Algorithms and data structures

This course will examine various data structures for storing and accessing information together with relationships between the items being stored, and algorithms for efficiently finding solutions to various problems, both relative to the data structures and queries and operations based on the relationships between the items stored. We will conclude by looking at some theoretical limitations of algorithms and what we can compute.

1. Introduction and review

We will start with a course introduction, a review of the relevant mathematical material students should have already seen in previous courses, an overview of C++ and a focus on mathematical induction.

1.1 Introduction

Often, we are faced with an issue of dealing with items in order of their priority relative to each other. Items waiting for a service will arrive at different times, but they must be serviced in order of the priority.

1.2 Mathematical background

Students must understand: the floor and ceiling functions, l'Hôpital's rule, logarithms (both that all logarithms are scalar multiples of each other and that logarithms grow slower than any polynomial n^d where $d > 0$, the sums of arithmetic and geometric series, approximations to the sums of polynomial series, a general understanding of recurrence relations, the concept of weighted averages and the concept of a combination.

1.3 C++

The C++ programming language is similar to C, Java and C#. Where it differs from C# and Java are in its memory allocation model (explicitly having to deallocate memory as opposed to relying on a garbage collector), pointers explicitly recording the location in address in memory where an object is stored, and the concept of a pre-processor. Where it differs from all three languages is the concept of templates: allowing the user of the class to specify types. C++ also uses namespaces to prevent collisions on large projects. We will use the `std` namespace of the standard template library (STL).

1.4 Mathematical induction

A proof by induction attempts to show that a statement $f(n)$ is true for all values $n \geq n_0$ by first showing that $f(n_0)$ is true, and then then showing that $f(n) \rightarrow f(n + 1)$; that is, if we assume that $f(n)$ is true, it follows that $f(n + 1)$ is also true.

2. Algorithm analysis

We describe containers to store items, relationships between items we may also want to record, the concept of abstract data types, data structures and algorithms that will implement these structures and solve problems, and the asymptotic analysis we will use to analyze our algorithms.

2.1 Containers, relations and abstract data types (ADTs)

All problem solving on a computer involves storing, accessing, manipulating and erasing data on a computer. Now, there are operations that we may want to perform on containers (taking the union of two containers, finding the intersection, emptying a container, determining the number of objects in the container, etc.) Usually, however, we want to store more than just data: we also want to store relationships between the items. In this case, we may also want to either make queries or perform operations based on those relationships. In general, we do not need to perform all possible operations in all situations. When we come across a description of a container and relevant set of instructions that is used very often, we can describe this as an abstract data type. The relationships we are interested in are linear orderings, hierarchical orderings, partial orderings, equivalence relations, weak orderings (a linear ordering of equivalence relations), and adjacency relations. We look at examples of all of these. We consider how relationships can be defined. In some cases, there is a global mechanism for comparing any two objects (3.5412 < 5.2793); in others, the relationship is locally defined (Ali's manager is Bailey). We quickly described two abstract data types: the List ADT and the Sorted List ADT.

2.2 Data structures and algorithms

The allocation of space for storing information in computer memory may be described as either contiguous (as in arrays) or node based (as in linked lists). A third form is index based, where an array points to a sequence at different locations in memory. We look at how a tree could be defined in a manner similar to that of a linked list, only with multiple possible successors. We consider graphs and the Unix inode as examples of hybrid structures. We consider how some operations may be slow or fast given the underlying data structure, and we ask whether or not such descriptions of the run time can be described quantitatively as opposed to qualitatively.

2.3 Asymptotic analysis

We will look consider the growth of functions without regard to the coefficients of the leading terms. For the functions we will be interested in, we will consider the limit of the ratio $\frac{f(n)}{g(n)}$, and if this limit is 0, finite, or infinite, we will say $f(n) = o(g(n))$, $f(n) = \Theta(g(n))$ or $f(n) = \omega(g(n))$, respectively; that is, either $f(n)$ grows significantly slower, at the same rate, or significantly faster, respectively. We observe that this defines a weak ordering on the functions we have an interest in where $1 = o(\ln(n))$, $\ln(n) = o(n^k)$ for any $k > 0$, $n^p = o(n^q)$ if $0 \le p < q$, $n = o(n \ln(n))$, $p^n = o(q^n)$ for $1 < p < q$, and $p^n = o(n!)$. In some cases, we may have a collection of functions, some of which are $o(g(n))$ while others are $\Theta(g(n))$. In this case, we could say that the collection of functions is $O(g(n))$. We have a similar definition for $\Omega(g(n))$.

