Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks

- Definitions:
  - Offline Problem:
    - We provide the computer with some input and after some time receive some acceptable output
  - Algorithm
    - A step-by-step procedure for solving a problem or accomplishing some end
  - Program
    - an algorithm expressed in a language the computer can understand
  - An algorithm solves a problem if it produces an acceptable output on EVERY input

- Goals of this course:
  1) To learn how to convert (nontrivial) algorithms into programs
     - Often what seems like a fairly simple algorithm is not so simple when converted into a program
     - Other algorithms are complex to begin with, and the conversion must be carefully considered
     - Many issues must be dealt with during the implementation process
       - Let's hear some

2) To see and understand differences in algorithms and how they affect the run-times of the associated programs
   - Many problems can be solved in more than one way
   - Different solutions can be compared using many factors
     - One important factor is program run-time
       - Sometimes a better run-time makes one algorithm more desirable than another
       - Sometimes a better run-time makes a problem solution feasible where it was not feasible before
     - However, there are other factors for comparing algorithms
       - Let's hear some

- Determine resource usage as a function of input size
  - Which resources?
- Ignore multiplicative constants and lower order terms
  - Why?
- Measure performance as input size increases without bound (toward infinity)
  - Asymptotic performance
- Use some standard measure for comparison
  - Do we know any measures?

- Big O
  - Upper bound on the asymptotic performance
- Big Omega
  - Lower bound on the asymptotic performance
- Theta
  - Upper and lower bound on the asymptotic performance – Exact bound
- Compare on the board

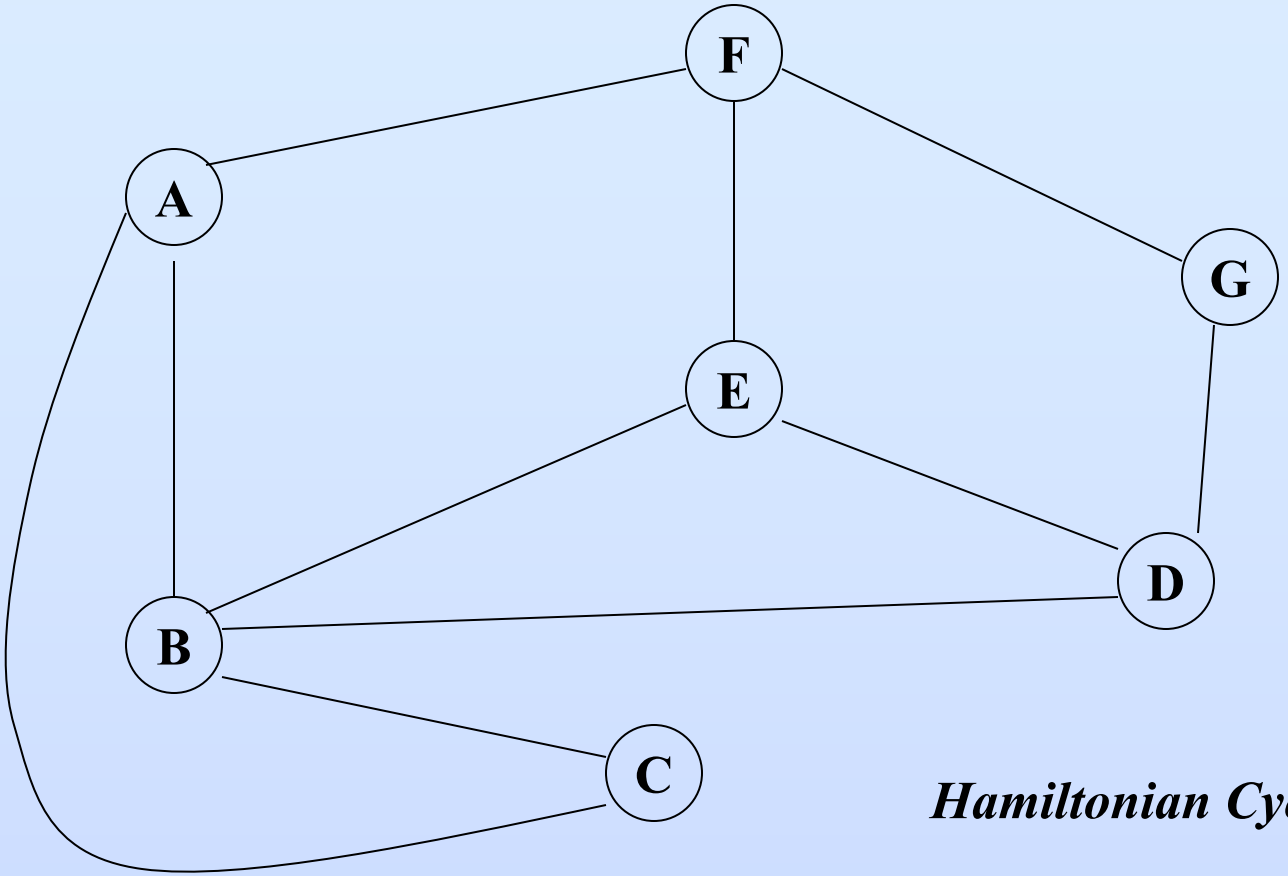
- So why would we ever use Big O?
  - Theta is harder to show in some cases
  - Lower bounds are typically harder to show than upper bounds
    - So sometimes we can just determine the upper bound

- So is algorithm analysis really important?
  - Yes! Different algorithms can have considerably different run-times
    - Sometimes the differences are subtle but sometimes they are extreme
    - Let's look at a table of growth rates on the board
      - Note how drastic some of the differences are for large problem sizes

- Consider 2 choices for a programmer
  1) Implement an algorithm, then run it to find out how long it takes
  2) Determine the asymptotic run-time of the algorithm, then, based on the result, decide whether or not it is worthwhile to implement the algorithm
  - Which choice would you prefer?
  - Discuss

- Idea:
  - We find a solution to a problem by considering (possibly) all potential solutions and selecting the correct one
- Run-time:
  - The run-time is bounded by the number of possible solutions to the problem
  - If the number of potential solutions is exponential, then the run-time will be exponential

- Example: Hamiltonian Cycle
  - A Hamiltonian Cycle (HC) in a graph is a cycle that visits each node in the graph exactly one time
    - See example on board and on next slide
  - Note that an HC is a permutation of the nodes in the graph (with a final edge back to the starting vertex)
    - Thus a fairly simple exhaustive search algorithm could be created to try all permutations of the vertices, checking each with the actual edges in the graph
      - See text Chapter 44 for more details



Hamiltonian Cycle Problem

A solution is:

ACBEDGFA

- Unfortunately, for N vertices, there are N! permutations, so the run-time here is poor

- Pruning and Branch and Bound
  - How can we improve our exhaustive search algorithms?
    - Think of the execution of our algorithm as a tree
      - Each path from the root to a leaf is an attempt at a solution
      - The exhaustive search algorithm may try every path
    - We'd like to eliminate some (many) of these execution paths if possible
      - If we can prune an entire subtree of solutions, we can greatly improve our algorithm runtime

- For example on board:
  - Since edge (C, D) does not exist we know that no paths with (C, D) in them should be tried
    - > If we start from A, this prunes a large subtree from the tree and improves our runtime
  - Same for edge (A, E) and others too

Important note:

- Pruning / Branch and Bound does NOT improve the algorithm asymptotically
  - The worst case is still exponential in its run-time
- However, it makes the algorithm practically solvable for much larger values of N

- Exhaustive Search algorithms can often be effectively implemented using recursion
  - Think again of the execution tree
    - Each recursive call progresses one node down the tree
    - When a call terminates, control goes back to the previous call, which resumes execution
      - BACKTRACKING

- Idea of backtracking:
  - Proceed forward to a solution until it becomes apparent that no solution can be achieved along the current path
    - At that point UNDO the solution (backtrack) to a point where we can again proceed forward
  - Example: 8 Queens Problem
    - How can I place 8 queens on a chessboard such that no queen can take any other in the next move?
      - Recall that queens can move horizontally, vertically or diagonally for multiple spaces
    - See on board

- 8 Queens Exhaustive Search Solution:
  - Try placing the queens on the board in every combination of 8 spots until we have found a solution
    - This solution will have an incredibly bad run-time
    - 64 C 8 = (64!)/[(8!)(64-8)!]
      = (64*63*62*61*60*59*58*57)/40320
      = **4,426,165,368 combinations**

      (multiply by 8 for total queen placements)
  - However, we can improve this solution by realizing that many possibilities should not even be tried, since no solution is possible
  - Ex: Any solution has exactly one queen in each column

  - This would eliminate many combinations, but would still allow 8^8 = 16,777,216 possible solutions (x 8 = 134,217,728 total queen placements)
- If we further note that all queens must be in different rows, we reduce the possibilities more
  - Now the queen in the first column can be in any of the 8 rows, but the queen in the second column can only be in 7 rows, etc.
  - This reduces the possible solutions to 8! = 40320 (x 8 = 322,560 individual queen placements)
  - We can implement this in a recursive way
- However, note that we can prune a lot of possibilities from even this solution execution tree by realizing early that some placements cannot lead to a solution
  - Same idea as for Hamiltonian cycle – we are pruning the execution tree

  - Ex: No queens on the same diagonal
  - See example on board
- Using this approach we come up with the solution as shown in 8-Queens handout
  - JRQueens.java
- Idea of solution:
  - Each recursive call attempts to place a queen in a specific column
    - A loop is used, since there are 8 squares in the column
  - For a given call, the state of the board from previous placements is known (i.e. where are the other queens?)
  - If a placement within the column does not lead to a solution, the queen is removed and moved "down" the column

- When all rows in a column have been tried, the call terminates and backtracks to the previous call (in the previous column)
- If a queen cannot be placed into column i, do not even try to place one onto column i+1 – rather, backtrack to column i-1 and move the queen that had been placed there
- Using this approach we can reduce the number of potential solutions even more
- See handout results for details

- Another example: finding words in a Boggle Board
  - Idea is to form words from letters on mixed up printed cubes
  - The cubes are arranged in a two dimensional array, as shown below
  - Words are formed by starting at any location in the cube and proceeding to adjacent cubes horizontally, vertically or diagonally
  - Any cube may appear at most one time in a word
  - For example, FRIENDLY and FROSTED are legal words in the board to the right

F	R	O	O
Y	I	E	S
L	D	N	T
A	E	R	E

- This problem is very different from 8 Queens
- However, many of the ideas are similar
  - Each recursive call adds a letter to the word
  - Before a call terminates, the letter is removed
- But now the calls are in (up to) eight different directions:
  - For letter [i][j] we can recurse to
    - letter [i+1][j] letter [i-1][j]
    - letter [i][j+1] letter[i][j-1]
    - letter [i+1][j+1] letter[i+1][j-1]
    - letter [i-1][j+1] letter[i-1][j-1]
- If we consider all possible calls, the runtime for this is **enormous!**
  - Has an approx. upper bound of $16! \approx 2.1 \times 10^{13}$

- Naturally, not all recursive calls may be possible
  - We cannot go back to the previous letter since it cannot be reused
    - > Note if we could words could have infinite length
  - We cannot go past the edge of the board
  - We cannot to to any letter that does not yield a valid prefix to a word
    - > Practically speaking, this will give us the greatest savings
    - > For example, in the board shown (based on our dictionary), no words begin with FY, so we would not bother proceeding further from that prefix
  - Execution tree is pruned here as well
  - Show example on board

- Building a crossword puzzle is yet another problem
  - In this case words are only in a straight line across or down
  - However, words affect each other
    - Choice of a word in one direction could affect many words in the other direction
  - For more details go to recitation

- Since a focus of this course is implementing algorithms, it is good to look at some implementation issues
  - Consider building / unbuilding strings that are considered in the Boggle game
    - Forward move adds a new character to the end of a string
    - Backtrack removes the most recent character from the end of the string
    - In effect our string is being used as a Stack – pushing for a new letter and popping to remove a letter

- We know that Stack operations push and pop can both be done in Θ(1) time
  - Unless we need to resize, which would make a push linear
- Unfortunately, the String data type in Java stores a constant string – it cannot be mutated
  - So how do we "push" a character onto the end?
  - In fact we must create a new String which is the previous string with one additional character
  - This has the overhead of allocating and initializing a new object for each "push", with a similar overhead for a "pop"
  - Thus, push and pop have become Θ(N) operations, where N is the length of the string
    - > Very inefficient!

  - For example:

    S = new String("ReallyBogusLongStrin");

    S = S + "g";

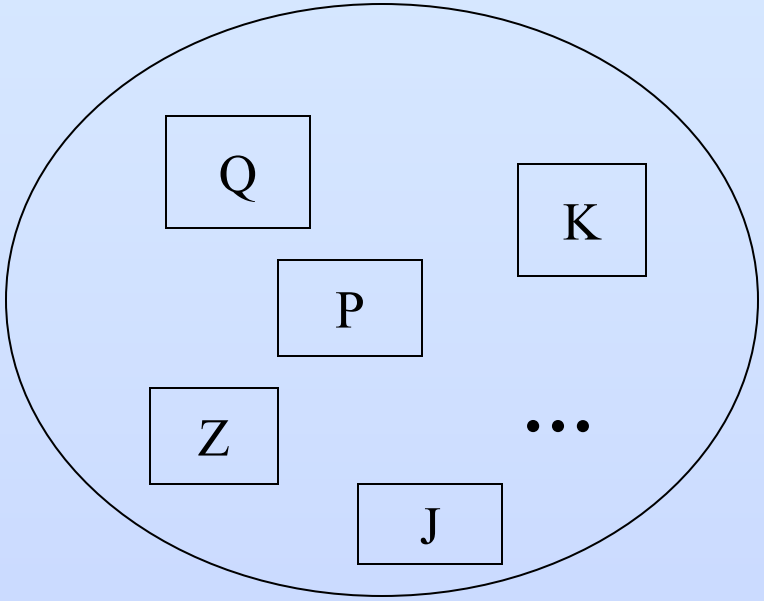
  - Consider now a program which does many thousands of these operations – you can see why it is not a preferred way to do it

- To make the "push" and "pop" more efficient (Θ(1)) we could instead use a StringBuffer (or StringBuilder)
  - append() method adds to the end without creating a new object
  - Reallocates memory only when needed
  - However, if we size the object correctly, reallocation need never be done
    - > Ex: For Boggle (4x4 square)

      S = new StringBuffer(16);

- Consider the task of searching for an item within a collection
  - Given some collection C and some key value K, find/retrieve the object whose key matches K

K



- How do we know how to search so far?
  - Well let's first think of the collections that we know how to search
    - Array/Vector
      - Unsorted
        - How to search? Run-time?
      - Sorted
        - How to search? Run-time?
    - Linked List
      - Unsorted
        - How to search? Run-time?
      - Sorted
        - How to search? Run-time?

- Binary Search Tree
  - Slightly more complicated data structure
  - Run-time?
    - Are average and worst case the same?

