's microinstructions has 6 control fields, each one triggering the following number of control lines:

7, 4, 5, 2, 1, 3.

What is the fewest number of bits required to describe all the control fields.

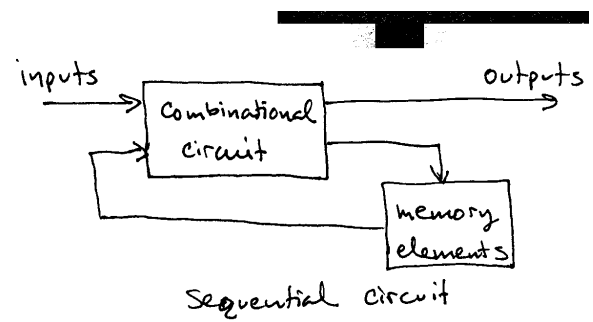
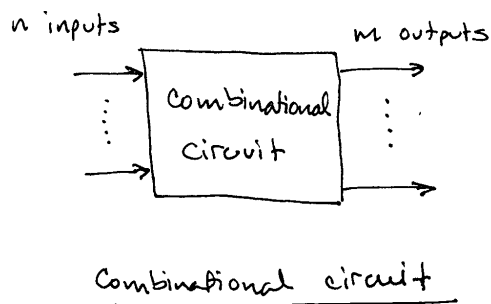
Answer: each field needs not only bits to control the lines, but also an inactive option. (equivalent of one more line) So, the number of lines are 8, 5, 6, 3, 2, 4, requiring  $3+3+3+2+1+2 = 14$  bits.

Micro instructions (microprograms) are best stored in ROM because they are not subject to change.

### 3.7 Combinational and Sequential logic

combinational: output depends only on present combination of inputs

sequential: output depends on present combination of inputs and past inputs by storing information in memory cells



#### 3.7.1 Flip flops

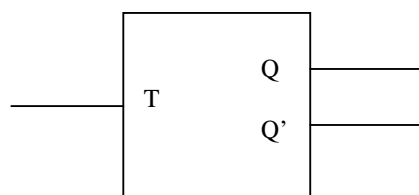
R-S flip flop

J-K flip flop

D flip-flop

T flip-flop

Changes output on every same edge of trigger signal (either always rising or always falling)



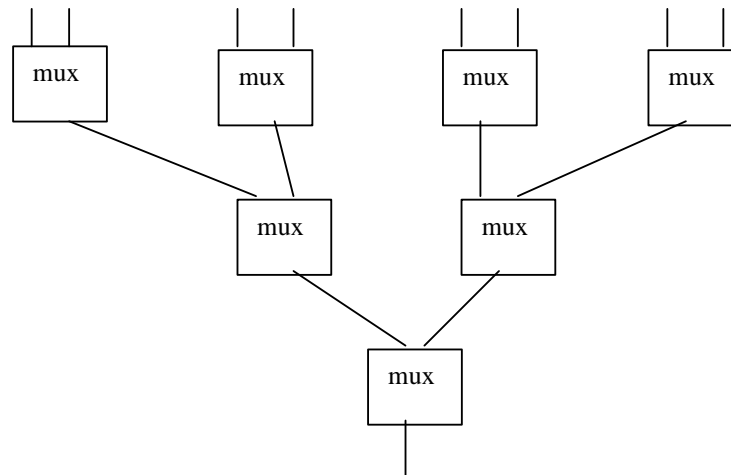
#### 3.7.2 Adders

Ripple carry adder

The carry of one bit is fed into the add of the next bit, so the bits are added sequentially  $O(n)$  where  $n$  is the number of bits (linear time)

### 3.7.3 Multiplexers

Multiplexers can be combined to increase the number of inputs, eg, an 8-input MUX can be constructed from 7 2-input MUX's



### 3.8 Hamming code, hamming distance

The distance (hamming distance) between two words  $x$ ,  $y$  is the number of bits in which they differ, ie,  $|x \oplus y|$ . The distance of a code of several words is the minimum distance between all combinations of words.