2.4 Algorithm analysis

Determining the run time of code requires us to consider the various components. All operators in C++ run in $\Theta(1)$ time. If two blocks of code run in $O(f(n))$ and $O(g(n))$ time, respectively, if they are run in series, the run time is $O(f(n) + g(n))$. Therefore, any finite and fixed set of operators run serially may also be said to run in $\Theta(1)$ time. A loop that cycles n times will run in $\Theta(n)$ time if the body is $\Theta(1)$, but if there is the possibility of finishing early, we would say it runs in $O(n)$ time. If the body of the loop also has a run time depending on n, the run time is $O(n\,f(n))$. If, however, the body of the loop iterates over the sequence $a \le k \le b$, and the run time of the body depends on k, say $O(f(k))$, the run time may be calculated as $O\left(\sum_{k=a}^{b} f(k)\right)$. If a function is called and we are not aware of its run time, we may represent it by a placeholder T. If we are aware that the run time depends on a parameter, we may write $T(n)$. In some cases, we may simply determine that the run time of a function $S(n) = O(n) + T(n)$. For example, one function may iterate through an array of size n and then call another function. When a function calls itself, however, we call that a recursive function. For example, $T(n) = T(n-1) + \Theta(1)$ or $S(n) = S(n/2) + \Theta(1)$ when $n > 1$. In general, we assume that $T(1) = \Theta(1)$; that is, the time it takes to solve a trivial sized problem is $\Theta(1)$. In the case of these two examples, we can solve the recurrence relations to determine that $T(n) = \Theta(n^2)$ while $S(n) = \Theta(\ln(n))$.

3. Lists, stacks and queues

We will now look at data structures for storing items linearly in an order specified by the programmer (an explicitly defined linear order).

3.1 Lists

There are numerous occasions where the programmer may want to specify the linear order. Operations we may want to perform on a list are insert an object at particular location, move to the previous or next object, or remove the object at that location. Both arrays and singly linked lists are reasonable for some but not all of these operations. We introduce doubly linked lists and two-ended arrays to reduce some of the run times but at a cost of more memory. We observe that in general, it is often possible to speed up If the objects being linearly ordered are selected from a finite and well defined alphabet, the list is referred to as a *string*. This includes text but also DNA where the alphabet is comprised of four amino acids adenine, thymine, guanine and cytosine (A, T, G and C).

3.2 Stacks

One type of container we see often is a last-in—first-out container: items may be inserted into the container (*pushed* onto the stack) in any order, but the item removed (popped) is always the one that has most recently been pushed onto the stack. The last item pushed onto the stack is at the *top* of the stack. This defines an *abstract stack* or *Stack ADT*. This is simple to implement efficiently (all relevant operations are $\Theta(1)$) with a singly linked list and with a one-ended (standard) array. Stacks, despite being trivial to implement, are used in parsing code (matching parentheses and XML tabs), tracking function calls, allowing undo and redo operations in applications, in reverse-Polish operations, and is the format for assembly language instructions. With respect to the array-based implementation, we focus on the amortized effect on the run time if the capacity is doubled when the array is full, and when we increase the capacity by a constant amount. In the first case, operations have an amortized run time of $\Theta(1)$ but there is $O(n)$ unused memory, while in the second the amortized run-time is $O(n)$ while the unused memory is $\Theta(1)$.

3.3 Queues

Another type of container we see often is a first-in—first-out container, a behavior desirable in many client-server models where clients waiting for service enter into a queue (pushed onto the back of the queue) and when a server becomes reading, it begins servicing the client that has been waiting the longest in the queue (the client is popped off the front of the queue). This defines an *abstract queue* or *Queue ADT*. This can be implemented efficiently (all relevant operations are $\Theta(1)$) with either a singly linked list or a two-ended cyclic array. With respect to the array-based implementation, we focus on the characteristics of a cyclic array, including the requirement for doubling the capacity of the array when full.