- So far binary search of a sorted array and a BST search are the best we have
  - Both are pretty good, giving $O(\log_2 N)$ search time
- Can we possibly do any better?
  - Perhaps if we use a very different approach

- Consider BST search for key K
  - For each node T in the tree we have 4 possible results
    1) T is empty (or a sentinel node) indicating item not found
    2) K matches T.key and item is found
    3) K < T.key and we go to left child
    4) K > T.key and we go to right child
  - Consider now the same basic technique, but proceeding left or right based on the current bit within the key

- Call this tree a Digital Search Tree (DST)
- DST search for key K
  - For each node T in the tree we have 4 possible results
    1) T is empty (or a sentinel node) indicating item not found
    2) K matches T.key and item is found
    3) Current bit of K is a 0 and we go to left child
    4) Current bit of K is a 1 and we go to right child
  - Look at example on board

- Run-times?
  - Given N random keys, the height of a DST should average $O(\log_2 N)$
    - Think of it this way – if the keys are random, at each branch it should be equally likely that a key will have a 0 bit or a 1 bit
      - Thus the tree should be well balanced
  - In the worst case, we are bound by the number of bits in the key (say it is b)
    - So in a sense we can say that this tree has a constant run-time, if the number of bits in the key is a constant
      - This is an improvement over the BST

- But DSTs have drawbacks
  - Bitwise operations are not always easy
    - Some languages do not provide for them at all, and for others it is costly
  - Handling duplicates is problematic
    - Where would we put a duplicate object?
      - Follow bits to new position?
      - Will work but Find will always find first one
        - > Actually this problem exists with BST as well
      - Could have nodes store a collection of objects rather than a single object

- Similar problem with keys of different lengths
  - What if a key is a prefix of another key that is already present?
- Data is not sorted
  - If we want sorted data, we would need to extract all of the data from the tree and sort it
- May do b comparisons (of entire key) to find a key
  - If a key is long and comparisons are costly, this can be inefficient

- Let's address the last problem
  - How to reduce the number of comparisons (of the entire key)?
  - We'll modify our tree slightly
    - All keys will be in exterior nodes at leaves in the tree
    - Interior nodes will not contain keys, but will just direct us down the tree toward the leaves
  - This gives us a Radix Search Trie
    - Trie is from reTRIEval (see text)

- Benefit of simple Radix Search Tries
  - Fewer comparisons of entire key than DSTs
- Drawbacks
  - The tree will have more overall nodes than a DST
    - Each external node with a key needs a unique bit-path to it
  - Internal and External nodes are of different types
  - Insert is somewhat more complicated
    - Some insert situations require new internal as well as external nodes to be created
      - We need to create new internal nodes to ensure that each object has a unique path to it
      - See example

- Run-time is similar to DST
  - Since tree is binary, average tree height for N keys is $O(\log_2 N)$
    - However, paths for nodes with many bits in common will tend to be longer
  - Worst case path length is again b
    - However, now at worst b bit comparisons are required
    - We only need one comparison of the entire key
  - So, again, the benefit to RST is that the entire key must be compared only one time

- How can we improve tries?
  - Can we reduce the heights somehow?
    - Average height now is $O(\log_2 N)$
  - Can we simplify the data structures needed (so different node types are not required)?
  - Can we simplify the Insert?
- We will examine a couple of variations that improve over the basic Trie

- RST that we have seen considers the key 1 bit at a time
  - This causes a maximum height in the tree of up to b, and gives an average height of $O(\log_2 N)$ for N keys
  - If we considered m bits at a time, then we could reduce the worst and average heights
    - Maximum height is now b/m since m bits are consumed at each level
    - Let $M = 2^m$
      - Average height for N keys is now $O(\log_M N)$, since we branch in M directions at each node

- Let's look at an example
  - Consider 2^{20} (1 meg) keys of length 32 bits
    - Simple RST will have
      - > Worst Case height = 32
      - > Ave Case height = $O(\log_2[2^{20}]) \approx 20$
    - Multiway Trie using 8 bits would have
      - > Worst Case height = 32/8 = 4
      - > Ave Case height = $O(\log_{256}[2^{20}]) \approx 2.5$
- This is a considerable improvement
- Let's look at an example using character data
  - We will consider a single character (8 bits) at each level
  - Go over on board

- So what is the catch (or cost)?
  - Memory
    - Multiway Tries use considerably more memory than simple tries
  - Each node in the multiway trie contains M pointers/references
    - In example with ASCII characters, M = 256
  - Many of these are unused, especially
    - During common paths (prefixes), where there is no branching (or "one-way" branching)
      - > Ex: through and throughout
    - At the lower levels of the tree, where previous branching has likely separated keys already

- Idea:
  - Save memory and height by eliminating all nodes in which no branching occurs
    - See example on board
  - Note now that since some nodes are missing, level i does not necessarily correspond to bit (or character) i
    - So to do a search we need to store in each node which bit (character) the node corresponds to
      - However, the savings from the removed nodes is still considerable

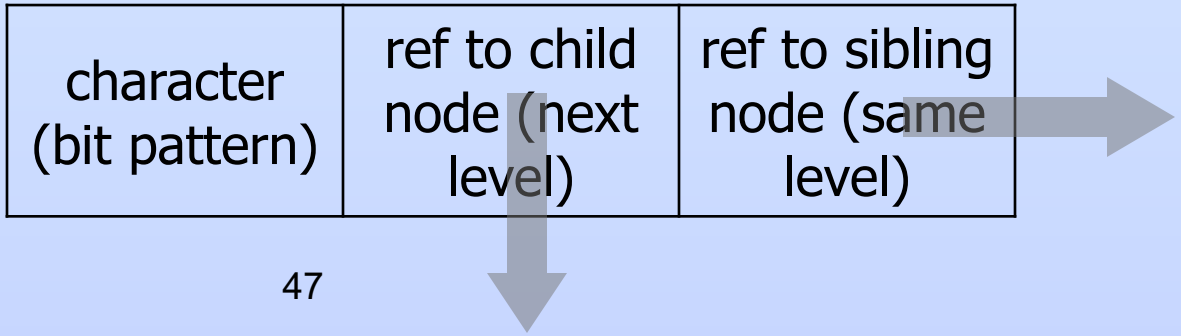
- Also, keep in mind that a key can match at every character that is checked, but still not be actually in the tree
  - Example for tree on board:
    - If we search for TWEEDLE, we will only compare the T**E**E
    - However, the next node after the E is at index 8. This is past the end of TWEEDLE so it is not found
- Run-time?
  - Similar to those of RST and Multiway Trie, depending on how many bits are used per node

- So Patricia trees
  - Reduce tree height by removing "one-way" branching nodes
  - Text also shows how "upwards" links enable us to use only one node type
    - TEXT VERSION makes the nodes homogeneous by storing keys within the nodes and using "upwards" links from the leaves to access the nodes
      - > So every node contains a valid key. However, the keys are not checked on the way "down" the tree – only after an upwards link is followed
  - Thus Patricia saves memory but makes the insert rather tricky, since new nodes may have to be inserted between other nodes
    - See text

- Even with Patricia trees, there are many unused pointers/references, especially after the first few levels
  - Continuing our example of character data, each node has 256 pointers in it
    - Many of these will never be used in most applications
    - Consider words in English language
      - Not every permutation of letters forms a legal word
      - Especially after the first or second level, few pointers in a node will actually be used
  - How can we eliminate many of these references?

- Idea of de la Briandais Trees (dlB)
  - Now, a "node" from multiway trie and Patricia will actually be a linked-list of nodes in a dlB
    - Only pointers that are used are in the list
      - Any pointers that are not used are not included in the list
    - For lower levels this will save an incredible amount
  - dlB nodes are uniform with two references each
    - One for sibling and one for a single child

de la Briandais node



- For simplicity of Insert, we will also not have keys stored at all, either in internal or external nodes
  - Instead we store one character (or generally, one bit pattern) per node
  - Nodes will continue until the end of each string
    - We match each character in the key as we go, so if a null reference is reached before we get to the end of the key, the key is not found
    - However, note that we may have to traverse the list on a given level before finding the correct character
- Look at example on board

- Run-time?
  - Assume we have S valid characters (or bit patterns) possible in our "alphabet"
    - Ex. 256 for ASCII
  - Assume our key contains K characters
  - In worst case we can have up to Θ(KS) character comparisons required for a search
    - Up to S comparisons to find the character on each level
    - K levels to get to the end of the key
  - However, this is unlikely
    - Remember the reason for using dlB is that most of the levels will have very few characters
    - So practically speaking a dlB search will require Θ(K) time

- Implementing dlBs?
  - We need minimally two classes
    - Class for individual nodes
    - Class for the top level DLB trie
  - Generally, it would be something like:

Java	C++

`public class DLB {`
 	
`  private DLBnode root;`
 	
`  // constructor plus other`
`  // methods`
 	
`  private class DLBnode {`
`  public char value;`
`  public DLBnode rightSib;`
`  public DLBnode child;`
`  }`
`}`
	`class DLBnode`
	`{`
	`  public:`
	`  char value;`
	`  DLBnode * rightSibling;`
	`  DLBnode * child;`
	`}`
	`class DLB`
	`{`
	`  private:`
	`  DLBnode * root;`
	`  public:`
	`  // constructor plus other`
	`  // methods`
	`}`

- Search Method:
  - At each level, follow rightSibling references until
    - > character is matched (PROCEED TO NEXT LEVEL) or
    - > NULL is reached (string is not found)
  - Proceed down levels until "end of string" character coincides with end of key
    - > For "end of string" character, use something that you know will not appear in any string. It is needed in case a string is a prefix of another string also in the tree
- Insert Method
  - First make sure key is not yet in tree
  - Then add nodes as needed to put characters of key into the tree
    - > Note that if the key has a prefix that is already in the tree, nodes only need to be added after that point
    - > See example on board

- Delete Method:
  - This one is a bit trickier to do, since you may have to delete a node from within the middle of a list
  - Also, you only delete nodes up until the point where a branch occurred
    - > In other words, a prefix of the word you delete may still be in the tree
    - > This translates to the node having a sibling in the tree
  - General idea is to find the "end of string" character, then backtrack, removing nodes until a node with a sibling is found
    - > In this case, the node is still removed, but the deletion is finished
    - > Determining if the node has a sibling is not always trivial, nor is keeping track of the pointers
  - See example on board

- Also useful (esp. for Assignment 1)
  - Search prefix method
    - This will proceed in the same way as Search, but will not require an "end of string" character
    - In fact, Search and Search prefix can easily be combined into a single 4-value method:
      - Return 0 if the prefix is not found in the trie
      - Return 1 if the prefix is found but the word does not exist (no "end of string" character found)
      - Return 2 if the word is found
      - Return 3 if the word is found and it is a prefix
    - This way a single method call can be used to determine if a string is a valid word and / or a valid prefix
    - **For maximum credit, this approach must be used in Assignment 1 Part B**

- So far what data structures do we have that allow for good searching?
  - Sorted arrays (lgN using Binary Search)
  - BSTs (if balanced, search is lgN)
  - Using Tries (Theta(K) where we have K characters in our string)
- Note that using Tries gives us a search time that is independent of N
  - However, Tries use a lot of memory, especially if strings do not have common prefixes

- Can we come up with another Theta(1) search that uses less memory for arbitrary keys?
- Let's try the following:
  - Assume we have an array (table), T of size M
  - Assume we have a function h(x) that maps from our key space into indexes {0,1,…,M-1}
    - Also assume that h(x) can be done in time proportional to the length of the key
  - Now how can we do an Insert and a Find of a key x?

- Insert

```
i = h(x);
T[i] = x;
```

- Find

```
i = h(x);
if (T[i] == x) return true;
else return false;
```

- This is the simplistic idea of hashing
  - Why simplistic?
  - What are we ignoring here?
    - Discuss

- Simple hashing fails in the case of a collision:

  $h(x_1) == h(x_2)$, where $x_1 != x_2$

- Can we **avoid collisions** (i.e. guarantee that they do not occur)?
  - Yes, but only when size of the key space, K, is less than or equal to the table size, M
    - When |K| <= M there is a technique called ***perfect hashing*** that can ensure no collisions
    - It also works if N <= M, but the keys are known in advance, which in effect reduces the key space to N
      - > Ex: Hashing the keywords of a programming language during compilation of a program

- When |K| > M, by the pigeonhole principle, collisions cannot be eliminated
  - We have more pigeons (potential keys) than we have pigeonholes (table locations), so at least 2 pigeons must share a pigeonhole
  - Unfortunately, this is usually the case
  - For example, an employer using SSNs as the key
    - Let M = 1000 and N = 500
    - It seems like we should be able to avoid collisions, since our table will not be full
    - However, |K| = 10^9 since we do not know what the 500 keys will be in advance (employees are hired and fired, so in fact the keys change)

- So we must redesign our hashing operations to work despite collisions
  - We call this collision resolution
- Two common approaches:

1) Open addressing
   - If a collision occurs at index i in the table, try alternative index values until the collision is resolved
     - Thus a key may not necessarily end up in the location that its hash function indicates
     - We must choose alternative locations in a consistent, predictable way so that items can be located correctly
     - Our table can store at most M keys

2) Closed addressing

- Each index i in the table represents a collection of keys
  - Thus a collision at location i simply means that more than one key will be in or searched for within the collection at that location
  - The number of keys that can be stored in the table depends upon the maximum size allowed for the collections

- Before discussing resolution in detail
  - Can we at least keep the number of collisions in check?
  - Yes, with a good hash function
    - The goal is to make collisions a "random" occurrence
      - Collisions will occur, but due to chance, not due to similarities or patterns in the keys
  - What is a good hash function?
    - It should utilize the entire key (if possible) and exploit any differences between keys

- Let's look at some examples
  - Consider hash function for Pitt students based on phone numbers
    - Bad: First 3 digits of number
      - > Discuss
    - Better?
      - > See board
  - Consider hash function for words
    - Bad: Add ASCII values
      - > Discuss
    - Better?
      - > See board and text

- Generally speaking we should:
  - Choose M to be a prime number
  - Calculate our hash function as

    h(x) = f(x) mod M

    - where f(x) is some function that converts x into a large "random" integer in an intelligent way
      - It is not actually random, but the idea is that if keys are converted into very large integers (much bigger than the number of actual keys) collisions will occur because of pigeonhole principle, but they will be less frequent

- Back to Collision Resolution
  - Open Addressing
    - The simplest open addressing scheme is Linear Probing
      - Idea: If a collision occurs at location i, try (in sequence) locations i+1, i+2, … (mod M) until the collision **is resolved**
      - For Insert:
        - > Collision is resolved when an empty location is found
      - For Find:
        - > Collision is resolved (found) when the item is found
        - > Collision is resolved (not found) when an empty location is found, or when index circles back to i
      - Look at an example

- Performance
  - **Theta(1)** for Insert, Search for normal use, subject to the issues discussed below
    - > In normal use at most a few probes will be required before a collision is resolved
- Linear probing issues
  - What happens as table fills with keys?
  - Define LOAD FACTOR, α = N/M
  - How does α affect linear probing performance?
  - Consider a hash table of size M that is empty, using a good hash function
    - > Given a random key, x, what is the probability that x will be inserted into any location i in the table?