Eg: the distance of the following code

00011000

11000111

01010010

11111111

is 3 because the minimum distance is between

$|00011000 \oplus 01010010| = 3$  and this is the minimum between all combinations of words.

To detect  $d$  errors, need distance  $(d + 1)$  code

To correct  $d$  errors, need distance  $(2d + 1)$  code

### 3.9 Storage Devices

Disk transfers involve seek time, rotational latency, and block transfer time. Rotational latency, if not given, is estimated as  $\frac{1}{2}$  a track, ie,  $\frac{1}{2}$  a rotation.

Eg: a disk has a seek time of 20 ms, rotates 20 revs/sec, has 100 words per block, and 300 words per track.

The total time to access one block is: (note that one rotation is  $1/20$  or 50ms)

Seek time + rot. Latency + transfer time =  $20 \text{ ms} + 50\text{ms} / 2 + 50\text{ms} / 3 = 62\text{ms}$

## 4. Miscellaneous (5%)

### 4.1 Linear algebra

#### Similar matrices

Two square matrices A,B represent the same linear transformation w.r.t. different bases iff  $\exists$  some matrix P s.t.  $B = P^{-1}AP$ .

We say A is similar to B

Similarity invariants: A property of square matrices is said to be a similarity invariant or invariant under similarity if that property is shared by any two similar matrices. Similar matrices have the same determinant, rank, nullity, trace, characteristic polynomial and eigenvalues, and are invertible or non-invertible simultaneously.

LU, LDU decomposition of a matrix (see linear algebra text by Strang)

Multiplying 2 square matrices requires  $O(n^3)$  time (cubic)

### 4.2 Properties of exponents and logarithms

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b x/y = \log_b x - \log_b y$$

$$\log_b x^p = p \log_b x$$

$$\log_b a = \log_x a / \log_x b$$

$$b^{\log_c a} = a^{\log_c b}$$

$$\log_b 1 = 0$$

$$\log_b b = 1$$

$$b^{\log_b a} = a$$

$$a^c = b^{c \log_b a}$$

$$a^b * a^c = a^{b+c}$$

$$a^b / a^c = a^{b-c}$$

$$(a^b)^c = a^{bc} \neq a^{(b^c)} \quad \text{eg } (2^2)^3 = 2^{2*3} = 2^6 \text{ but } 2^{(2^3)} = 2^8$$



### 4.3 Linear regression using least squares approximation

Given a data set such as the following eg:

x	y
1	1.87
2	2.19
3	2.06
4	2.31
5	2.26
6	2.39
7	2.61
8	2.56
9	2.82
10	2.96

The idea is to approximate by  $y = mx + b$  and form the sum of squares of difference between the actual  $y$ 's and computed  $y$ 's.

$$E = \sum (y_{\text{act}} - mx - b)^2$$

The above is differentiated w.r.t.  $m$  and  $b$ , and each is set to 0 to find minimum, resulting in the following formulae:

$$m = (n \sum xy - \sum x \sum y) / (n \sum x^2 - (\sum x)^2)$$

$$b = (n \sum y \sum x^2 - \sum xy \sum x) / (n \sum x^2 - (\sum x)^2)$$

where  $n$  is the number of points

in the above eg,  $m = .108$  and  $b = 1.811$

### 4.4 Sums of arithmetic and geometric series

arithmetic series:  $a, a + b, a + 2b, a + 3b, \dots, a + (n-1)b$

each successive term is  $b$  larger than previous term

$$\sum_{i=0}^{n-1} a + bi = \frac{n(2a + (n-1)b)}{2} \quad \text{ie, } n \text{ times the average of the first and last terms}$$

eg:  $3+5+7+9+11$   $a=3$ (first term),  $b=2$ (step),  $n=5$ (number of terms),  $\Sigma = 35$