3.4 Deques

A less common container stores items as a contiguous list but only allows insertions and erases at either end (pushes and pops at the front and back). This defines an abstract doubly ended queue or *abstract deque* or *Deque ADT*. This can be implemented efficiently using a two ended array but requires a doubly linked list for an efficient implementation using a linked list. For this data structure, we look at the concept of an iterator: an object that allows the user to step through the items in a container without gaining access to the underlying data structure.

4. Trees and hierarchical orders

We will now look at data structures for storing items linearly in an order specified by the programmer (an explicitly defined linear order). However, to start, we will first look at trees and their obvious purpose: to store hierarchical orders.

4.1 The tree data structure

A tree is a node-based data structure where there is a single root node, and each node can have an arbitrary number of *children* (the degree of a node being the number of children it has). Nodes with no children are *leaf nodes* while others are *internal nodes*. The concepts of ancestors and descendants is defined and a node and all its descendants is considered to be a sub-tree within a tree. We define paths within a tree and the lengths of paths, the depth of a node, and the height of a tree. We look at how the tree structure can be used to define markups in HTML and how XML, in general, defines a tree structure.

4.2 Abstract trees

An abstract tree stores nodes within a hierarchical ordering. Questions we may ask include determining children and parents, and getting references to either the parent or iterating through the children. Operations include adding and removing sub-trees. To implement this, we consider a class which stores a value and a reference to the parent. In addition, children are stored in a linked list of references to children. If the linked list is empty, the node is a leaf node. We observe how we can implement the various queries and operations defined above on such a data structure. We also look at how hierarchies are almost always locally defined: at the very least, either every node must specify its parent, or each node must store its children.

4.3 Tree traversals

In stepping through all the entries in an array or linked list, one need only walk through the *n* entries. In a tree, this is more difficult. We have already seen how we can perform a breadth-first traversal of a tree using a queue. Another approach is a depth-first traversal where a node is visited, and then the children are visited in order using the same depth-first traversal order. Operations at any node may be performed before the children are visited or after, depending on whether calculations are required for the sub-trees, or whether the children are returning information that is to be used by the node. Operations that must be performed prior to visiting the children are referred to as *pre-order* and those that require information from the children are *post-order*. We look at how this can be used to print a directory hierarchy in a standard format, and how to calculate the total memory used by a directory and all its sub-directories including the files in those directories.

5. Ordered trees

We will look at trees where there are a fixed number of children, and the order of the children is relevant. We will start with binary trees and then look at N-ary trees.

5.1 Binary trees

A binary tree is where each node has two named children: left and right children forming left and right sub-trees. We define an *empty node* or *null sub-tree* as any child which does not exist. A full node is a node with two children. A full binary tree is a binary tree where every internal node is full. The implementation of such a structure is quite straight-forward, and recursive algorithms can be used to make various calculations such as determining the size and height of trees. As one application, we consider ropes: full binary trees where each child is either a string or is itself another rope. The rope defines a string formed by the concatenation of the children.

5.2 Perfect binary trees

A perfect binary tree of height $h = 0$ is a single leaf node. A perfect binary tree of height $h > 0$ is a tree which has two sub-trees, both of which are perfect binary trees of height $h - 1$. Similarly, you can define a perfect binary tree as a tree where all internal nodes are full and all leaf nodes are at the same depth. The number of nodes is $n = 2^{h+1} - 1$ and the height is $\lg(n + 1) - 1 = \Theta(\ln(n))$. There are 2^h leaf nodes, so over half the entries are leaf nodes and the average depth of a node is approximately $h - 1$. This will be the ideal case for all other binary trees.

5.3 Complete binary trees

A complete binary tree is one that is filled in breadth-first traversal order. The height of such a tree is still $h = \lfloor \lg(n) \rfloor = \Theta(n)$. The benefit of such an ordering is that the tree can be represented not using nodes, but as an array filled through a breadth-first traversal order. In this case, if the root is at index $k = 1$, then the parent of the node at index k is $\lfloor k/2 \rfloor$ while the children are at $2k$ and $2k + 1$. We observe that for a general tree, the memory use of such a representation would be $O(2^n)$.