$1/M$

- Consider now a hash table of size M that has a cluster of C consecutive locations that are filled
  - > Now given a random key, x, what is the probability that x **will be inserted** into the location immediately following the cluster?

    $$(C+1)/M$$

  - > Discuss
- How can we "fix" this problem?
  - > Even AFTER a collision, we need to make all of the locations available to a key
  - > This way, the probability from filled locations will be redistributed throughout the empty locations in the table, rather than just being pushed down to the first empty location after the cluster
  - > Discuss

- Double Hashing
  - Idea: When a collision occurs, increment the index, just as in linear probing. However, now do not automatically choose 1 as the increment value
    - > Instead use a second hash function to determine the **increment**
  - Discuss
  - Look at example
- We must be careful to ensure that double hashing always "works"
  - Make sure increment is > 0
  - Make sure no index is tried twice before all are tried once
    - > Show example

- As N increases, double hashing shows a definite improvement over linear probing
  - Discuss
- However, as N approaches M, both schemes degrade to Theta(N) performance
  - Since there are only M locations in the table, as it fills there become fewer empty locations remaining
  - Multiple collisions will occur even with double hashing
  - This is especially true for inserts and unsuccessful finds
    - > Both of these continue until an empty location is found, and few of these exist
    - > Thus it could take close to M probes before the collision is resolved
    - > Since the table is almost full Theta(M) = Theta(N)

- Open Addressing Issues
  - We have just seen that performance degrades as N approaches M
    - Typically for open addressing we want to keep the table partially empty
      - > For linear probing, α = 1/2 is a good rule of thumb
      - > For double hashing, we can go a bit higher
  - What about delete?
    - Discuss problem
    - Discuss pseudo-solution
    - Can we use hashing without delete?
      - > Yes, in some cases (ex: compiler using language keywords)

- Closed Addressing
  - Most common form is separate chaining
    - Use a simple linked-list at each location in the table
      - Look at example
      - Discuss performance
    - Note performance is dependent upon chain length
      - Chain length is determined by the load factor, α
      - As long as α is a small constant, performance is still Theta(1)
        - > Graceful degradation
        - > However, a poor hash function may degrade this into Theta(N)
        - > Discuss

- Would other collections improve over separate chaining?
  - Sorted array?
    - Space overhead if we make it large and copying overhead if we need to resize it
    - Inserts require shifting
  - BST?
    - Could work
      - > Now a poor hash function would lead to a large tree at one index – still Theta(logN) as long as tree is relatively balanced
    - But is it worth it?
      - > Not really – separate chaining is simpler and we want a good hash function anyway

- Basic Idea:
  - Given a pattern string P, of length M
  - Given a text string, A, of length N
    - Do all characters in P match a substring of the characters in A, starting from some index i?
  - Brute Force (Naïve) Algorithm:

```
int brutesearch(char *p, char *a)
{
  int i, j, M = strlen(p), N = strlen(a);
  for (i = 0, j = 0; j < M && i < N; i++, j++)
    if (a[i] != p[j]) { i -= j; j = -1; }
  if (j == M) return i-M; else return i;
}
```

    - **Do example**

- Performance of Naïve algorithm?
  - Normal case?
    - Perhaps a few char matches occur prior to a mismatch

Theta(N + M) = Theta(N) when N >> M

  - Worst case situation and run-time?

A = XXXXXXXXXXXXXXXXXXXXXXXXXXXXY

P = XXXXY

• P must be completely compared each time we move one index down A

M(N-M+1) = Theta(NM) when N >> M

- Improvements?
  - Two ideas
    - **Improve the worst case performance**
      - Good theoretically, but in reality the worst case does not occur very often for ASCII strings
      - Perhaps for binary strings it may be more important
    - **Improve the normal case performance**
      - This will be very helpful, especially for searches in long files

- KMP (Knuth Morris Pratt)
  - Improves the worst case, but not the normal case
  - Idea is to prevent index from ever going "backward" in the text string
    - This will guarantee Theta(N) runtime in the worst case
  - How is it done?
    - Pattern is preprocessed to look for "sub" patterns
    - As a result of the preprocessing that is done, we can create a "next" array that is used to determine the next character in the pattern to examine

- We don't want to worry too much about the details here

```
int kmpsearch(char *p, char *a)
{
  int i, j, M = strlen(p), N = strlen(a);
  initnext(p);
  for (i = 0, j = 0; j < M && i < N; i++, j++)
     while ((j >= 0) && (a[i] != p[j])) j = next[j];
  if (j == M) return i-M; else return i;
}
```

- Note that i never decreases and whenever i is not changing (in the while loop), j is increasing
- Run-time is clearly Theta(N+M) = Theta(N) in the worst case
- Useful if we are accessing the text as a continuous stream (it is not buffered)

- Let's take a different approach:
  - We just discussed hashing as a way of efficiently accessing data
  - Can we also use it for string matching?
  - Consider the hash function we discussed for strings:

    $s[0]*B^{n-1} + s[1]*B^{n-2} + \ldots + s[n-2]*B^{1} + s[n-1]$

    – where B is some integer (31 in JDK)
    - Recall that we said that if B == number of characters in the character set, the result would be unique for all strings
    - Thus, if the integer values match, so do the strings

- Ex: if B = 32
  - h("CAT") === $67*32^2 + 65*32^1 + 84 == 70772$
  - To search for "CAT" we can thus "hash" all 3-char substrings of our text and test the values for equality
- Let's modify this somewhat to make it more useful / appropriate
  1) We need to keep the integer values of some **reasonable size**
     - Ex: No larger than an int or long value
  2) We need to be able to **incrementally update** a value so that we can progress down a text string looking for a match

- Both of these are taken care of in the Rabin Karp algorithm
  1) The hash values are calculated "mod" a large integer, to guarantee that we won't get overflow
  2) Due to properties of modulo arithmetic, characters can be "removed" from the beginning of a string almost as easily as they can be "added" to the end
     - Idea is with each mismatch we "remove" the leftmost character from the hash value and we add the next character from the text to the hash value
     - Show on board
  - Let's look at the code

```
const int q = 33554393;
const int d = 32;
int rksearch(char *p, char *a)
  {
    int i, dM = 1, h1 = 0, h2 = 0;
    int M = strlen(p), N = strlen(a);
    for (i = 1; i < M; i++) dM = (d*dM) % q;
    for (i = 0; i < M; i++)
      {
        h1 = (h1*d+index(p[i])) % q; // hash pattern
        h2 = (h2*d+index(a[i])) % q; // hash beg. of text
      }
    for (i = 0; h1 != h2; i++)
      {
        h2 = (h2+d*q-index(a[i])*dM) % q; // remove 1st
        h2 = (h2*d+index(a[i+M])) % q;    // add next
        if (i > N-M) return N;
      }
    return i;
  }
```

- The algorithm as presented in the text is not quite correct – what is missing?
  - Does not handle collisions
  - It assumes that if the hash values match the strings match – this may not be the case
  - Although with such a large "table size" a collision is not likely, it is possible
  - How do we fix this?
    - If hash values match we then compare the character values
      - > If they match, we have found the pattern
      - > If they do not match, we have a collision and we must continue the search

- Runtime?
  - Assuming no or few collisions, we must look at each character in the text at most two times
    - Once to add it to the hash and once to remove it
  - As long as are arithmetic can be done in constant time (which it can as long as we are using fixed-length integers) then our overall runtime should be **Theta(N) in the average case**
  - Note: In the worst case, the run-time is Theta(MN), just like the naïve algorithm
    - However, this case is highly unlikely
      - > Why? Discuss
  - However, we still haven't really improved on the "normal case" runtime

- What if we took yet another approach?
  - Look at the pattern from right to left instead of left to right
    - Now, if we mismatch a character early, we have the potential to skip many characters with only one comparison
    - Consider the following example:

      A = ABCDVABCDWABCDXABCDYABCDZ

      P = ABCDE
    - If we first compare E and V, we learn two things:
      1) V does not match E
      2) V does not appear anywhere in the pattern
    - How does that help us?

- We can now skip the pattern over M positions, after only one comparison
  - Continuing in the same fashion gives us a very good search time
  - Show on board
- Assuming our search progresses as shown, how many comparisons are required?

  N/M
- Will our search progress as shown?
  - Not always, but when searching text with a relatively large alphabet, we often encounter characters that do not appear in the pattern
  - This algorithm allows us to take advantage of this fact

- Details
  - The technique we just saw is the mismatched character (MC) heuristic
    - It is one of two heuristics that make up the Boyer Moore algorithm
    - The second heuristic is similar to that of KMP, but processing from right to left
  - Does MC always work so nicely?
    - No – it depends on the text and pattern
    - Since we are processing right to left, there are some characters in the text that we don't even look at
    - We need to make sure we don't "miss" a potential match

- Consider the following:

  A = XYXYXXYXYYXYXYZXYXYXXYXYYXYXYX

  P = XYXYZ

  - Discuss on board
  - Now the mismatched character DOES appear in the pattern
    - When "sliding" the pattern to the right, we must make sure not to go farther than where the mismatched character in A is first seen (from the right) in P
    - In the first comparison above, X does not match Z, but it does match an X two positions down (from the right)
      > We must be sure not to slide the pattern any further than this amount

- How do we do it?
- Preprocess the pattern to create a skip array
  - Array indexed on ALL characters in alphabet
  - Each value indicates how many positions we can skip given a mismatch on that character in the text

```
for all i skip[i] = M
for (int j = 0; j < M; j++)
    skip[index(p[j])] = M - j - 1;
```

  - Idea is that initially all chars in the alphabet can give the maximum skip
  - Skip lessens as characters are found further to the right in the pattern

```
int mischarsearch(char *p, char *a)
  {
    int i, j, t, M = strlen(p), N = strlen(a);
    initskip(p);
    for (i = M-1, j = M-1; j >= 0; i--, j--)
      while (a[i] != p[j])
        {
          t = skip[index(a[i])];
          i += (M-j > t) ? M-j : t; // if we have
                // passed more chars (r to l) than
                // t, skip that amount rather than t
          if (i >= N) return N;
          j = M-1;
        }
    return i+1;
  }
```

- Can MC ever be poor?
  - Yes
  - Discuss how and look at example
  - By itself the runtime could be Theta(NM) – same as worst case for brute force algorithm
- This is why the BM algorithm has two heuristics
  - The second heuristic guarantees that the run-time will never be worse than linear
- Look at comparison table
  - Discuss

- Why do we use compression?
  1) To save space
     - Hard drives, despite increasing size, always seem to be filled with new programs and data files
     - 3.5" floppy drives are still used and are still very low capacity
  2) To save time/bandwidth
     - Many programs and files are now downloaded from the internet
     - Most people still have relatively slow connections
     - Compressed files allow for faster transfer times, and allow more people to use a server at the same time

- Major types of compression
  1) Lossy – some data is irretrievably lost during the compression process
     - Ex: MP3, JPEG, Dolby AC-3

       •Good for audio and video applications, where the perception of the user is required

       –Gives extremely large amounts of compression, which is useful for large audio and video files

       –If the quality is degraded somewhat, user may not notice or may not care

       –Many sophisticated algorithms are used to determine what data to "lose" and how it will have the least degradation to the overall quality of the file

2) Lossless – original file is exactly reproduced when compressed file is decompressed
   - Or, D(C(X)) = X where C is the compression and D is the decompression
   - Necessary for files that must be exactly restored
     - Ex: .txt, .exe, .doc, .xls, .java, .cpp and others
   - Will also work for audio and video, but will not realize the same compression as lossy
     - However, since it works for all file types, it can be used effectively for archiving all data in a directory or in multiple directories

- Most modern lossless compression techniques have roots in 3 algorithms:
  - Huffman
    - Variable length, 1 codeword-per-character code
  - LZ77
    - Uses a "sliding" window to compress groups of characters at a time
  - LZ78 / LZW
    - Uses a dictionary to store previously seen patterns, also compressing groups of characters at a time
- We will discuss Huffman and LZW in more detail

- What is used in actual compression programs?
  - Here are a few
    - unix compress, gif: LZW
    - pkzip, zip, gzip: LZ77 + Huffman

- Background:
  - Huffman works with arbitrary bytes, but the ideas are most easily explained using character data, so we will discuss it in those terms
  - Consider extended ASCII character set:
    - 8 bits per character
    - BLOCK code, since all codewords are the same length
      - 8 bits yield 256 characters
      - In general, block codes give:
        - For K bits, 2^K characters
        - For N characters, $\lceil \log_2 N \rceil$ bits are required
    - Easy to encode and decode

- What if we could use variable length codewords, could we do better than ASCII?
  - Idea is that different characters would use different numbers of bits
  - If all characters have the same frequency of occurrence per character we cannot improve over ASCII
- What if characters had different freqs of occurrence?
  - Ex: In English text, letters like E, A, I, S appear much more frequently than letters like Q, Z, X
  - Can we somehow take advantage of these differences in our encoding?

- First we need to make sure that variable length coding is feasible
  - Decoding a block code is easy – take the next 8 bits
  - Decoding a variable length code is not so obvious
  - In order to decode unambiguously, variable length codes must meet the prefix property
    - No codeword is a prefix of any other
    - See example on board showing ambiguity if PP is not met
- Ok, so now how do we compress?
  - Let's use fewer bits for our more common characters, and more bits for our less common characters

- Huffman Algorithm:
  - **Assume we have K characters and that each uncompressed character has some weight associated with it (i.e. frequency)**
  - **Initialize a forest, F, to have K single node trees in it, one tree per character, also storing the character's weight**
  - **while (|F| > 1)**
    - **Find the two trees, T1 and T2, with the smallest weights**
    - **Create a new tree, T, whose weight is the sum of T1 and T2**
    - **Remove T1 and T2 from the F, and add them as left and right children of T**
    - **Add T to F**

- See example on board

- Huffman Issues:
  1) Is the code correct?
     - Does it satisfy the prefix property?
  2) Does it give good compression?
  3) How to decode?
  4) How to encode?
  5) How to determine weights/frequencies?

1) Is the code correct?
   - Based on the way the tree is formed, it is clear that the codewords are valid
   - Prefix Property is assured, since each codeword ends at a leaf
     - all original nodes corresponding to the characters end up as leaves

2) Does it give good compression?
   - For a block code of N different characters, $\lceil \log_2 N \rceil$ bits are needed per character
     - Thus a file containing M ASCII characters, 8M bits are needed

- Given Huffman codes $\{C_0, C_1, \ldots C_{N-1}\}$ for the N characters in the alphabet, each of length $|C_i|$
- Given frequencies $\{F_0, F_1, \ldots F_{N-1}\}$ in the file
  - Where sum of all frequencies = M
- The total bits required for the file is:
  - Sum from 0 to N-1 of ($|C_i| * F_i$)
  - Overall total bits depends on differences in frequencies
    - The more extreme the differences, the better the compression
    - If frequencies are all the same, no compression
- See example from board

3) How to decode?

- This is fairly straightforward, given that we have the Huffman tree available

```
start at root of tree and first bit of file
while not at end of file
   if current bit is a 0, go left in tree
   else go right in tree // bit is a 1
   if we are at a leaf
      output character
      go to root
   read next bit of file
```

  - Each character is a path from the root to a leaf
  - If we are not at the root when end of file is reached, there was an error in the file

4) How to encode?

- This is trickier, since we are starting with characters and outputing codewords
  - Using the tree we would have to start at a leaf (first finding the correct leaf) then move up to the root, finally reversing the resulting bit pattern
  - Instead, let's process the tree once (using a traversal) to build an encoding TABLE.
  - Demonstrate inorder traversal on board

5) How to determine weights/frequencies?