note that  $\sum a + bi = O(n^2)$

geometric series:  $a, ar, ar^2, ar^3, \dots, ar^{n-1}$

each successive term is  $r$  times the previous term

$$\sum_{i=0}^{n-1} ar^i = \frac{a(r^n - 1)}{(r - 1)} \quad r \text{ can be greater or less than } 1. \text{ If } r = 1, \text{ formula does not work but } \Sigma = an$$

eg:  $1+2+4+8+16$   $a=1$ (first term),  $r=2$ (ratio),  $n=5$ (number of terms),  $\Sigma = 31$

eg:  $1+1/2+1/4+1/8+1/16$   $a=1$ (first term),  $r=1/2$ (ratio),  $n=5$ (number of terms),  $\Sigma = 31/16 = 1-15/16$

note that  $\sum ar^i = O(r^n)$

## 4.5 Combinatorics

### 4.5.1 counting assignments

Also called selection with replacement; ie, a selected choice is returned to the pool of options for the next choice, so all choices have the same number of options

Eg: How many different ways can 4 rolls of the dice come up  $6 \times 6 \times 6 \times 6 = 6^4$

### 4.5.2 Permutations

Also called selection without replacement, so once an option is selected it is not available for the next choice.

Eg: In a horserace with 10 horses, how many different ways can win, place, show occur ( $1^{\text{st}}$ ,  $2^{\text{nd}}$ ,  $3^{\text{rd}}$ )  
 $10 * 9 * 8 = {}_{10}P_3 = 10! / (10-3)!$

$${}_mP_n = m! / (m-n)!$$

Sometimes a difficult quantity to compute can be easily computed as: the total possibilities - the complement of the quantity.

Eg: In Mastermind, a 4-peg code is selected from 6 colors. There are  $6^4 = 1296$  total codes. The number of codes where at least 1 color is repeated is hard to compute. However, the number of codes where no color is repeated (counting without replacement) is easy:  $6 \times 5 \times 4 \times 3 = 360 = {}_6P_4$ . So, the answer is  $1296 - 360 = 936$ .

### 4.5.3 Combinations

unordered selection without replacement

$${}_mC_n = m! / (m-n)!n!$$

### 4.5.4 Orderings with identical items

eg: anagrams

how many orderings of the letters **eilltt** are there?

$$6! / (2!2!) = 180$$

this is a normal permutations calculation of all things taken ( $6!$ ), but dividing by the factorials of the numbers of items in each class, single items being  $1!$ .

We can generalize the permutations (no items same) and combinations (all items same) calculation into a PC (perm-comb) calculation as  $mPC_n = m! / [(m-n)! i_1! * i_2! * \dots]$  where  $i_i$  is the number of same items of a given class

### 4.5.5 Distribution of objects to bins

bins must be unique; objects may be unique or some identical

case 1: all identical objects

eg: how many ways can we distribute 4 apples to 3 children?

3 bins (children); 4 objects (apples)

think of a string of  $3+4-1$  spaces  $\_ \_ \_ \_ \_$  those spaces need to be filled with 4 A's (the apples) and 2 \*'s (dividers between bins) eg  $AAA * A * = 3$  apples for  $1^{\text{st}}$  child, 1 apple for  $2^{\text{nd}}$  child, no apples for  $3^{\text{rd}}$  child.

Of the bins + objects - 1 spaces, choose "bins-1" number of them for the partition symbols or choose "objects" number of them for the object symbols.

$$\text{objects} + \text{bins} - 1 C_{\text{objects}} = (\text{objects} + \text{bins} - 1)! / (\text{bins} - 1)! (\text{objects})!$$

case 2: several classes of identical objects, classes are distinguishable

same as above, but instead of the usual  ${}_pC_q = p! / (p-q)!q!$ , modify the computation to divide by

$(i_1! * i_2! * \dots i_n!)$  instead of  $q!$  as in the denominator of the orderings of identical items computation.  
 $\text{objects} + \text{bins} - 1 \text{ } C'_{\text{objects}} = (\text{objects} + \text{bins} - 1)! / (\text{bins} - 1)! (i_1! * i_2! * \dots i_n!)$  where the  $i$ 's are the number of members in each class of objects.