5.4 *N*-ary trees

An N-ary tree binary tree is where each node has N identifiable children. We define an *empty node* or *null sub-tree* as any child which does not exist. A full node is a node with N children. A full N-ary tree is an N-ary tree where every internal node is full. The implementation of such a structure is quite straight-forward, and recursive algorithms can be used to make various calculations such as determining the size and height of trees. One application we consider are tries, 26-ary trees where each branch represents another character in a word being stored. The root represent the empty string, and characters are added to this string as one steps down the tree.

5.5 Balanced trees

The heights of perfect and complete binary trees is $\Theta(\ln(n))$, while in general the height of binary tree is $O(n)$. In general, operations that must access leaf nodes would require us to traverse down the tree, so any such operations would be $O(n)$. We will look at various definitions of balance. In general, if a tree is balanced, it will be shown that the height of the tree is $o(n)$. Usually, this will be $\Theta(\ln(n))$. Height-balancing such as AVL balancing (which we will see) has us restrict the difference in heights of the two sub-trees to at most one. The null-path-length of a tree is the shortest distance to a null sub-tree. Null-path-length balancing has us restrict the difference in the null-path-lengths between the two sides, as is shown in red-black trees. Finally, the weight of a tree is the number of null sub-trees. Consequently, a weight-balanced tree restricts the ratio of the weights of the two sub-trees to a maximum amount. All of these restrictions apply to all nodes within the tree, not just the root.

6. Binary search trees

Next we look at using trees for storing linearly ordered data. We will use ordered trees to themselves define a linear order on the node and their children.

6.1 Binary search trees

A binary search tree is defined where anything less than the current node appears in the left sub-tree, while anything greater than the current node appears in the right sub-tree. Operations can be performed recursively to find, for example, the smallest object, the largest object, and finding an object. Inserting a new node is performed by following the same path one would follow to find that node, and the new node replaces the null sub-tree found. Erase is more difficult in the case of erasing a full node. In this case, either the minimum entry from the right sub-tree or the maximum entry of the left sub-tree can be copied up and that object is recursively removed from the appropriate tree. We discussed how we could implement operations such as find next and accessing the k^{th} entry quickly. All these operations are $\mathrm{O}(h)$. Consequently, in the worst case, the operations are $\mathrm{O}(n)$; however, if we can restrict the height of the tree to $\Theta(\ln(n))$, all these operations will be performed in logarithmic time.

6.2 AVL trees

By requiring that the height of the two sub-trees differs by at most one, the height will be no worse than $\log_{\phi}(n) - 1.3277 = \Theta(\ln(n))$. After an insertion or erase, as one traverses back to the root node, it is necessary to check each node as to whether or not it is balanced. If it is not, there is one of four possible cases, represented by the insertion of 3, 2, 1; 3, 1, 2; 1, 2, 3; and 1, 3, 2 into an empty binary search tree. Each of these can be, with a fixed number of assignments be corrected to be a perfect binary tree of height $h = 1$. Applying these corrections requires at most $\Theta(1)$ time for insertions and $\mathrm{O}(\ln(n))$ time for erases; neither changing the run time of the original operation.

6.3 Multiway search trees

Suppose an ordered trees contains references to N sub-trees interspaced with $N - 1$ elements. In this case, if the $N - 1$ elements are linearly ordered, we can require that all the entries in the k^{th} sub-tree fall between the two surrounding elements, while the left-most sub-tree contains elements less than the left-most element, and the right-most sub-tree contains elements greater than the right-most element. If such a tree is perfect, it allows us to store objects in a tree of height $h = \log_N(n + 1) - 1 = \Theta(\ln(n))$. While such a tree is more complex than a binary search tree, it has the potential to have a height that is a factor of $\log_2 N$ times shorter than a corresponding binary tree.

6.4 B+ trees

A B+ tree is a tree that is used as an associative container. Each leaf node contains up to L objects including keys and the associated information. Internal nodes are multi-way trees where the intermediate values are the smallest entries in the leaf nodes of the second through last sub-trees. If a B+ tree has no more than L entries, those entries are stored in a root node that is a leaf node. Otherwise, we require that leaf nodes are at least half full and all at the same depth, internal nodes are multiway nodes that, too, are at least half full, and the root node is a multiway node that is at least half full. When an insertion occurs into leaf node that is filled, it is split in two, and a new node is added to the parent. If parent is already full, it too is split. This recurses possibly all the way back to the root, in which case, the root node will have to be split and a new root node will be created.