- 2-pass algorithm
  - Process the original file once to count the frequencies, then build the tree/code and process the file again, this time compressing
  - Ensures that each Huffman tree will be optimal for each file
  - However, to decode, the tree/freq information must be stored in the file
    - Likely in the front of the file, so decompress first reads tree info, then uses that to decompress the rest of the file
    - Adds extra space to file, reducing overall compression quality

- Overhead especially reduces quality for smaller files, since the tree/freq info may add a significant percentage to the file size
- Thus larger files have a higher potential for compression with Huffman than do smaller ones
  - However, just because a file is large does NOT mean it will compress well
  - The most important factor in the compression remains the relative frequencies of the characters

- Using a static Huffman tree
  - Process a lot of "sample" files, and build a single tree that will be used for all files
  - Saves overhead of tree information, but generally is NOT a very good approach

  - There are many different file types that have very different frequency characteristics
    - > Ex: .cpp file vs. .txt containing an English essay
    - > .cpp file will have many ;, {, }, (,)
    - > .txt file will have many a,e,i,o,u,., etc.
    - > A tree that works well for one file may work poorly for another (perhaps even expanding it)

- Adaptive single-pass algorithm
  - Builds tree as it is encoding the file, thereby not requiring tree information to be separately stored
  - Processes file only one time
  - We will not look at the details of this algorithm, but the LZW algorithm and the self-organizing search algorithm we will discuss next are also adaptive

- What is Huffman missing?
  - Although OPTIMAL for single character (word) compression, Huffman does not take into account patterns / repeated sequences in a file
  - Ex: A file with 1000 As followed by 1000 Bs, etc. for every ASCII character will not compress AT ALL with Huffman
    - Yet it seems like this file should be compressable
    - We can use run-length encoding in this case (see text)
      - However run-length encoding is very specific and not generally effective for most files (since they do not typically have long runs of each character)

- Consider two searching heuristics:
  - Move-to-Front (MTF)
    - Search a list for the desired key
    - When found, remove the key from its current position and place it in the front of the list
      - If the list is an array, shift other keys down
  - Transpose
    - Search a list for the desired key
    - When found, swap positions with the key before it in the list
      - In other words, "transpose" it with the key that was before it

- Idea for both heuristics is that frequently accessed keys will end up near the front of the list
- These improve search times for popular items

- How can these be used in compression?
  - Consider a list of uncompressed codewords
    - For example, single bytes 0-255
  - In original form, each requires 8 bits to represent (as we previously discussed)
  - Like Huffman, we'd like to represent more frequently used characters with fewer bits, and less frequently used characters with more bits

- General Approach
  - Maintain two arrays:
    - One is indexed on the original byte codes and is storing a position value (call this Array1)

Code Value	0	1	2	…	65	…	83	
Position	2	25	120		0		1	

> 'A' (ASCII 65) is in position 0 (i.e. at the front of the list)

    - One is indexed on the position and is storing a byte code (call this Array2)

Position	0	1	2	…	25	…	120	
Code Value	65	83	0		1		2	

> Code at position 0 is 65 ('A')

- Compression:
  - When we read in a byte / character, do the following:
    - Using Array1, find the position, pos, of the character in Array2
    - Output the correct code value, as explained in the next slides
    - Update Array2 using the correct heuristic (MTF or Transpose) and also update Array1 accordingly
  - Idea is that frequently seen characters will end up close to the front of the array
    - These will be able to be expressed with fewer bits, as explained next

- Idea of the output:
  - Initial bits will be the binary representation of a number from 0 to 7, or 3 bits
    - This is equal to the (number of bits – 1) required to represent pos in binary when leading 0s are removed
    - The –1 is necessary 3 (unsigned) bits can represent 0-7, but our number of bits is 1-8
  - Next bits are the value of pos in binary, with leading 0s removed
    - For example, the pos value 9, or 00001001, can be represented in 4 bits
  - Clearly, the positions closer to the front can be represented by fewer bits
    - This is how we achieve compression

Compression through self-organizing search

Bits Needed	(minus 1)	(in binary)	Pos	Pos in binary	Output
1	0	000	0	00000000	0000
1	0		1	00000001	0001
2	1	001	2	00000010	00110
2	1		3	00000011	00111
3	2	010	4	00000100	010100
3	2		5	00000101	010101
3	2		6	00000110	010110
3	2		7	00000111	010111
4	3	011	8	00001000	011000
			...	...	...

- Output is generated as follows:
  - > Shift out the leading (8 – Bits Needed) bits from the position in binary
  - > Append the result to the leading 3 bits (Bits Needed-1 in binary)

- Original Data: ABCABAABC
  - Original arrays shown below
  - See trace on board

Code Value	0	1	…	65	66	67	…	
Position	0	1		65	66	67		

Position	0	1	…	65	66	67	…	
Code Value	0	1		65	66	67		

- Note that for each number of bits from 2 on, ½ of the possible codewords are not used
  - Ex: 00110, 00111 are used but 00100 and 00101 are not
  - With slightly more complicated preprocessing, we can use all of these codewords
    - But we must be careful so that all possible positions can be represented and unambiguously decoded
  - This will improve our overall compression quality since it will pack more possible values into shorter codewords
- Compression quality will also improve if we consider more initial possibilities
  - Ex: Instead of 8 bits use 16 bits for uncompressed data
    - Now we have 2^{16} possible position values

- Decompression
  - We know in advance the uncompressed key size, k (ex: 8 bits or 16 bits)
  - Array is initialized just as in compression phase
    - Since we are only looking up positions, we only need one array (Array2), indexed on position
  - while not eof
    - Read in $\lg_2 k$ bits as an integer, N
    - Read in N+1 bits as an integer, pos (the position value)
    - Using Array2, find the key at location pos and output it
    - Update Array2 using MTF or Transpose
  - See example on board

- It is natural to compare the self-organizing search compression algorithm to Huffman
  - Both compress fixed size original codewords into variable length codewords
    - For a file with extreme frequency characteristics, Huffman may do better
    - However, self-organizing search handles changes in frequencies better
      - > Recall example: 1000As1000Bs1000Cs…
      - > To Huffman, all frequencies will be the same
      - > Using MTF after the first occurrence of each letter, the others will have excellent compression
- But compression is still 1-1 original codeword to compressed codeword

- Clearly, the size of the original codewords considered is important
  - With 8 bits at a time, the best compression we can do is about ½ (4/8), since we need 3 bits then at least 1 bit
  - With 16 bit original codewords, now we can achieve a "best ratio" of 5/16 since we need 4 bits then at least 1 bit
  - With 32 bit original codewords, we can do even better – 6/32
    - However, note now that we will need 2^{32} positions in our arrays – this is likely too large for us to use effectively (even if we had the memory, the run-times would be prohibitive)

- Idea of LZW:
  - Instead of using variable length codewords to encode single characters, use block codewords to to encode groups of characters
    - The more characters that can be represented by a single codeword, the better the compression
      - Ex: Consider the word "the" – 24 bits using ASCII
        - > If we can encode the entire word in one 12 bit codeword, we have cut its size in half
    - Groups are built up gradually as patterns within the file are repeated

- LZW Compression Algorithm:
  - See handout
  - See lzw.txt and notes from board
- LZW Decompression Algorithm
  - See handout
  - See lzw2.txt and notes from board

- Why / how does it work?
  - The compression and decompression algorithms are both building the EXACT SAME (codeword, string) dictionary as they proceed
    - Compression stores them as (string, codeword)
      - During compression, strings are looked up and codewords are returned
    - Decompression stores them as (codeword, string)
      - During decompression, codewords are looked up and strings are returned
    - As long as both follow the same steps, the compressed file does not need to store any extra information about the code

- This is an adaptive algorithm
  - The compression adapts to the patterns as they are seen, and the decompression does the same
- However, as we discussed, the decompression algorithm is one step "behind" the compression algorithm in building the dictionary
  - In most situations this is not a problem
  - However, if, during compression, the (pattern, codeword) that was just added to the dictionary is immediately used in the next step, the decompression algorithm will not yet know the codeword
  - Luckily this special case can be recognized and handled relatively easily
    - See lzw3.txt

1) How to represent / manage the dictionary
2) How many bits to use for the codewords
3) What to do when / if we run out of codewords
4) How to do I/O with fractions of bytes
   - This issue applies to Huffman as well, so discussion here will be applicable there

1) How to represent / manage the dictionary

- What operations do we need?
  - Insert and Lookup
- For a file with M characters, we will need to do M Lookups
  - Number of Inserts depends on how long our patterns get
  - Thus we want these to be VERY FAST
    - Sorted Array takes much too long for Inserts
    - BST would be ok, but even lgM per Lookup is probably more time than we want to spend
      - Would yield total of Theta(MlgM) total time
    - Do we have any better options?

- Two most likely candidates for ENCODING dictionary:
  - Trie or Hash Table
  - Both allow lookups in time proportional to string length, independent of the number of strings
  - Trie insert is a bit tricky, but if we use a DLB, it is not too bad
- Consider implementation done in lzw.c
  - Hash table used, but in a clever way
  - Instead of storing the entire string each time (wasting a lot of space), they store the codeword for the "prefix" and the last character
    - > This works because strings are built one character at a time, always adding on the strings already in the dictionary
  - See code and board

- For DECODING, the idea is even easier
  - Now codewords are the key values, and the strings are returned
  - We don't need to hash anything
  - We can simply use the codeword values to index an array of strings
    - > In lzw.c it is not actually the string but rather the prefix code, last character pair in the same way as for encoding
    - > See code

2) How many bits to use for the codewords

- Fewer bits:
  - Smaller codewords, giving compression EARLIER in the process
  - Fewer available codewords, limiting the compression LATER in the process
- More bits:
  - Larger codewords, delaying actual compression until longer patterns are found
  - More available codewords, allowing for greater compression LATER in the process
- Ex: Consider 10 bits vs. 16 bits

- Can we get the "best of both worlds"?
  - We'd like to use fewer bits earlier in the process, to get compression sooner
  - We'd like to use more bits later in the process, to get greater compression later in the file
  - In fact this is exactly what the Unix compress algorithm does
    - It starts out using 9 bits for codewords, adding an extra bit when all codewords for previous size are used. Ex:
      - 9 bits for codewords 0-511
      - 10 bits for codewords 512-1023
      - 11 bits for codewords 1024-2047
      - etc
    - Decompress does the same so it works!

3) What to do when / if we run out of codewords

- If we use a block code of a specific size, we have a finite number of codewords that we can represent
  - Even the "compress" version would eventually stop adding bits due to I/O issues (we will discuss next)
- When all codewords have been used, what do we do?

- Two primary options, each with advantages and disadvantages:
  - Keep compressing as before, but simply stop adding new entries to the dictionary
    - > Adv: Maintains long patterns that have already been built up
    - > Disadv: If file content changes (with new patterns) those will not be compressed effectively
  - Throw out entire dictionary, then start again with the single characters
    - > Adv: Allows new patterns to be compressed
    - > Disadv: Until new patterns are built and added to the dictionary, compression is minimal

4) How to do I/O with fractions of bytes

- Unless we pick an exact multiple of a byte for our codeword size (8, 16, 24, 32 bits) we will need to input and output fractions of bytes
- We will not actually input / output fractions of bytes
  - Rather we will keep a buffer and read / write exact numbers of bytes, processing the necessary bits from the buffer
  - This involves some bit operations to be done
    - Shifting, bitwise OR, etc.
- See lzw.c – go to recitation!

- In practice, which is better, LZW or Huffman?
  - For most files, LZW gives better compression ratios
  - It is also generally better for compressing archived directories of files
  - Why?

    •Files can build up very long patterns, allowing LZW to get a great deal of compression

    •Different file types do not "hurt" each other with LZW as they do with Huffman – with each type we simply have to build up new patterns

- Let's compare
  - See compare.txt handout
  - Note that for a single text file, Huffman does pretty well
  - For large archived file, Huffman does not as well
  - gzip outperforms Huffman and LZW
    - Combo of LZ77 and Huffman
    - See: http://en.wikipedia.org/wiki/Gzip
      http://www.gzip.org/
      http://www.gzip.org/algorithm.txt

- How much can we compress a file (in a lossless way)?
  - Ex: Take a large file, and recursively compress it K times
    - If K is large enough, maybe I can compress the entire file down to 1 bit!
  - Of course this won't work, but why not?
    - Clearly, we cannot unambiguously decompress this file – we could make an infinite number of "original" files

- Generally speaking, the amount we can compress a file is dependent upon the amount of **entropy** in the data
  - Informally, entropy is the amount of uncertainty / randomness in the data
    - Information entropy is very similar in nature to thermodynamic entropy, which you may have discussed in a physics or chemistry class
  - The more entropy, the less we can compress, and the less entropy the more we can compress
    - Ex: File containing all A's can be heavily compressed since all of the characters are the same
    - Ex: File containing random bits cannot be compressed at all

- When we compress a file (ex: using compress or gzip) we are taking patterns / repeated sequences and substituting codewords that have much more entropy
- Attempts to compress the result (even using a different algorithm) may have little or no further compression
  - However, in some cases it may be worth trying, if the two algorithms are very different
- For more info, see:
- http://en.wikipedia.org/wiki/Lossless_data_compression
- http://en.wikipedia.org/wiki/Information_entropy

- Integer Multiplication
  - With predefined int variables we think of multiplication as being Theta(1)
    - Is the multiplication really a constant time op?
    - No -- it is constant due to the constant size of the numbers (32 bits), not due to the algorithm
  - What if we need to multiply very large ints?
    - Ex: RSA that we will see shortly needs ints of sizes up to 2048 bits
  - Now we need to think of good integer multiplication algorithms

- GradeSchool algorithm:
  - Multiply our integers the way we learn to multiply in school
    - However, we are using base 2 rather than base 10
    - See example on board
  - Run-time of algorithm?
    - We have two nested loops:
      - >Outer loop goes through each bit of first operand
      - >Inner loop goes through each bit of second operand
    - Total runtime is Theta(N^2)
  - How to implement?
    - We need to be smart so as not to waste space
    - The way we learn in school has "partial products" that can use Theta(N^2) memory

- Instead we can calculate the answer column by column, saving a lot of memory
- Let's look at base-10 pseudocode (base 2 is similar)

```
function Mult (X[0..n-1], Y[0..n-1]) returns Z[0..2n-1]
   // note: X and Y have n digits Z has UP TO (2n) digits
   // The LEAST significant digits (smaller powers of ten)
   // stored in the smaller indices
      long  S = 0
      for j from 0 to (2n-1) do  // go through digits of result
          for  i  from  0  to  (n-1)  do    // digits of X
                  if  (0 <= (j-i) <= (n-1))  then
                       S = S + X[i] * Y[j-i]
          Z[j]  =  S  mod  10   // remainder to get curr digit
          S  =  S / 10          // integer division for carry
      return  Z
end function
```

- Can we improve on Theta(N^2)?
  - How about if we try a divide and conquer approach?
  - Let's break our N-bit integers in half using the high and low order bits:

$$X = 1001011011001000$$

$$= 2^{N/2}(X_H) + X_L$$

where X_H = high bits = 10010110

X_L = low bits = 11001000

- Let's look at this in some more detail
  - Recall from the last slide how we break our N-bit integers in half using the high and low order bits:

$$X = 1001011011001000$$
$$= 2^{N/2}(X_H) + X_L$$
$$\text{where } X_H = \text{high bits} = 10010110$$
$$X_L = \text{low bits} = 11001000$$

  - Given two N-bit numbers, X and Y, we can then re-write each as follows:

$$X = 2^{N/2}(X_H) + X_L$$
$$Y = 2^{N/2}(Y_H) + Y_L$$

- Now, the product, X*Y, can be written as:

$$XY = (2^{N/2}(X_H) + X_L) * (2^{N/2}(Y_H) + Y_L)$$
$$= \mathbf{2^N X_H Y_H + 2^{N/2}(X_H Y_L + X_L Y_H) + X_L Y_L}$$

- Note the implications of this equation
  - The multiplication of 2 N-bit integers is being defined in terms of
    - 4 multiplications of N/2 bit integers
    - Some additions of ~N bit integers
    - Some shifts (up to N positions)
  - But what does this tell us about the overall runtime?

- How to analyze the divide and conquer algorithm?
  - Analysis is more complicated than iterative algorithms due to recursive calls
  - For recursive algorithms, we can do analysis using a special type of mathematical equation called a Recurrence Relation
    - Idea is to determine two things for the recursive calls
      1) How much work is to be done during the current call, based on the current problem size?
      2) How much work is "passed on" to the recursive calls?