#### 4.5.6 Tricks of the trade

break a count into a sequence (product) of steps

eg: the number of 1-pair poker hands (5 cards in a hand)

- |   |              |
|---|--------------|
| a. choose the rank of the pair            | 13           |
| b. choose the ranks for the other 3 cards | ${}_{12}C_3$ |
| c. choose the suits for the pair          | ${}_4C_2$    |
| d. choose the suits for the other 3 cards | $4^3$        |
| e. multiply it all together               |              |

compute a count as a difference of counts

eg: the number of straights in poker (but not straight flushes) (straight is consecutive ranks, flush is same suit)

- |    |  |  |
|----|--|--|
| a. | count the number of straight flushes (easier)      |  |
|    | choose the starting rank                           | 9 (10 is the highest rank that can start a straight) |
|    | choose the suit                                    | 4  |
|    | multiply   | 36   |
| b. | count the number of straights regardless of suit   |  |
|    | choose the starting rank                           | 9  |
|    | choose the suit                                    | $4^5$  |
|    | multiply   | 9216   |
| c. | subtract b-a to get straights that are not flushes |  |
|    |  | 9180   |

express a count as a sum of options

eg: in 10 coin tosses, how many different ways can 8 or more heads appear?

- |                        |                 |
|------------------------|-----------------|
| a. 8 heads             | ${}_{10}C_8$    |
| b. 9 heads             | ${}_{10}C_9$    |
| c. 10 heads            | ${}_{10}C_{10}$ |
| d. add it all together | 56              |

#### 4.5.7 Sample problems

Number of boolean functions

Q: How many boolean functions of 3 variables are there?

A: There are 8 combinations of the 3 inputs

x y z f

0 0 0 ?

0 0 0

...

1 1 1

For each row, f can be either 0 or 1 (2 choices)  $2^8 = 256$

Q: Of those 256 functions, how many have the property that  $f(x',y',z') = f'(x,y,z)$ ?

A: Of the 8 rows in the truth table, 4 are independent and 4 are complements, and are thus fixed by the above constraint. Therefore  $2^4$  or 16 such functions exist.

The time complexity of determining whether an arbitrary boolean function of  $n$  variables will produce a 1 value is exponential (N-P complete) because a truth table containing  $2^N$  rows is required.

Number of binary relations

If  $|A| = m$  and  $|B| = n$ , then the number of binary relations from  $A$  to  $B = 2^{mn}$

Why? The number of unique pairs  $(a,b)$  is  $mn$ . Then, each pair may either be included in a relation or not (2 choices) 2 is multiplied  $mn$  times, or  $2^{mn}$  possible relations will result. (Consider a 2-valued coefficient 0 or 1 multiplied by each pair to indicate whether it exists or not in the relation)

Eg:  $A = \{1,2,3\}$ ,  $B = \{a,b\}$ ,  $|A| = 3$ ,  $|B| = 2$

Possibilities are  $a_0(1,a)$ ,  $a_1(1,b)$ ,  $a_2(2,a)$ ,  $a_3(2,b)$ ,  $a_4(3,a)$ ,  $a_5(3,b)$  [  $mn = 6$  total terms]

The  $a_i$  coeffs. Can be 0 or 1, so  $2^6$  total possibilities

Number of binary operations

Let  $|S| = n$ . The number of binary operations that can be defined on  $S$  is  $n^{(n^2)}$

First, form a table  $S \times S$  which will have  $n^2$  entries. Then, each result has  $n$  choices, so  $n$  must be multiplied  $n^2$  times.

Number of functions

How many functions are there from  $A$  to  $B$  if  $A$  has  $m$  elements and  $B$  has  $n$  elements?