7. Priority queues

In this topic, we will examine the question of storing priority queues. We will look at the abstract data type and we will then continue to look at binary min-heaps. While there are numerous other data structures that could be used to store a heap, almost all are node-based. Given the emphasis on node-based data structures in the previous topics, we will now focus on an array-based binary min-heap. Students are welcome to look at other implementations (leftist heaps, skew heaps, binomial heaps and Fibonacci heaps).

7.1 Priority queue abstract data type

Often, we are faced with an issue of dealing with items in order of their priority relative to each other. Items waiting for a service will arrive at different times, but they must be serviced in order of the priority.

7.2 Binary min-heaps

We require an array-based data structure that can implement the operations relevant to a priority queue in an efficient manner. Storing a min-heap structure (where the children are greater than the parent) allows us to use a complete tree, which has an elegant array-based representation. However, to achieve stability (guaranteeing that two items with the same priority are serviced according to their arrival time) requires $\Theta(n)$ additional memory by creating a lexicographical linear ordering based on an arrival-order-counter and the actual priority.

8. Sorting algorithms

Given an array of unordered entries, we would like to sort the entries so that they are located relative to their linear ordering.

8.1 Introduction to sorting

Given an unsorted list of items that are linearly or weakly ordered, it is often necessary to order the items based on their relative linear ordering. We will assume that the items are stored in an array, and we will define a sorting algorithm to be *in-place* if it uses $\Theta(1)$ additional memory (a few local variables). In some cases, *in-place* is defined if the additional memory is $o(n)$. There are six sorting design techniques we will consider: insertion, selection, exchange, merging, and distribution. We also define a measure of the *unsortedness* of a list, namely, the number of *inversions*. We look at a very brief overview of a proof that if a sorting algorithm uses comparison to perform the sort, the binary decision tree must contain $n!$ items, and the minimum average depth of nodes in a binary tree with N nodes is $\ln(N)$; consequently, the average number of operations is therefore $\Omega(n \ln(n))$.

8.2 Insertion sort

In order to sort a list, we start with a list of size 1—which is, by definition, sorted—and then given a list of size k, it inserts the next item into the list by placing it into the correct location. Naïvely, the algorithm may appear to be $O(n^2)$; however, a better description is $\Theta(n + d)$ where d is the number of inversions. The number of comparisons is exactly $n + d$.

8.3 Bubble sort

While having a name catchy name and using a simple idea that appears to be related to insertion sort, bubble sort performs significantly worse than insertion sort. A naïve implementation of the algorithm has a runtime of $\Theta(n^2)$, and while various modifications to the algorithm, including alternating between bubbling up and sinking down, restricting the search space, etc., the run time can be reduced to $\Theta(n + d)$ where d is the number of inversions, but unlike insertion sort, the number of comparisons is $n + 1.5d$.

8.4 Heap sort

In order to sort a list, we could consider using a priority queue. For example, inserting n items into a binary min-heap using their value to represent their priority requires, at worst, $\sum_{k=1}^{n} \ln(n) = \Theta(n \ln(n))$ time, and taking those same n items out again takes the same amount of time. However, the items will come out of the heap in order of their values; consequently, the items will come out in linear order. The only issue is that this requires $\Theta(n)$ additional memory. Instead, converting the array of unsorted items into a binary max-heap, popping the top n times, and placing the popped items into the vacancy left as a result of the pop allows us to sort the list in place.

8.5 Merge sort

Another approach to sorting a list would be to split the list into two, sort each of the halves, and then merge the two sorted lists back into one complete sorted list. Merging two lists of size $n/2$ requires $\Theta(n)$ time. If merge sort is applied recursively, we may describe the run time as $T(n) = 2T(n/2) + \Theta(n)$ when $n > 1$, but this has the additional consequence that the run time is now $\Theta(n \lg(n)) = \Theta(n \ln(n))$. We will consider exactly why the runtime is reduced to this amount when we consider the master theorem in our topic on divide-and-conquer algorithms. Unfortunately, the merging process requires an additional Θ(n) memory over-and-above the original array.