- Let's examine the recurrence relation for the divide and conquer multiplication algorithm
  - We will assume that the integers are divided exactly in half at each recursive call
    - Original number of bits must be a power of 2 for this to be true

1) Work at the current call is due to shifting and binary addition. For an N-bit integer this should require operations proportional to N
2) Work "passed on" is solving the same problem (multiplication) 4 times, but with each of half the original size
   - $X_H Y_H$, $X_H Y_L$, $X_L Y_H$, $X_L Y_L$

- So we write the recurrence as

T(N) = 4T(N/2) + Theta(N)

  - Or, in words, the operations required to multiply 2 N-bit integers is equal to 4 times the operations required to multiply 2 N/2-bit integers, plus the ops required to put the pieces back together

- If we emphasized analysis in this course, we would now learn how to solve this recurrence
  - We want a Theta run-time in direct terms of N – i.e. we want to remove the recursive component
  - There are a number of techniques that can accomplish this

- However, for our purposes we will just state that the solution is Theta(N^2)
  - Note that this is the SAME run-time as Gradeschool
  - Further, the overhead for this will likely make it slower overall than Gradeschool
  - So why did we bother?
    - > If we think about it in a more "clever" way, we can improve the solution so that it is in fact better than Gradeschool

- Karatsuba's Algorithm
  - If we can **reduce** the number of **recursive calls** needed for the divide and conquer algorithm, perhaps we can improve the run-time
    - How can we do this?
    - Let's look at the equation again

$$XY = \underbrace{2^N X_H Y_H}_{(M_1)} + \underbrace{2^{N/2}(X_H Y_L + X_L Y_H)}_{(M_2)\ (M_3)} + \underbrace{X_L Y_L}_{(M_4)}$$

    - Note that we don't really NEED M_2 and M_3
      - All we need is the SUM OF THE TWO, $M_2 + M_3$
    - If we can somehow derive this sum using only one rather than two multiplications, we can improve our overall run-time

- Now consider the following product:

$$(X_H + X_L) * (Y_H + Y_L) = X_HY_H + X_HY_L + X_LY_H + X_LY_L$$

- Using our M values from the previous slide, this equals

$$M_1 + M_2 + M_3 + M_4$$

- The value we want is $M_2 + M_3$, so define M_{23}

$$M_{23} = (X_H + X_L) * (Y_H + Y_L)$$

- And our desired value is $M_{23} - M_1 - M_4$
- Ok, all I see here is wackiness! How does this help?
  - Let's go back to the original equation now, and plug this back in

$$XY = 2^N X_H Y_H + 2^{N/2}(X_H Y_L + X_L Y_H) + X_L Y_L$$

$$= 2^N M_1 + 2^{N/2}(M_{23} - M_1 - M_4) + M_4$$

- Only 3 distinct multiplications are now needed!

- Looking back, we see that M_{23} involves multiplying at most (N/2)+1 bit integers, so asymptotically it is the same size as our other recursive multiplications
- We have to do some extra additions and two subtractions, but these are all Theta(N) operations
- Thus, we now have the following recurrence:
- T(N) = 3T(N/2) + Theta(N)
- This solves to Theta(N^{lg3}) ≈ **Theta($N^{1.58}$)**
  - Now we have an asymptotic improvement over the Gradeschool algorithm
  - Still a lot of overhead, but for large enough N it will run faster than Gradeschool
  - See karat.txt

- Can we do even better?
  - If we multiply the integers indirectly using the Fast Fourier Transform (FFT), we can achieve an even better run-time of Theta(NlgN)
  - Don't worry about the details of this algorithm
    - But if you are interested, read the text Ch. 41
- For RSA, we will be ok using a GradeSchool / Karatsuba Hybrid
  - GradeSchool used for smaller sizes, switching to Karatsuba for larger sizes
    - Karatsuba has a lot of overhead, so GS is better for smaller number of bits

- How about integer powers: X^Y
  - Natural approach: simple for loop

```
ZZ ans = 1;
for (ZZ ctr = 1; ctr <= Y; ctr++)
     ans = ans * X;
```

  - This seems ok – one for loop and a single multiplication inside – is it linear?
  - Let's look more closely
    - Total run-time is
      1) Number of iterations of loop *
      2) Time per multiplication

- We already know 2) since we just did it
  - Assuming GradeSchool, Theta(N^2) for N-bit ints
- How about 1)
  - It seems linear, since it is a simple loop
  - In fact, it is LINEAR IN THE VALUE of Y
  - However, our calculations are based on N, the NUMBER OF BITS in Y
  - What's the difference?
    - > We know an N-bit integer can have a value of up to $\approx 2^N$
    - > So linear in the value of Y is exponential in the bits of Y
  - Thus, the iterations of the for loop are actually Theta(2^N) and thus our total runtime is Theta($N^2 2^N$)
- This is RIDICULOUSLY BAD
    - > Consider N = 512 – we get $(512)^2(2^{512})$
    - > Just how big is this number?

- Let's calculate in base 10, since we have a better intuition about size
- Since every 10 powers of 2 is approximately 3 powers of ten, we can multiply the exponent by 3/10 to get the base 10 number
- So $(512)^2(2^{512}) = (2^9)^2(2^{512}) = 2^{530} \approx 10^{159}$
- Let's assume we have a 1GHz machine (10^9 cyc/sec)
- This would mean we need 10^{150} seconds
- $(10^{150}\text{sec})(1\text{hr}/3600\text{sec})(1\text{day}/24\text{hr})(1\text{yr}/365\text{days}) = (10^{150}/(31536000))$ years $\approx 10^{150}/10^8 \approx 10^{142}$ years
- This is ridiculous!!

- But we need exponentiation for RSA, so how can we do it more efficiently?

- How about a divide and conquer algorithm
  - Divide and conquer is always worth a try
- Consider

  $X^Y = (X^{Y/2})^2$ when Y is even

  how about when Y is odd?

  $X^Y = X * (X^{Y/2})^2$ when Y is odd
- Naturally we need a base case

  $X^Y = 1$ when $Y = 0$
- We can easily code this into a recursive function
- What is the run-time?

- Let's see…our problem is to calculate the exponential X^Y for X and Y
  - Let's first calculate the answer in terms of the value of Y, then we will convert to bits later
  - So we have a recursive call with an argument of ½ the original size, plus a multiplication (again assume we will use GradeSchool)
    - We'll also put the multiplication time back in later
    - For now let's determine the number of function calls
  - How many times can we divide Y by 2 until we get to a base case?

$$\lg(Y)$$

- So we have lg(Y) recursive calls
- But remember that Y can have a value of up to 2^N
- Converting back to N, we have

  $lg(Y) = lg(2^N) = N$
- Since we have one or two multiplications per call, we end up with a total runtime of

  $Theta(N^2*N) = Theta(N^3)$

- This is an AMAZING improvement
  - Consider again N = 512
  - $N^3 = 134217728$ – less than a billion
  - On a 1GHz machine this would take less than a second (assuming one operation per cycle – in reality it may take a few seconds)

- Can we improve even more?
  - Well removing the recursion can always help
  - If we start at X, then square repeatedly, we get the same effect as the recursive calls
  - Square at each step, and also multiply by X if there is a 1 in the binary representation of Y (from left to right)
  - Ex: $X^{45} = X^{101101} =$

    $1, X \quad X^2 \quad X^4, X^5 \quad X^{10}, X^{11} \quad X^{22} \quad X^{44}, X^{45}$

    1 0 1 1 0 1
  - Same idea as the recursive algorithm but building from the "bottom up"
  - See Text p. 526

- GCD(A, B)
  - Largest integer that evenly divides A and B
    - i.e. there is no remainder from the division
  - Simple, brute-force approach?
    - Start with min(A,B) since that is the largest possible answer
    - If that doesn't work, decrement by one until the GCD is found
    - Easy to code using a simple for loop
  - Run-time?
    - We want to count how many mods we need to do
    - **Theta(min(A, B))** – is this good?

- Remember exponentiation?
- A and B are N bits, and the loop is linear in the VALUE of min(A,B)
- This is exponential in N, the number of bits!

How can we improve?

- Famous algorithm by Euclid
- GCD(A,B) = GCD(B, A mod B)
- Ok let's try it
  - GCD(30,24) = GCD(24,6) = GCD(6,0) = ?????
- What is missing?

The base case: **GCD(A,B) = A when B = 0**

Now GCD(6,0) = 6 and we are done

- Run-time of Euclid's GCD?
  - Let's again count number of mods
  - Tricky to analyze exactly, but in the worst case it has been shown to be linear in N, the number of bits
  - Similar improvement to exponentiation problem
  - Also can be easily implemented iteratively
    - See handout gcd.txt
- Extended GCD
  - It is true that GCD(A,B) = D = AS + BT for some integer coefficients S and T
    - Ex: GCD(30,24) = 6 = (30)(1) + (24)(-1)
    - Ex: GCD(99,78) = 3 = (99)(-11) + (78)(14)

- With a bit of extra logic (same Theta run-time), GCD can also provide the coefficients S and T
- This is called the Extended Greatest Common Divisor algorithm

- Arithmetic summary
  - We have looked at multiplication, exponentiation, and GCD (XGCD)
  - These will all be necessary when we look at public key encryption next

- What is Cryptography?
  - Designing of secret communications systems
    - A **SENDER** wants a **RECEIVER** (and no one else) to understand a **PLAINTEXT** message
    - Sender **ENCRYPTS** the message using an *encryption algorithm* and some *key parameters*, producing **CIPHERTEXT**
    - Receiver has *decryption algorithm* and key parameters, and can restore original plaintext

- Why so much trouble?
  - **CRYPTANALYST** would like to read the message
    - Tends to be very clever
    - Can usually get a copy of the ciphertext
    - Usually knows (or can figure out) the *encryption algorithm*
  - So the **key parameters** (and how they affect decryption) are really the only thing preventing the cryptanalyst from decrypting
    - And he/she hopes to decrypt your message without them (or possibly figure them out in some way)

- Early encryption schemes were quite simple
  - Ex. Caesar Cipher
    - Simple shift of letters by some integer amount
    - Key parameter is the shift amount

ABCDEFGHIJKLMNO . . . <- original alphabet

ABCDEFGHIJKLMNOPQR . . . <- letters used in code

shift = 3

KHOOR <- ciphertext

- Almost trivial to break today
  - Try each shift until right one is found
    - Only 25 possible shifts for letters, 255 for ASCII
- Try arbitrary substitution instead: Substitution Cipher
  - Code alphabet is an arbitrary permutation of original alphabet
    - Sender and receiver know permutation, but cryptanalyst does not
  - Much more difficult to break by "guessing"
    - With alphabet of S characters, S! permutations

- But still relatively easy to break today in other ways
  - Frequency tables, sentence structure
  - Play "Wheel of Fortune" with the ciphertext
  - Once we guess some letters, others become easier
    - Not a trivial program to write, by any means
    - But run-time is not that long – that is the important issue

Hey Pat, can I buy a vowel?

These "before and after"s always give me trouble

- Better if "structure" of ciphertext differs from that of plaintext
  - Instead of substitution, "add" a key value to the plaintext

```
ALGORITHMS ARE COOL   <- original message
ABCDABCDABCDABCDABC   <- key sequence
--------------------
BNJSSKWLNUCESGCDPQO   <- ciphertext
```

- Vigenere Cipher
  - Now the same ciphertext character may represent more than one plaintext character
  - The longer the key sequence, the better the code
    - If the key sequence is as long as the message, we call it a Vernam Cipher
  - This is effective because it makes the ciphertext appear to be a "random" sequence of characters
    - The more "random", the harder to decrypt

- Vernam Cipher is provably secure for one-time use (as long as key is not discovered)
  - Since a "random" character is added to each character of the message, the ciphertext appears to be completely random to the cryptanalyst
  - However, with repeated use its security diminishes somewhat
  - Used in military applications when absolute security is required
    - See

http://www.pro-technix.com/information/crypto/pages/vernam_base.html

- Variations and combinations of this technique are used in some modern encryption algos
  - Ex. 3DES, Lucifer (BLOCK CIPHERS)
- These algorithms can be effective, but have a **KEY DISTRIBUTION** problem
  - How can key pass between sender and receiver securely?
    - Problem is recursive (must encrypt key; must encrypt key to decrypt key, etc).
  - How can we prevent multiple sender/receivers from reading each other's messages?
    - Need a different key for each user

- Solution is **PUBLIC-KEY** cryptography
  - Key has two parts
    - A **public** part, used for encryption
      - Everyone can know this
    - A **private** part, used for decryption
      - Only receiver should know this
  - Solves distribution problem
    - Each receiver has his/her own pair of keys
    - Don't care who knows public part
    - Since senders don't know private part, they can't read each other's messages
- **RSA** is most famous of these algorithms

- How/why does RSA work?

  Let E = encryption (public) key operation

  Let D = decryption (private) key operation

  1) D(E(plaintext)) = plaintext
     - E and D operations are inverses
  2) All E, D pairs must be distinct
  3) Knowing E, it must be VERY difficult to determine D (exponential time)
  4) E and D can be created and used in a reasonable amount of time (polynomial time)

- Theory?
  - Assume plaintext is an integer, M
    - **C** = ciphertext **= M^E mod N**
      - So E simply is a power
      - We may need to convert our message into an integer (or possibly many integers) first, but this is not difficult – we can simply interpret each block of bits as an integer
    - **$M = C^D$ mod N**
      - D is also a power
    - Or **M^{ED} mod N = M**
  - So how do we determine E, D and N?

- Not trivial by any means
  - This is where we need our extremely large integers – 512, 1024 and more bits
- Process
  - Choose random prime integers X and Y
  - Let N = XY
  - Let PHI = (X-1)(Y-1)
  - Choose another random prime integer (less than PHI and relatively prime to PHI) and call this E
  - Now all that's left is to calculate D
    - We need some number theory to do this
    - We will not worry too much about the theory

- We must calculate D such that ED mod PHI = 1
- This is equivalent to saying ED = 1 + K(PHI), for some K or, rearranging, that 1 = ED – K(PHI) or 1 = (PHI)(-K) + ED
  - Luckily we can solve this using XGCD
  - We know already that GCD(PHI, E) = 1
    - Why?
  - We also know, using number theory, that GCD(PHI,E) = 1 = (PHI)S + (E)T for some S and T
  - XGCD calculates values for S and T
  - We don't really care about S
  - But T is the value D that we need to complete our code!
- Let's look at an example

X = 7, Y = 11 (random primes)

XY = N = 77

(X-1)(Y-1) = PHI = 60

E = 37 (random prime < PHI)

Solve the following for D:

37D mod 60 = 1

Converting to form from prev. slides and solve using XGCD:

GCD(60,37) = 1 = (60)(-8)+(37)(13)

So D = 13

- $C = M^{37} \bmod 77$
- $M = C^{13} \bmod 77$

- In order for RSA to be useable
  1) The keys must be able to be generated in a reasonable (polynomial) amount of time
  2) The encryption and decryption must take a reasonable (polynomial) amount of time
  3) Breaking the code must take an extremely large (exponential) amount of time
  - Let's look at these one at a time

1) What things need to be done to create the keys and how long will they take?

- Long integer multiplication
  - We know this takes Theta(N^2), Theta($N^{1.58}$) or Theta(NlgN) depending on the algorithm
- Mod
  - Similar to multiplication
- GCD and XGCD
  - Worst case ≈ N mods
- What is left?
- Random Prime Generation

- General algorithm:
  - It turns out that the best (current) algorithm is a probabalistic one:

```
Generate random integer X
while (!isPrime(X))
      Generate random integer X
```

  - As an alternative, we could instead make sure X is odd and then add 2 at each iteration, since we know all primes other than 2 itself are odd
    - The distribution of primes generated in this way is not as good, but for purposes of RSA it should be fine

- Runtime can be divided into two parts:
  1) How many iterations of the loop are necessary (or, how many numbers must be tried)?
  2) How long will it take to test the primality of a given number?
     - Overall run-time of this process is the product of 1) and 2)

  1) Based on the distribution of primes within all integers, it is likely that a prime will be found within $\ln(2^N)$ random picks
     - This is linear in N, so it is likely that the loop will iterate N or fewer times

2) This is much more difficult: How to determine if a number is prime or composite?
   - Brute force algorithm: Try all possible factors
   - Well not actually ALL need to be tried:
     > From 2 up to square root of X
     > But X is an N bit number, so it can have a value of up to 2^N
     > This means that we will need to test up to $2^{N/2}$ factors, which will require an excessive amount of time for very large integers
     > If mod takes Theta(N^2), our runtime is now Theta($N^2 2^{N/2}$) – very poor!
   - Is there a better way? Let's use a probabilistic algorithm

- **Miller-Rabin Witness algorithm**:

```
Do K times
   Test a (special) random value (a "witness")
   If it produces a special equality – return true
If no value produces the equality – return false
```

  - If true is ever returned – number is definitely NOT prime, since a factor was found
  - If false is returned all K times, probability that the number is NOT prime is 2^{-K}
    - > If we choose K to be reasonably large, there is very little chance of an incorrect answer
- Run-time?
  - Each test requires $\approx$ N multiplications
  - Assuming GS algorithm
    - > KN^3 in the worst case

- Multiplying 1) and 2) as we stated we get
  - $N * KN^3 = KN^4$ if Gradeschool mult is use
  - This is not outstanding, since N^4 grows quickly, but in reality the algorithm will usually run much more quickly:
    - A better multiplication alg. (Karatsuba or FFT) can be used
    - A prime may be found in fewer than N tries
    - The Miller-Rabin test may find a factor quickly for some of the numbers, not requiring the full K tests

2) How long does it take to use RSA encryption and decryption?