$n^m$

(All  $m$  elements of  $A$  are used, and for each, there is are  $n$  choices for  $B$ . So,  $n$  is multiplied  $m$  times)

Number of arcs in a complete undirected graph of  $n$  nodes. (no self loops)

$n(n-1) / 2$ , or  ${}_nC_2$

Number of arcs in a complete directed graph of  $n$  nodes. (including self loops)

$n^2$

Number of elements in a power set of a set of  $n$  items:  $2^n$

Form an  $n$ -bit vector. Each bit can be 0 or 1 indicating the absence or presence of an original set item in the resulting power set element.

Eg  $S = \{a,b,c\}$

a b c

0 0 0             $\{\}$

0 0 1             $\{c\}$

0 1 0             $\{b\}$

0 1 1             $\{b,c\}$

...

1 1 1             $\{a,b,c\}$

Number of ways to traverse a grid going only 2 directions:  ${}_n C_n$

given the following grid with 0,0 in the southwest corner and  $n,n$  in the northeast, the number of ways to get from the southwest to the northeast corner moving on gridlines and only going either north or east is

${}_n C_n$

List the steps in a string of  $2n$  characters each one either E or N. There can be only  $n$  of each, so choose  $n$  out of  $2n$  (order doesn't matter, all the N's are the same and all the S's are the same).

Number of 10-bit numbers that start with 01 or end with 01

01.....00             $2^6$

01.....10             $2^6$

01.....11             $2^6$

00.....01             $2^6$

10.....01             $2^6$

11.....01             $2^6$

01.....01             $2^6$

total  $7 * 2^6 = 7*64 = 448$

## 4.6 Probability

### Bayes rule

$$P(w | x) = \frac{P(x | w) P(w)}{P(x)}$$

Eg: company X shipped 5 computer chips, 1 of which was defective. Company Y shipped 4 computer chips, 2 of which were defective. One computer chip is to be chosen at random from the 9 chips shipped by the companies. If it is defective, what is the probability that it came from company Y?

$$P(Y | \text{defect}) = P(\text{defect} | Y) P(Y) / P(\text{defect}) = (1/2)(4/9) / (3/9) = 2/3$$

### Total probability theorem

If  $P(x)$  in the denominator of Bayes rule is not known, it can be computed as:  $P(x) = \sum_{i=1}^m P(x | w_i) P(w_i)$

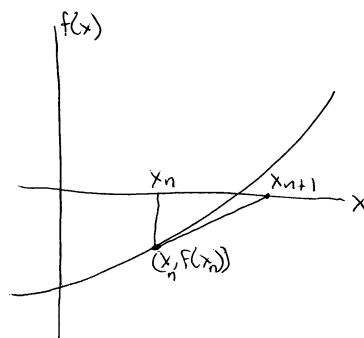
## 4.7 Statistics

$$\text{Mean } \mu = 1/n \sum_{i=1}^n d_i$$

$$\text{Variance } \sigma^2 = 1/n \sum_{i=1}^n (\mu - d_i)^2 \quad (\text{sometimes } 1/(n-1) \text{ is used instead of } 1/n)$$

$$\text{Standard deviation } \sigma = \sqrt{\sigma^2}$$

## 4.8 Newton's Method for solving equations (Newton-Raphson method)



$$f'(x_n) = \frac{-f(x_n)}{x_{n+1} - x_n}$$
$$x_{n+1} - x_n = \frac{-f(x_n)}{f'(x_n)}$$
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

The growth of the number of iterations to compute  $x$  to  $b$  bits of accuracy is  $O(\log b)$

The number of good digits (and bits) doubles each iteration. (there is a theorem that proves it is quadratic convergence)

## 4.9 Databases

### 4.9.1 Data models

Object based models

E-R model (largely academic)

Object-oriented databases

Record based models

Relational databases

Network databases (old)

Comparison of network and relational models

	network	hierarchical	relational
key	physical		logical
speed	faster		slower
data organization	moderate	worst	best