8.6 Quicksort

The most significant issue with merge sort is that it requires Θ(n) additional memory. Instead, another approach would be to find the median element and then rearrange the remaining entries based on whether they are less than or greater than the median. We can do this in-place in Θ(n) time, in which case, the run-time would be equal to that of merge sort. Unfortunately, selecting the median is difficult at best. We could randomly choose a pivot, but this would have a tendency of dividing the interval in a ratio of 1:3, on average. There would also be a higher likelihood of

9. Hash tables and relation-free storage

9.1 Introduction to hash tables

When we store linearly ordered data that we intend to both access and modify at arbitrary locations, this requires us to use a tree structure that ultimately requires most operations to run in $O(\ln(n))$ time—the linear ordering prevents us from having run-times that are $o(\ln(n))$. If, however, we don't care about the relative order (What comes next? What is the largest?), we don't need the tree structure. Instead, we just need a simple formula—a hash value—that tells us where to look in an array to find the object. The issue is, it is very difficult to find hash functions that generate unique hash values on a small range from 0 to $M - 1$, so we must deal with collisions. Our strategy will be to find $\Theta(1)$ functions that first map the object onto a 32-bit number (our hash value), this hash value is mapped to the range $0, \ldots, M - 1$, and then deal with collisions.

9.2 Hash functions

We are going to define hash functions as functions that take an object that deterministically takes us to a 32-bit value. For example, the address of an object is a 32-bit value which is fine so long as different objects are considered to be different under our definition. On the other hand, it is also possible to simply assign each newly created object a unique hash value in the constructor. For certain objects, two instances may be equal under a specific definition (two strings both containing the characters "Hi!"), in which case, the hash function must be an arithmetic function of the properties of the object that distinguish it in such a way that two equal objects have the same hash value.

9.3 Mapping down to 0, …, M – 1

We are storing objects in an array of size M; consequently, we must map the hash value to that range. Just taking the value modulo M is sub-optimal for a number of reasons. Instead, if we restrict ourselves to arrays that are powers of two ($M = 2^m$), it is better to multiply the hash value by a large prime, and then take the middle m bits.

9.4 Chained hash table

The easiest way to deal with collisions in a hash table is to associate each of the n bin with a linked list or other container. All operations insertion, access, modifications, and erases are performed on the individual containers associated with the possible hash values. Thus, if the operations on the original were, for example, $O(n)$, the corresponding operations on the individual items would be $O(\lambda)$, where λ is the load factor.

9.6 Open addressing

One issue with using, for example, chained hash tables or scatter tables is that they require $\Theta(m + n)$ additional memory over-and-above memory required to store the items. This is because an explicit indicator is being used to store a reference to the next bin. Alternatively, an implicit rule could be used to indicate the next location to search.

9.7 Linear probing

The easiest rule to follow is to check successive cells in order. In this case, determining membership requires us to also follow the same rule until we find the object, find an empty cell, or (in the worst case) iterate through the entire array. Each bin must be marked as either `OCCUPIED` or `UNOCCUPIED`. When erasing, bins cannot be left empty if there is an object where the empty bin lies on the search path between the original bin and the bin it is currently stored in. One issue with linear probing is that it leads to primary clustering: once clusters start growing, they accelerate in their growth leading to longer run times.

9.9 Double hashing

One solution to the issue of primary clustering seen with linear probing is to give each object a different jump size. The easiest way to do this is to calculate for each object a second hash value. This jump size must be co-prime (relatively prime) with the array size. For array sizes that are powers of two, this requires the jump size to be odd. One issue with such a scheme is that it is no longer possible to efficiently determine during an erase what other objects may be appropriately placed into a particular bin. Consequently, each bin must be marked as `OCCUPIED`, `UNOCCUPIED` or `ERASED`.

10. Equivalence relations and disjoint sets

An equivalence relation allows one to break a larger set into equivalence classes. If the larger class is finite, an equivalence class breaks the set into finite disjoint sets. We look at how it is possible to represent a set broken into disjoint subsets.