- Power-mod operation – raising a very large integer to a very large integer power mod a very large integer
  - Same run-time as the regular power function, and can be done in the same way using the divide and conquer approach
  - Requires Theta(N) multiplications, for a total of Theta(N^3) assuming Gradeschool is used
- This is the dominant time for both encryption and decryption

3) How easily can RSA be broken?

- Recall that we have 3 values: E, D and N
  - Most obvious way of breaking RSA is to factor N
    - Factoring N gives us X and Y (since N = XY)
    - But PHI = (X-1)(Y-1)
      - Once he/she knows PHI, cryptanalyst can determine D in the same way we generated D
    - So is it feasible to factor N?
      - There is no known polynomial-time factoring algorithm
      - Thus, it will require exponential time to factor N
      - But, with fast computers, it can be done
        - > Team from Germany factored 200-decimal digit RSA number last year – but computation was considerable

Time to Break RSA

- However, if we make N large enough, we can be pretty sure that factoring will not be possible
- RSA Security recommends 768 bits for moderate security, 1024 bits for normal corporate use and 2048 bits for extremely valuable keys that may be targeted
- An important thing to remember, is that since the factoring time is still exponential, doubling the size of the key will increase the time to factor it by many thousands/millions/billions of times
  - > Yet no one has proven that factoring requires exponential time (although it is widely believed). If someone finds a polynomial factoring algorithm, RSA would be useless!
- But we should still be careful in choosing keys, and in how long we keep a key
  - > See RSA Security FAQ for more info

- Indirectly breaking a code:
  - If we just want to break one message, and not the keys, there are simple tricks that can be tried
    - For example, say you know all of the possible messages that may be sent
      - > Ex: All possible credit card numbers
      - > Encrypt each one and compare result to the ciphertext
      - > The one that matches is the message sent
    - We can combat this by padding messages with extra bits (random is best) before sending them
- Other techniques exist as well
- Generally, if we are careful and use large keys, RSA is quite secure

- Applications
  - Regular data encryption, as we discussed
    - But RSA is relatively slow compared to block ciphers
    - Block ciphers are faster, but have key-distribution problem
    - How can we get the "best of both"?
  - RSA (Digital) Envelope
    - Sender encrypts message using block cipher
    - Sender encrypts block cipher key using RSA
    - Sender sends whole package to receiver
    - Receiver decrypts block cipher key using RSA
    - Receiver uses key to decrypt rest of message

Digital Signatures

- Notice that E and D are inverses
- It doesn't matter mathematically which we do first
- If I DECRYPT my message (with signature appended) and send it, anyone who then ENCRYPTS it can see the original message and verify authenticity
  - Mostly, but there are a few issues to resolve

- Note in this case that we are NOT trying to keep anyone from reading the message
  - Everyone can get the public key and can thus successfully "encrypt" the message
- What about the issues to resolve?
  1) As with regular encryption, RSA used in this way is a bit slow
  - Can we avoid decrypting and encrypting the entire message?
    - > Yes, instead of decrypting the whole message, the sender processes the message using a technique similar to hashing (but more complex). This produces a "signature" for the message and the signature is what we actually "decrypt" using the private key. Then the message and signature are sent to the receiver.

- The receiver "encrypts" the decrypted signature to restore the original signature. The receiver then runs the same processing algorithm on the message, and compares the result with the signature. If they match, the message is valid – otherwise it has been tampered with

2) How do we know that the key used actually belonged to the sender?
   - Ex: Joe Schmoe gets fired by his boss and sends a digitally signed message stating that his boss is a total weasel. Before getting the message, his boss decides to rehire Joe, but then sees the message. Joe says "well, I didn't send that message – I haven't used that RSA key for years – it must be from someone pretending to be me"
   - How does the boss know if Joe is lying?
   - We need authentication of keys

- Graph: G = (V, E)
  - Where V is a set of vertices and E is a set of edges connecting vertex pairs
  - Used to model many real life and computer-related situations
    - Ex: roads, airline routes, network connections, computer chips, state diagrams, dependencies
  - Ex:

    V = {A, B, C, D, E, F}

    E = {(A,B), (A,D), (A,E), (A,F), (B,C), (C,D), (C, F), (D, E)

    - This is an undirected graph

- Undirected graph
  - Edges are unordered pairs: (A,B) == (B,A)
- Directed graph
  - Edges are ordered pairs: (A, B) != (B,A)
- Adjacent vertices, or neighbors
  - Vertices connected by an edge

- Let v = |V| and e = |E|
  - What is the relationship between v and e?
  - Let's look at two questions:

    1) Given v, what is the minimum value for e?

    2) Given v, what is the maximum value for e?

1) Given v, minimum e?
   - Graph definition does not state that any edges are required: 0
2) Given v, maximum e? (graphs with max edges are called complete graphs)
   - Directed graphs
     - Each vertex can be connected to each other vertex
     - "Self-edges" are typically allowed
       - Vertex connects to itself – used in situations such as transition diagrams
     - v vertices, each with v edges, for a total of **v^2 edges**

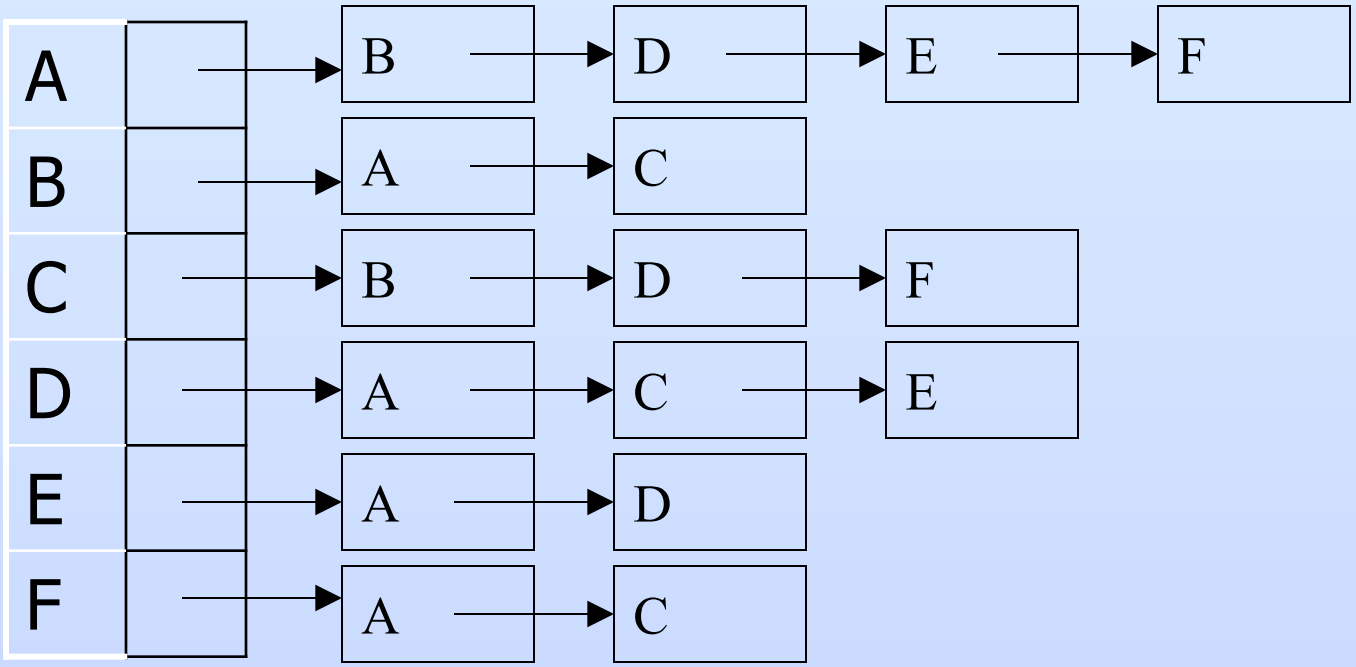
- Undirected graphs
  - "Self-edges" are typically not allowed
  - Each vertex can be connected to each other vertex, but (i,j) == (j,i) so the total edges is ½ of the total number of vertex pairs
  - Assuming v vertices, each can connect to v-1 others
    - This gives a total of (v)(v-1) vertex pairs
    - But since (i,j) == (j,i), the total number of edges is (v)(v-1)/2
- If $e <= v \lg v$, we say the graph is SPARSE
- If $e \approx v^2$, we say the graph is DENSE

- Representing a graph on the computer
  - Most often we care only about the connectivity of the graph
    - Different representations in space of the same vertex pairs are considered to be the same graph
  - Two primary representations of arbitrary graphs
    - Adjacency Matrix
      - Square matrix labeled on rows and columns with vertex ids
      - M[i][j] == 1 if edge (i,j) exists
      - M[i][j] == 0 otherwise

- Plusses:
  - Easy to use/understand
  - Can find edge (i,j) in Theta(1)
  - M^P gives number of paths of length P
- Minuses:
  - Theta(v^2) memory, regardless of e
  - Theta(v^2) time to initialize
  - Theta(v) to find neighbors of a vertex

	A	B	C	D	E	F
A	0	1	0	1	1	1
B	1	0	1	0	0	0
C	0	1	0	1	0	1
D	1	0	1	0	1	0
E	1	0	0	1	0	0
F	1	0	1	0	0	0

- Adjacency List
  - Array of linked lists
  - Each list [i] represents neighbors of vertex i



- Plusses:
  - Theta(e) memory
    - For sparse graphs this could be much less than v^2
  - Theta(d) to find neighbors of a vertex
    - d is the degree of a vertex (# of neighb)
    - For sparse graphs this could be much less than v
- Minuses
  - Theta(e) memory
    - For dense graphs, nodes will use more memory than simple location in adjacency matrix
  - Theta(v) worst case to find one neighbor
    - Linked list gives sequential access to nodes – neighbor could be at end of the list

- Overall
  - Adjacency Matrix tends to be better for dense graphs
  - Adjacency List tends to be better for sparse graphs

- Path: A sequence of adjacent vertices
- Simple Path: A path in which no vertices are repeated
- Simple Cycle: A simple path except that the last vertex is the same as the first
- Connected Graph: A graph in which a path exists between all vertex pairs
  - Connected Component: connected subgraph of a graph
- Acyclic Graph: A graph with no cycles
- Tree: A connected, acyclic graph
  - Has exactly v-1 edges

- How to traverse a graph
  - Unlike linear data structures, it is not obvious how to systematically visit all vertices in a graph
  - Two famous, well-known traversals
    - Depth First Search (DFS)
      - Visit deep into the graph quickly, branching in other directions only when necessary
    - Breadth First Search (BFS)
      - Visit evenly in all directions
      - Visit all vertices distance i from starting point before visiting any vertices distance i+1 from starting point

- DFS is usually done recursively
  - Current node recursively visits first unseen neighbor
  - What if we reach a "dead-end" (i.e. vertex with no unseen neighbors)?
    - Backtrack to previous call, and look for next unseen neighbor in that call
  - See code in graphs graphs1.txt
  - See example in graph.doc
  - See trace on board from in-class notes

- BFS is usually done using a queue
  - Current node puts all of its unseen neighbors into the queue
    - Vertices that have been seen but not yet visited are called the **FRINGE**
      - For BFS the fringe is the vertices in the queue
  - Front item in the queue is the next vertex to be visited
  - Algorithm continues until queue is empty
  - See graphs1.txt and graph.doc
  - See trace on board from in-class example

- Both DFS and BFS
  - Are initially called by a "search" function
    - If the graph is connected, the search function will call DFS or BFS only one time, and (with a little extra code) a SPANNING TREE for the graph is built
    - If the graph is not connected, the search function will call DFS or BFS multiple times
      - First call of each will terminate with some vertices still unseen, and loop in "search" will iterate to the next unseen vertex, calling visit() again
      - Each call (with a little extra code) will yield a spanning tree for a connected component of the graph

- Run-times?
  - DFS and BFS have the same run-times, since each must look at every edge in the graph twice
    - During visit of each vertex, all neighbors of that vertex are considered, so each edge will be considered twice
  - However, run-times differ depending upon how the graphs are represented
    - Recall that we can represent a graph using either an adjacency matrix or an adjacency list

- Adjacency Matrix
  - For a vertex to consider all of its neighbors we must traverse a row in the matrix -> Theta(v)
  - Since all vertices consider all neighbors, the total run-time is Theta(v^2)
- Adjacency List
  - For a vertex to consider all of its neighbors we must traverse its list in the adjacency list array
    - Each list may vary in length
    - However, since all vertices consider all neighbors, we know that all adjacency list nodes are considered
    - This is 2e, so the run-time here is Theta(v + e)
      - > We need the v in there since e could be less than v (even 0). In this case we still need v time to visit the nodes

- A biconnected graph has at least 2 distinct paths (no common edges or vertices) between all vertex pairs
  - Idea: If one path is disrupted or broken, there is always an alternate way to get there
- Any graph that is not biconnected has one or more articulation points
  - Vertices, that, if removed, will separate the graph
  - See on board and bicon.html for example
- Any graph that has no articulation points is biconnected
  - Thus we can determine that a graph is biconnected if we look for, but do not find any articulation points

- How to find articulation points?
  - Variation of DFS
  - Consider graph edges during DFS
    - Tree Edge: edge crossed when connecting a new vertex to the spanning tree
    - Back Edge: connects two vertices that were already in the tree when it was considered
      - These back edges are critical in showing biconnectivity
      - These represent the "alternate" paths between vertices in the graph
      - Will be used in finding articulation points

- Consider now the DFS spanning tree in a "top-to-bottom" way
  - Root is the top node in the tree, with DFS # = 1
  - Every vertex "lower" in the tree has a DFS number that is larger than the vertices that are "higher" in the tree (along path back to the root)
- Given this DFS spanning tree
  - A vertex, X is not an articulation point if every child of X, say Y, is able to connect to a vertex "higher" in the tree (above X, with smaller DFS # than X) without going through X

- Connection does not have to be a direct edge
  - We could go down through Y's descendents, then eventually back up to a vertex above X
- If any child is not able to do this, X is an articulation point
- How to determine this on the computer?
  - For each vertex, when we visit it, we need to keep track of the minimum DFS # that we can reach from this vertex and any of its descendents
  - We then use this min reachable DFS # to determine if a vertex is an articulation point or not:
    - For any vertex X, and its child in the tree Y, if min reachable DFS # for Y is >= DFS #(X), then X is an articulation point

```
int visit(int k) // DFS to find articulation points
  {
    struct node *t;
    int m, min;
    val[k] = ++id; // assign DFS number to current vertex
    min = id;      // initialize min reachable vertex to DFS
                   // number of vertex itself
    for (t = adj[k]; t != z; t = t->next)  // go through adj list
      if (val[t->v] == unseen)  // if neighbor is unseen, will
        {                       // be a child in DFS tree
          m = visit(t->v);   // get min reachable DFS # of child
                             // via recursive call
          if (m < min) min = m;  // if this is a smaller DFS #
                                 // than previous, update min
          if (m >= val[k]) cout << name(k);  // if it is >= to
                             // current DFS #, node is an
        }                    // articulation point
      else if (val[t->v] < min) min = val[t->v];  // seen neighbor
                             // this is a back edge.  If DFS # of
    return min;              // this neighbor less than previous,
  }                          // update min
```