### 4.9.2 Query languages for relational databases

#### Relational algebra

Selection  $\sigma$

Projection  $\pi$

Cartesian product  $\times$

Rename  $\rho$

Set union  $\cup$

Set difference  $-$

(extension) natural join  $\bowtie$  (select tuples where common attributes are equal and remove duplicates)

eg:

$\pi_{br\_name, assets}(\text{branch} \bowtie \text{deposit} \bowtie \sigma_{city=Chicago}(\text{customer}))$

#### Relational tuple calculus

Eg:

$\{t \mid \exists s \in \text{deposit}(s[name]=t[name])\}$

#### SQL (structured query language)

Eg:

SELECT deposit.cust\_name

FROM deposit

WHERE deposit.balance = 1000 AND deposit.cust\_name = loan.cust\_name

#### QBE (query by example)

### 4.9.3 Efficiency

Always perform selection as early and often as possible, to reduce the number of tuples. Projections also reduce size (fewer columns) and should also be performed as early and often as possible. Joins always expand size, so perform joins on the smallest possible relations (via projection and selection)

### 4.9.4 Atomic transactions

#### Log-based recovery

Log entries:  
transaction  $T_i$  <starts>  
<read>  
...  
< write>  
...  
<commit>  
...  
<checkpoint>

to perform log-based recovery after a crash, from the last checkpoint, any transaction that has started and committed needs to be redone. Any that has started but not committed needs to be undone.

#### Concurrency

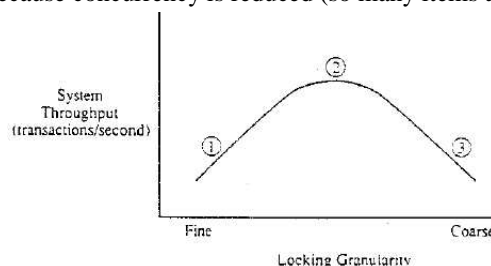
Concurrent execution of transactions that is equivalent (in terms of data written and read) to some serial execution of one transaction after another is said to be serializable. This is desirable, because otherwise results would be non-deterministic.

2 ops. are said to conflict if they access the same data item and at least one is a write op. If a schedule can be transformed into a serial schedule through a series of swaps of non-conflicting ops., then we say the schedule is conflict serializable.

#### Locking protocols

Shared and exclusive locks exist, and a transaction has 2 phases: a growing phase during which it can request locks but not release them, and a shrinking phase during which it can release locks but not request any new ones. This will guarantee serializability, although it is not free from deadlock.

In terms of efficiency of locking protocols, the graph below shows the relationship between locking granularity and throughput. Fine granularity means locking few data items, and coarse granularity means locking many items. In region 1, throughput increases because locking overhead decreases, and in region 3, throughput decreases because concurrency is reduced (so many items are locked that processes have to wait).



#### Timestamp protocols

Each transaction entering the system is given a timestamp of its start time and then any conflicts among data items are resolved in timestamp order.

## 4.7 Networking

### 4.7.1 Reference models

OSI (7 layers)

*Please Do Not Try Selling Poison Apples* (Snow White and the 7 Dwarfs)

*Physical DataLink Network Transport Session Presentation Application*

TCP/IP

*How Is The Apple?* (4 layers)

*Host-toNetwork Internet Transport Application*

Hybrid

*Please Do Not Throw Anything* (5 layers)

*Physical DataLink Network Transport Application*

### 4.7.2 802.3 (Ethernet)

frame format

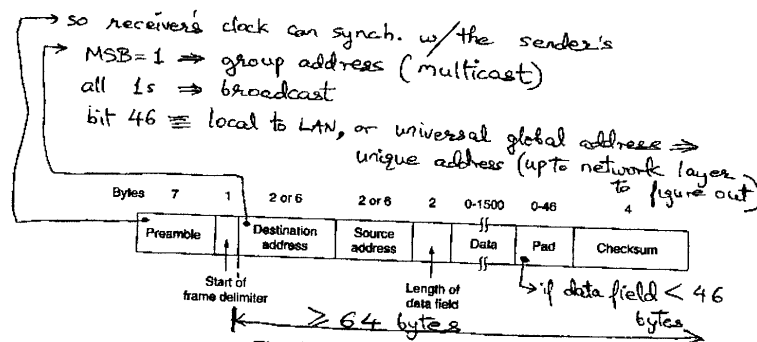


Fig. 4-21. The 802.3 frame format.