10.1 Disjoint sets

If we want to break a finite set of *n* objects into disjoint sets, the easiest data structure is a parental forest where one states that two objects are in the same set if the they have the same root. Two disjoint sets are joined to form a single set by making the shorter tree a sub-tree of the other. Now, all objects in both trees have the same root. On average, given random insertions, the height is essentially $\Theta(1)$, but in the worst case, the height is $O(\ln(n))$.

11. Graph algorithms

A graph stores adjacency relations, where two nodes are considered connected or not. We describe graphs, mechanisms for storing graphs, and algorithms for solving problems related to graphs.

11.1 Graph theory and graph ADTs

A graph is a collection of vertices V and edges E (the number of vertices and edges is denoted by $|V|$ and $|E|$). In an undirected graph, edges are ordered pairs of unique vertices. Therefore, the maximum number of edges is $|E| \leq \binom{|V|}{2} = \frac{|V|(|V|-1)}{2} = \mathrm{O}(|V|^2)$. If a vertex is one of the two end-points of an edge, that vertex is said to be coincident with the edge. We define sub-graphs and vertex-induced sub-graphs, paths, simple paths, cycles, and simple cycles. We define weighted graphs, were each edge has a weight associated with it. We describe how a graph can be a *tree* or a *forest* and the concept of an acyclic graph—one with no cycles. We then define directed graphs and modify the definitions seen previously for undirected graphs. We then quickly cover the means of storing graphs and some general questions that may be asked of abstract graph.

11.2 Graph data structures

A graph can be stored efficiently as either an adjacency matrix or an adjacency list. An undirected graph can also be stored as a lower-triangular matrix. A sparse matrix format may be useful in storing an adjacency matrix if the number of edges is $\mathrm{O}(|V|)$.

11.4 Topological sorts

A directed acyclic graph is a representation of a partial ordering on a finite number of vertices. A topological sorting of the vertices in a directed acyclic graph is one such that $v_1 < v_2$ in the topological sort whenever $v_1 \prec v_2$ in the partial ordering. In fact, a graph is a DAG if and only if it has a topological sorting and this is based on the fact that every DAG has at least one vertex of in-degree zero. The algorithm for generating a topological sorting is similar: sequentially remove vertices that have in-degree zero. Initialize an array of in-degrees, and place all objects with in-degree zero into a queue. Then, pop the vertices from the queue and decrement the in-degrees of each adjacent neighbor. Any neighbor that has its in-degree decremented to zero is also placed into the queue. After $|V|$ iterations, we have a topological sorting and the run time is $\Theta(|V| + |E|)$ requiring $\Theta(|V|)$ memory.

11.5 Minimum spanning trees

A minimum spanning tree of a connected graph is a collection of $|V| - 1$ edges such that the sub-graph containing those edges is still connected. Removing an edge from a spanning tree produces an unconnected graph. Adding an edge to a spanning tree produces a cycle. If a graph is not connected, it is possible to find a minimum spanning forest, where there is a minimum spanning tree for each connected sub-graph. Minimum spanning trees can be used, for example, in power distribution, both on chips as well as high-voltage power transmission. While two algorithms are common, Kruskal's and Prim's, we will focus on the latter.

11.5*a* Prim's algorithm

Prim's algorithm finds a minimum spanning tree by starting with a vertex. Then, given a sub-graph for which we have found a minimum spanning tree (which we have with just a single vertex), it considers all edges where one vertex is in the sub-graph and the other is not. Of all of these edges, the edge with minimum weight is added and the corresponding vertex is added to the sub-graph. After $|V|$ iterations, the minimum spanning tree is generated. If at some point, there are no edges touching the sub-graph, the original graph was not connected. The algorithm starts with three arrays with one entry per vertex: the first is a Boolean array with each entry initialized to false (the vertex has not yet been *visited*), the second is a distance array with each entry initialized to infinity with the exception of the initial vertex which has this entry set to 0, and the third array is a reference to another vertex where each is initialized to null. Then we iterate: find a vertex v that has not been visited. Mark v as visited and for each adjacent vertex w, check if the edge between v and w is less than the current distance associated with w. If it is less, update the distance to that value and set the vertex reference of w to v. If at some point all vertices are visited, we are done and we have a minimum spanning tree. If at some point every unvisited vertex has distance infinity, the graph is unconnected. The array of references to vertices forms a parental tree. The run-time of the algorithm is $\Theta(|E| \ln(|V|))$ and it requires $\Theta(|V|)$ memory.