- Algorithm shown works for all vertices except the root of the DFS tree
  - Child of root cannot possibly reach a smaller DFS # than that of the root, since the root has DFS # 1
  - Root of DFS tree is an articulation point if it has two or more children in the DFS tree
    - Idea is that since we backtrack only when necessary in the DFS algorithm, having two children in the tree means that we couldn't reach the second child except by going back through the root
    - This implies that the first child and second child have no way of connecting except through the root
      - > Thus in this case the root is an articulation point

- In unweighted graphs, all edges are equivalent
- Often in modeling we need edges to represent distance, flow, capacity, bandwidth, etc
  - Thus we must put WEIGHTS on the edges to represent these values
- Representations:
  - In an adjacency matrix, we can substitute the weight for one (zero still means no edge)
  - In an adjacency list, we can add a field to our linked list nodes for weight of the edge

- In an unweighted graph, we have seen algorithms to determine
  - Spanning Tree of the graph
    - Both DFS and BFS generate this during the traversal process
  - Shortest Path between two vertices
    - BFS does this by determining spanning tree based on number of edges vertex is away from starting point
    - The spanning tree generated by BFS shows the shortest path (in terms of number of edges) from the starting vertex to all other vertices in the graph
- However, in a weighted graph, these ideas can be more complex

- Minimum Spanning Tree
  - The spanning tree of a graph whose edge weights sum to the minimum amount
  - Clearly, a graph has many spanning trees, not all of which are minimum
- Weighted Shortest Path
  - The path between two vertices whose edge weights sum to the minimum value
  - Note that now the fewest edges does not necessarily imply the shortest path

- Prim's Algorithm
  - Idea of Prim's is very simple:
    - Let T be the current tree, and T' be all vertices and edges not in the current tree
    - Initialize T to the starting vertex (T' is everything else)
    - while (#vertices in T < v)
      - Find the smallest edge connecting a vertex in T to a vertex in T'
      - Add the edge and vertex to T (remove them from T')
  - As is often the case, implementation is not as simple as the idea

- Naïve implementation:
  - At each step look at all possible edges that could be added at that step
  - Let's look at the worst case for this impl:
    - Complete graph (all edges are present)
      - Step 1: (v-1) possible edges (from start vertex)
      - Step 2: 2(v-2) possible edges (first two vertices each have (v-1) edges, but shared edge between them is not considered)
      - Step 3: 3(v-3) possible edges (same idea as above)
      - …
    - Total: Sum (i = 1 to v-1) of i(v-i)
      - This evaluates to Theta(v^3)

- Better implementation:
  - Do we have to consider so many edges at each step?
    - No. Instead, let's just keep track of the current "best" edge for each vertex in T'
    - Then at each step we do the following:
      - Look at all of the "best" edges for the vertices in T'
      - Add the overall best edge and vertex to T
      - Update the "best" edges for the vertices remaining in T', considering now edges from latest added vertex
    - This is the idea of Priority First Search (PFS)

- In the PFS implementation (adj. list)
  - "Best" edges for each vertex are kept in the val[] array and are also maintained in a priority queue
  - At each step the overall best vertex (i.e. the one with the smallest edge) is removed from the PQ
    - Then its adjacency list is traversed, and the remaining vertices in the PQ are updated if necessary
  - Algorithm continues until the PQ is empty
- Adj. matrix implementation has some important differences – see later notes
- See graphs2.txt and handout for trace
- Also **see PFS.cpp** for commented code

- Run-time (adjacency list)?
  - Algo. looks at each edge in the graph, just as BFS does
    - However, now for each edge, a PQ operation will be done
      - > Insert a vertex, edge pair
      - > Delete the vertex with the smallest edge
      - > Update a vertex to give it a new (better) edge
    - If we implement the PQ in a smart way, we can do these operations very efficiently
      - > Use a HEAP for the PQ itself, allowing Insert and Delete to be time Theta(lg v)
      - > Have an additional indexing array to locate the vertices in the heap, enabling Update to also be time Theta(lg v)
    - Thus, assuming the graph is connected, the overall run-time is Theta(e(lg v)) for an adjacency list

- What about an adjacency matrix?
  - We don't need a heap for the PQ in this case
  - Instead traverse the val[] array at each step to find the best edge
    - Since it takes Theta(v) to traverse the row of the matrix to find the neighbors of the current vertex, an extra V to find the best vertex to pick next does not add any extra asymptotic time
  - Thus, the overall run-time is the same as for BFS – Theta(v^2)
- How do they compare?
  - elgv vs v^2
  - Depending upon how e relates to v, one may be preferable -- see notes from board

- Djikstra's Shortest Path algorithm
  - Can be implemented using the SAME PFS that we just discussed for MST
  - The only difference is the measure used for the priority
    - With MST, we wanted the smallest overall weight
      - For each vertex we wanted the smallest edge that could connect it to the tree
    - With SP, we want the smallest weight PATH from the starting vertex to each other vertex
      - For each vertex we want the smallest PATH that could connect it to the starting vertex
  - See trace in handout and on board

- We saw need for a PQ in adjacency list PFS
- Let's look a bit closer at PQs
  - We need 3 primary operations
    - Insert an object into the PQ
    - Find the object with best priority
      - Often called FindMin or FindMax
    - Remove the object with best priority
      - Often called DeleteMin or DeleteMax
  - How to implement?
    - 2 obvious implementations:
      - Unsorted Array
      - Sorted Array

- Unsorted Array PQ:
  - Insert: ***Add new item at end of array: Theta(1) run-time***
  - FindMin: ***Search array for smallest: Theta(N) run-time***
  - DeleteMin: ***Search array for smallest and delete it: Theta(N) run-time***

For a N inserts and deletes, total time is Theta(N^2)

- Sorted Array PQ:
  - Insert: ***Add new item in reverse sorted order: Theta(N) run-time***
  - FindMin: ***Smallest item at end of array: Theta(1) run-time***
  - DeleteMin: ***Delete item from end of array: Theta(1) run-time***

For a N inserts and deletes, total time is Theta(N^2)

- How can we improve our overall run-time?
  - We could use a BST
    - This would give Theta(lgN) for each operation
      - Discuss
    - However, it is MORE than we need
      - It maintains a complete ordering of the data, but we only need a PARTIAL ORDERING of the data
  - Instead we will use a HEAP
    - Complete binary tree such that for each node T in the tree

      T.val < T.lchild.val

      T.val < T.rchild.val
    - See example on the board

- Note that we don't care how T.lchild.val relates to T.rchild.val
  - But BST does care, which is why it gives a complete ordering
- Ok, how do we do our operations:
  - FindMin is easy – ROOT of tree
  - Insert and DeleteMin are not as trivial
  - For both we are altering the tree, so we must ensure that the HEAP PROPERTY is reestablished

- Idea of Insert:
  - Add new node at next available leaf
  - Push the node "up" the tree until it reaches its appropriate spot
    - We'll call this upHeap
  - See example on board
- Idea of DeleteMin:
  - We must be careful since root may have two children
    - Similar problem exists when deleting from BST
  - Instead of deleting root node, we overwrite its value with that of the last leaf

- Then we delete the last leaf -- easy to delete a leaf
- But now root value may not be the min
- Push the node "down" the tree until it reaches its appropriate spot
  - We'll call this downHeap
- See example on board

- Run-time?
  - Complete Binary Tree has height ≈ lgN
  - upheap or downheap at most traverse height of the tree
  - Thus Insert and DeleteMin are always Theta(lgN) worst case
  - For N Inserts + DeleteMins total = Theta(NlgN)

- How to Implement a Heap?
  - We could use a linked binary tree, similar to that used for BST
    - Will work, but we have overhead associated with dynamic memory allocation and access
  - But note that we are maintaining a complete binary tree for our heap
  - It turns out that we can easily represent a complete binary tree using an array

- Idea:
  - Number nodes row-wise starting at 1
  - Use these numbers as index values in the array
  - Now, for node at index i

    Parent(i) = i/2
    LChild(i) = 2i
    RChild(i) = 2i+1

  - See example on board
- Now we have the benefit of a tree structure with the speed of an array implementation

- Recall the PQ operations:
  - Insert
  - FindMin (or FindMax)
  - DeleteMin (or DeleteMax)
- Recall operations needed for PFS:
  - Insert
  - DeleteMax
    - Since items are stored using negative priorities, the max value, when negated, will be the min positive value
  - Update – assign a new priority to a vertex
    - How to do this one?
    - Do you see why it is a problem?

- In order to allow Update in our PQ, we must be able to do a general Find() of the data
  - PQ is only partially ordered, and further it is ordered on VALUES, not vertex ids
  - Finding an arbitrary id will take Theta(N)
- Luckily, we can do it better with a little thought
  - We can think of each entry in 2 parts:
    - A vertex id (int)
    - A priority value
  - To update, we need to locate the id, update the priority, then reestablish the Heap property

- We can do this using indirection
  - Keep an array indexed on the vertex ids which, for vertex i gives the location of i in the PQ
  - When we move i in the PQ we change the value stored in our array
  - This way we can always "find" a vertex in Theta(1) time
  - Now, total time for update is simply time to reestablish heap priority using upheap or downheap
- See pqtest.cpp, pq.h and pq.out

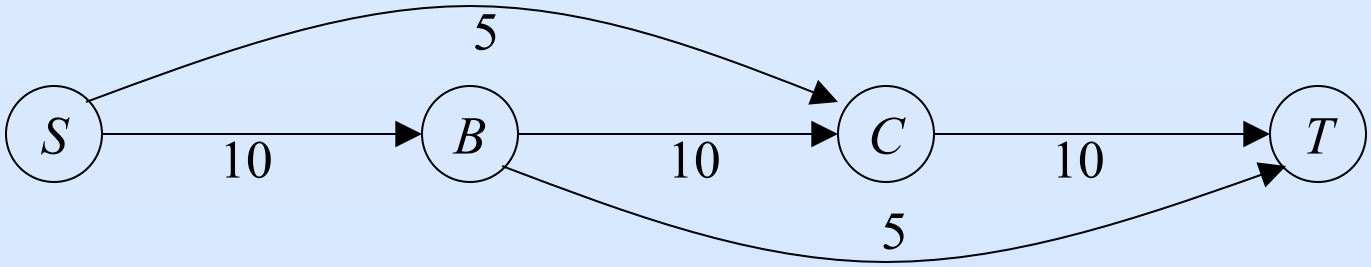
- Definitions:
  - Consider a directed, weighted graph with set V of vertices and set E of edges, with each edge weight indicating a capacity, c(u,v) for that edge
    - c(u,v) = 0 means no edge between u and v
  - Consider a source vertex, s with no in-edges and a sink vertex, t with no out-edges
  - A FLOW on the graph is another assignment of weights to the edges, f(u,v) such that the following rules are satisfied:

1) For All (u,v) in V, f(u,v) <= c(u,v)
   - No flow can exceed the capacity of the edge
2) For All (u,v) in V, f(u,v) = – f(v,u)
   - If a positive flow is going from u to v, than an equal weight negative flow is going from v to u
3) For All u in V – {s, t}, sum of (flow on edges from u) = 0
   - All vertices except the source and sink have an overall "net" flow of 0 – the amount coming in is the same as the amount going out

- Problem: What is the maximum flow for a given network and how can we determine it?

- Ford-Fulkerson approach:
  - For a given network with a given flow, we look for an augmenting path from the source to the sink
    - An augmenting path is a path from the source to the sink that provides additional flow along it
  - After finding an augmenting path, we update the residual network to reflect the additional flow
  - We repeat this process on the residual network until no augmenting path can be found
  - At this point, the maximum flow has been determined
- See examples on board

- Note the following example:



  - Let's look at 2 possible sequences of augmenting paths:
    1) SBT (5), SCT (5), SBCT (5)
       - Total Flow is 15
    2) SBCT (10), ????
       - We seem to be stuck, since there is no forward path from S to T
       - Yet the choice of paths should not affect our answer

- How to resolve this problem?
  - The augmenting paths do not represent final flow values for given edges
    - > They are simply a tool for determining the overall maximum flow
  - A given assignment for an edge may be MORE than what the final network flow would indicate
    - > Idea is that we overassigned it during an augmenting path, and the actual flow on that edge will in fact be less than the assigned value
  - Backward flow is a way to lessen the flow on a given edge while increasing the overall flow in the graph
  - In the example shown we can thus have the following augmenting paths:
    - > SBCT (10), SCBT (5)

- Augmenting path SCBT has an overall positive value of 5
- However, the link CB actually reduces the flow along that edge by 5, since it is in the backward direction on that edge

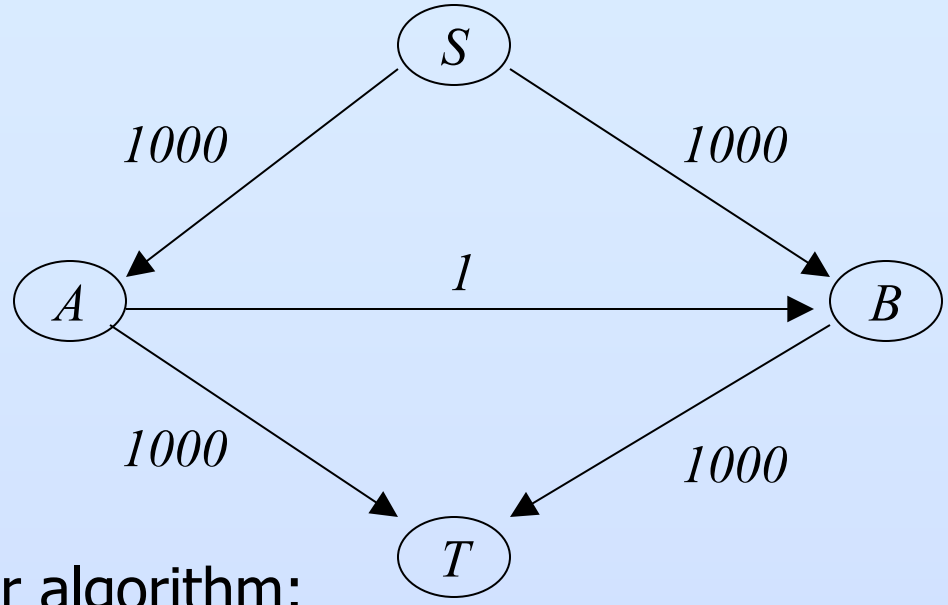
- Note that the final assignment on the edges for both sequences of augmenting paths is the same
  - This is not always the case
  - If the assignment of maximum flow is unique, they will be the same
  - If more than one assignment gives the same value for maximum flow, difference choices for augmenting paths can result in different assignments
    - But the overall flow will always be the same

- To implement FF, we need
  - A way to represent the graph, capacity, flow, etc.
    - What data structures will we need?
    - See notes on board and in flownew.cpp
    - Adjacency matrix (or list) in two parts:
      - size[][] gives capacity of edges
        - 0 if no edge
        - Directed, and for edge [i][j], size[j][i] = – size[i][j] (see rule 2)
      - flow[][] gives current flow on edges
        - Initially 0
        - flow[i][j] always less than size[i][j] (see rule 1)
        - Rule 2 applies here as well

- dad[] array to store augmenting path

- A way to determine an augmenting path for the graph
  - How that this be done in a regular, efficient way?
  - We need to be careful so that if an augmenting path exists, we will find it, and also so that "quality" of the paths is fairly good

- Ex: Consider the following graph



- Poor algorithm:
  - Aug. Paths SABT (1), SBAT (1), SABT (1), SBAT (1) …
  - Every other path goes along edge AB in the opposite direction, adding only 1 to the overall flow
  - 2000 Aug. Paths would be needed before completion

- Good algorithm:
  - Aug. Paths SAT (1000), SBT (1000) and we are done

- In general, if we can find aug. paths using some optimizing criteria we can probably get good results
- Edmonds and Karp suggested two techniques:
  - Use BFS to find the aug. path with fewest edges
  - Use PFS to find the aug. path with largest augment
    - In effect we are trying to find the path whose segments are largest
    - Since amount of augment is limited by the smallest edge, this is giving us a "greatest" path
- See board and flownew.cpp

- Let's look at flownew.cpp
  - For now we will concentrate on the "main" program
    - Later we will the PFS augmenting path algo
  - Main will work with BFS or PFS – it uses algo to build spanning tree starting at source, and continuing until sink is reached
    - Total flow is updated by the value for the current path
    - Each path is then used to update residual graph
      - > Path is traced from sink back to source, updating each edge along the way
    - If sink is not reached, no augmenting path exists and algorithm is finished

- Some computational problems are unsolvable
  - No algorithm can be written that will always produce the "right" answer
  - Most famous of these is the "Halting Problem"
    - Given a program P with data D, will P halt at some point?
      - It can be shown (through a clever example) that this cannot be determined for an arbitrary program
  - Other problems can be "reduced" to the halting problem
    - Indicates that they too are unsolvable

- Some problems are solvable, but require an exponential amount of time
  - We call these problems intractable
    - For even modest problem sizes, they take too long to solve to be useful
  - Ex: List all subsets of a set
    - We know this is Theta(2^N)
  - Ex: List all permutations of a sequence of characters
    - We know this is Theta(N!)