- Valid frame  $\geq 64$  bytes from dest. address to checksum
  - $\rightarrow$  to easily distinguish valid frame from garbage
  - $\rightarrow$  to prevent completion of Xmission before collision is detected, i.e., before first bit reaches other end of cable
- Short frame & collision  $\Rightarrow$  Xmission burst completes before noise burst gets back at  $2T \Rightarrow$  sender wrongly concludes successful Xmission
  - $\Rightarrow$  frame send time  $\geq 2T$
  - 10 Mbps LAN, Len = 2.5 km, 4 repeaters  $\Rightarrow$  frame time  $\geq 51.2 \mu s \approx 64$  bytes
  - High-speed  $\Rightarrow$  longer frames or shorter cables
- Checksum: hash on data field; CRC



## Characteristics

- Uses buses w/ multiple masters
- Collision detection method to ensure messages transmitted properly
- Length limited to a few hundred meters
- Packets limited in size
- *Does not use circuit switching to send messages*

## 4.7.3 IP protocol

### addresses

- Each host & router has a unique IP address
- If connected to 2 networks, 2 IP addresses
- IP addresses assigned by NIC (Network Info. Center)
- Dotted decimal notation

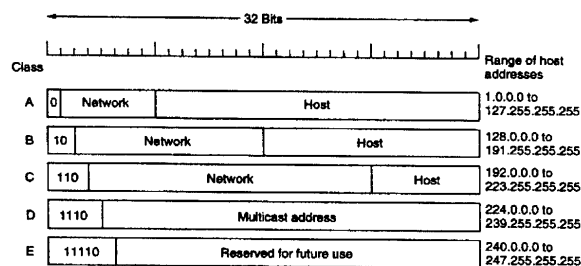


Fig. 5-47. IP address formats.

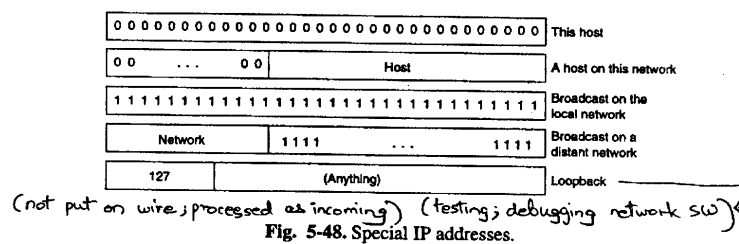


Fig. 5-48. Special IP addresses.

## Address resolution protocol

Determines the hardware address of a given IP address

#### 4.7.4 Transport layer

TCP: connection oriented, packets arrive in order in which they were sent, all packets sent along same route, order maintained at various routers along the way.

UDP: connectionless, datagrams reassembled only at destination, no guarantee of delivery, datagrams may be further fragmented during routing

#### 4.7.5 Network security

##### Encryption schemes

- Substitution ciphers – characters are substituted for each other according to a code; toys not useful for practical applications
- Transposition ciphers – characters are not disguised, simply reordered; also a toy
- One-time pad – XOR the data with a random bit string. Actually the most secure method known to date, known for decades, but not practical in actual use because the key needs to be as large as the data, and of course could never be memorized
- DES – secret key encryption circa '70s
- RSA – public key encryption currently used

Each network user has a public key and a private key. To send a message, the sender encrypts with the public key of the recipient, who then decrypts with his private key.

Key generation based on prime factorization of very large numbers. The public key can be made public because it is exceedingly difficult to factor it into the primes from which it is composed, those primes being used in the private key as well.