- Most useful algorithms run in polynomial time
  - Largest term in the Theta run-time is a simple power with a constant exponent
  - Most of the algorithms we have seen so far this term fall into this category

- Background
  - Some problems don't (yet) fall into any of the previous 3 categories:
    - We have not proven that any solution requires exponential execution time
    - No one has been able to produce a valid solution that runs in polynomial time
  - It is from within this set of problems that we produce NP-complete problems

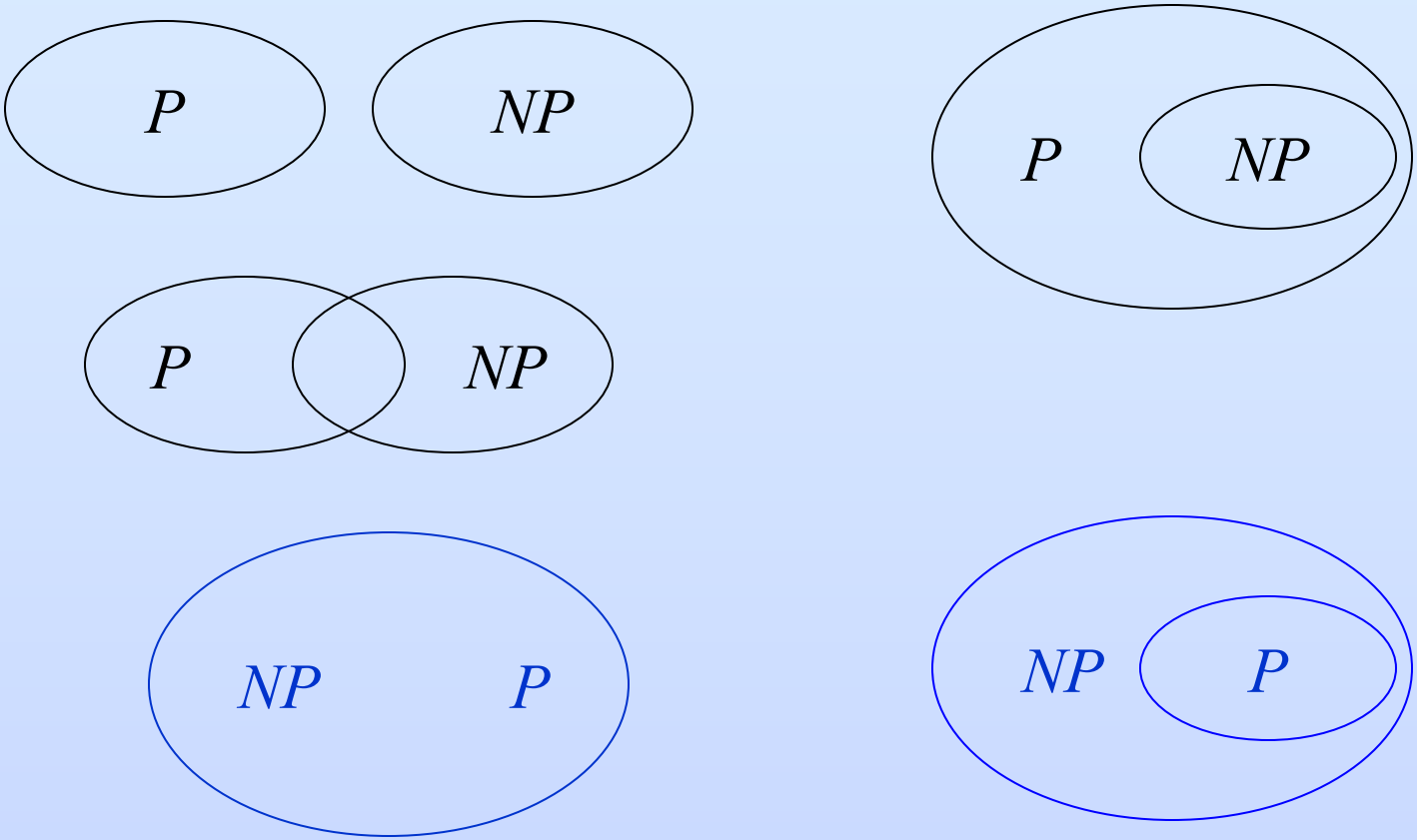
- More background:
  - Define P = set of problems that can be solved by deterministic algorithms in polynomial time
    - What is deterministic?
      - At any point in the execution, given the current instruction and the current input value, we can predict (or determine) what the next instruction will be
      - Most algorithms that we have discussed this term fall into this category

- Define NP = set of problems that can be solved by non-deterministic algorithms in polynomial time
  - What is non-deterministic?
    - Formally this concept is tricky to explain
      - > Involves a Turing machine
    - Informally, we allow the algorithm to "cheat"
      - > We can "magically" guess the solution to the problem, but we must verify that it is correct in polynomial time
    - Naturally, our programs cannot actually execute in this way
      - > We simply define this set to categorize these problems
  - http://www.nist.gov/dads/HTML/nondetermAlgo.html
  - http://en.wikipedia.org/wiki/Nondeterministic_algorithm

- Ex: TSP (Traveling Salesman Problem)
  - Instance: Given a finite set $C = \{c_1, c_2, \ldots c_m\}$ of cities, a distance $d(c_I, c_J)$ for each pair of cites, and in integer bound, B (positive)
  - Question: Is there a "tour" of all of the cities in C (i.e. a simple cycle containing all vertices) having length no more than B?
- In non-deterministic solution we "guess" a tour (ex: minimum length) and then verify that it is valid and has length <= B or not within polynomial time
- In deterministic solution, we need to actually find this tour, requiring quite a lot of computation
  - No known algo in less than exponential time

- So what are NP-Complete Problems?
  - Naturally they are problems in set NP
  - They are the "hardest" problems in NP to solve
    - All other problems in NP can be transformed into these problems in polynomial time
  - If any NP-complete problem can be solved in deterministic polynomial time, then all problems in NP can be
    - Since the transformation takes polynomial time, we would have
      - A polynomial solution of the NP-Complete problem
      - A polynomial transformation of any other NP problem into this one
      - Total time is still polynomial

- Consider sets P and NP:
  - We have 5 possibilities for these sets:



- 3 of these can be easily dismissed
  - We know that any problem that can be solved deterministically in polynomial time can certainly be solved non-deterministically in polynomial time
- Thus the only real possibilities are the two in blue:
  - P $\subset$ NP
    - > P is a subset of NP, as there are some problems solvable in non-deterministic polynomial time that are NOT solvable in deterministic polynomial time
  - P = NP
    - > The two sets are equal – all problems solvable in non-deterministic polynomial time are solvable in deterministic polynomial time

- Right now, we don't know which of these is the correct answer
  - We can show $P \subset NP$ if we can prove an NP-Complete problem to be intractable
  - We can show $P = NP$ if we can find a deterministic polynomial solution for an NP-Complete problem
- Most CS theorists believe the $P \subset NP$
  - If not, it would invalidate a lot of what is currently used in practice
    - Ex: Some security tools that are secure due to computational infeasibility of breaking them may not actually be secure
- But prove it either way and you will be famous!

- Situation:
  - You have discovered a new computational problem during your CS research
  - What should you do?
    - Try to find an efficient solution (polynomial) for the problem
    - If unsuccessful (or if you think it likely that it is not possible), try to prove the problem is NP-complete
      - This way, you can at least show that it is likely that no polynomial solution to the problem exists
      - You may also try to develop some heuristics to give an approximate solution to the problem

- How to prove NP-completeness?
  1) From scratch
     - Show the problem is in NP
     - Show that all problems in NP can be transformed to this problem in polynomial time
     - Very complicated – takes a lot of work to do
     - Luckily, we only need to do this once, thanks to method 2) below
  2) Through reduction
     - Show that some problem already known to be NP-complete is reducible (transformable) to the new problem in polynomial time
     - Idea is that, since one prob. can be transformed to the other in poly time, their solution times differ by at most a polynomial amount

- Ok, so a problem is NP-complete
  - But we may still want to solve it
  - If solving it exactly takes too much time using a deterministic algorithm, perhaps we can come up with an approximate solution
    - Ex: Optimizing version of TSP wants the minimum tour of the graph
      - Would we be satisfied with a tour that is pretty short but not necessarily minimum?
    - Ex: Course scheduling
    - Ex: Graph coloring

- We can use heuristics for this purpose
  - Goal: Program runs in a reasonable amount of time, yet gives an answer that is close to the optimal answer
- How to measure quality?
  - Let's look at TSP as an example
    - Let H(C) be the total length of the minimal tour of C using heuristic H
    - Let OPT(C) be the total length of the optimal tour
    - Ratio bound:
      - H(C)/OPT(C) gives us the effectiveness of H
      - How much worse is H than the optimal solution?

- For original TSP optimization problem, it can be proven that no constant ratio bound exists for any polynomial time heuristic
  - Discuss
- But, for many practical applications, we can restrict TSP as follows:
  - For each distance in the graph:

$$d(c_I,c_J) <= d(c_I,c_K) + d(c_K,c_J)$$

    - TRIANGLE INEQUALITY
      - A direct path between two vertices is always the shortest
  - Given this restriction, heuristics have been found that give ratio bounds of 1.5 in the worst case

- How do heuristics approach a problem?
  - Many different approaches, but we will look only at one: Local Search
    - Idea: Instead of optimizing the entire problem, optimize locally using a neighborhood of a previous sub-optimal solution
    - We are getting a local optimum rather than a global optimum
  - Let's look at an example local search heuristic for TSP (assuming triangle inequality)
    - We call this heuristic 2-OPT

- 2-OPT Idea:
  - Assume we already have a valid TSP tour
  - We define a neighborhood of the current tour to be all possible tours derived from the current tour by the following change:
    - Given that (non-adjacent) edges (a,b) and (c,d) are in the current tour, replace them with edges (a,c) and (b,d)
    - See picture on board
    - Note that each transformation guarantees a valid tour
    - For each iteration we choose the BEST such tour and repeat until no improvement can be made
    - See trace on board

- Implementation: can be done in a fairly straightforward way using an adjacency matrix
  - Pseudocode:

```
bestlen = length of initial tour
while not done do
   improve = 0;
   for each two-opt neighbor of current tour
       diff = (C(A,B) + C(C,D)) - (C(A,C) + C(B,D))
       if (diff > improve)
              improve = diff
              store current considered neighbor
   if (improve > 0)
       update new tour
       bestlen -= improve
   else
       done = true
```

  - Let's look at the code – twoopt.c

- Performance issues:
  - How big is a neighborhood (i.e. how many iterations will the foreach loop have)?
    - Consider "first" edge in the original tour
      Cannot consider with it any adjacent edges or itself
      Thus, if the tour has n total edges, we have n-3 neighbor tours with the first edge in them
    - Now consider the other edges in the original tour
      Same number of neighbor tours as the first
      However, each neighbor is considered twice (once from each original edge's point of view)
    - Thus we have (N-3)(N)/2 total neighbors of the original tour
    - Asymptotically Theta(N^2) neighbors

- How many iterations are necessary (i.e. how many iterations of outer while loop are required?)
  - In the worst case this can be extremely high (exponential)
  - But this is a rare (and contrived) case
  - In practice, few iterations of the outer loop are generally required
- What is the quality of the final tour?
  - Again, in the worst case it is not good
  - In normal usage, however, it does very well (a ratio bound of 1.05 on average, given a good initial tour)
- Variations on local search can improve the situation so that the worst case problems of 2-OPT go away
  - More sophisticated algorithms such as simulated annealing and genetic algorithms

- In Divide and Conquer problems
  - We break a large problem into subproblems and use the subproblem solutions to solve the original problem
  - However, in some situations this approach is not an efficient one
    - Inefficiency occurs when subproblems must be recalculated many times over and over
  - Famous example is the Fibonacci Sequence:
  - Recursively we see:
    - FIB(N) = FIB(N-1) + FIB(N-2) for N > 2
    - FIB(2) = FIB(1) = 1

- However, if we trace the execution of this problem, we see that the execution tree is very large – exponential in N
  - See simple trace on board
- If we approach this problem from the "bottom up", we can improve the solution efficiency
  - Start by solving FIB(1), then FIB(2), and build the solution until we get to the desired value of N
  - Now we are calculating each smaller solution only one time
- See fibo.java
  - Discuss

- Some problems that have exponential run-times can be solved in pseudo-polynomial time using dynamic programming
  - Idea: Given some instances of the problem, by starting at a solution to a small size and building up to a larger size, we end up with a polynomial run-time
  - Example: Subset Sum
    - Given a set of N items, each with an integer weight and another integer M
    - Is there a subset of the set that sums to M?

- Exhaustive Search
  - Try (potentially) every subset until one is found that sums to M or none are left
  - Theta(2^N) since there are that many subsets
- Branch and Bound
  - If we do this recursively, we can stop the recursion and backtrack whenever the current sum exceeds M
    - Greatly reduces the number of recursive calls required
    - Still exponential in the worst case
  - See subset.java

- Dynamic Programming
  - Solve problem for M = 1, M = 2, … up to the input value of M
  - Use the previous solutions in a clever way to help solve the next size
    - We are using an array of size M to store information from previous answers
  - Look at the code
    - See structure
    - Note that we are using a lot of space here
  - However, note that we can come up with situations in which this solution is very good (pseudo polynomial) but also in which it is very poor

N = 20, M = 1000

10 20 30 5 15 25 8 10 16 22 24 26 2 4 6 1 3 7 9 1000

- This instance shows the worst case behavior of the branch and bound solution
  - All combinations of the first 19 elements (2^19) must be tried but to no avail until the last element is tried
  - It is poor because we don't ever exceed the "bound" so the technique doesn't help us
  - The dynamic programming solution in this case is much better

N = 4, M = 1111111111

1234567890 1357924680 1470369258 1111111111

- This instance shows the worst case behavior of the dynamic programming solution
  - Recall that our array must be size M, or 1111111111
    - Already, we are using an incredible amount of memory
    - We must also process all answers from M = 1 up to M = 1111111111, so this is a LOT of work
  - Since N = 4, the branch and bound version will actually work in very few steps and will be much faster
  - This is why we say dynamic programming solutions are "pseudo-polynomial" and not actually polynomial