11.6 Single-source shortest path

Given a vertex v in a graph, we may want to find the shortest distance from that vertex to every other vertex in the same connected component of that graph. This is a slightly simpler problem than that of finding the distance from each vertex to every other vertex.

11.6*a* Dijkstra's algorithm

Dijkstra's algorithm finds the minimum distance to each node by starting out with finding the closest node to the source, and then asking whether or not it is possible to get to any other nodes more quickly through that node. Then, given a sub-graph of vertices for which the algorithm has found the minimum distance to all vertices within the graph, it considers the distances to all neighboring vertices by adding up the distance to the vertex in the sub-graph plus the weight of the connecting edge. Of all of these edges, the vertex with the minimum distance chosen and it has been determined that the minimum distance from the source to that vertex has been found. After $|V|$ iterations, the distance to all vertices is found. If at some point, there are no edges touching the sub-graph, the original graph was not connected. The algorithm starts with three arrays with one entry per vertex: the first is a Boolean array with each entry initialized to false (the vertex has not yet been *visited*), the second is a distance array with each entry initialized to infinity with the exception of the initial vertex which has this entry set to 0, and the third array is a reference to another vertex where each is initialized to null. Then we iterate: find a vertex v that has not been visited. Mark v as visited and for each adjacent vertex w, add the distance to v and add to that the weight of the edge between v and w. Let this be the distance to w through the vertex v. If it is less than what is currently recorded, update the distance to that value and set the vertex reference of w to v. If at some point all vertices are visited, we are done and we have a minimum spanning tree. If at some point every unvisited vertex has distance infinity, the graph is unconnected. The array of references forms a parental tree with the initial vertex at the root. The run-time of the algorithm is $\Theta(|E| \ln(|V|))$ and it requires $\Theta(|V|)$ memory.

12. Algorithm design

We have now seen a number of algorithms that solve various problems. We will now look at trying to describe the strategies used to solve these problems.

12.1 Algorithm design

Problems may have only one solution or perhaps many possible solutions. In general, a solution may be comprised of numerous components that together constitute the complete solution. For example, directions from Waterloo to Ottawa include solutions bring one from one decision point to the next—a decision point being a location where one must choose one of two or more possible paths; for example, intersections. A partial solution is a combination of components that could be extended to be a solution. A feasible solution is one that satisfies all possible requirements for the solution. An optimal solution would be defined as the best feasible solution according to some metric.

12.2 Greedy algorithms

A greedy algorithm is one where we use a very simple rule to extend a partial solution, usually starting with a *null* solution—one that contains no components toward a feasible solution. At each step, a relatively simple algorithm is used to extend the partial solution. The goal is that the sequence of partial solutions leads to a feasible solution. If there are numerous feasible solutions together with an optimality condition, the goal is that the sequence of partial solutions will lead to either an optimal or near optimal solution. Examples include Prim's algorithm, Kruskal's algorithm and Dijkstra's algorithm. In the 0/1 knapsack problem, any partial solution that does not exceed the maximum weight is a feasible solution. A greedy algorithm in this case can be shown to not necessarily lead to the optimal solution. The function deciding which is the next item to be put into the knapsack will also affect the result, with a density function being better than focusing only on the value or the constraining factor. We looked at this using the analogy of a project management problem. Another case where a greedy algorithm is optimal is minimizing the overall wait time of a number of processes by scheduling the shortest job next. Similarly, finding the maximal number of intervals that can be chosen so that none overlap may be found using a earliest-deadline-next greedy algorithm; however, if the intervals have weights, no greedy algorithm is known that will maximize the total weight of the chosen intervals.

12.3 Divide-and-conquer

Often, a problem can be solved by breaking a problem into smaller sub-problems, finding solutions those smaller sub-problems, and then recombining the results to produce a solution for the overall problem. Two questions are: Can we determine when it is beneficial to use a divide-and-conquer algorithm? What approaches should we be using to increase the efficiency of a divide-and-conquer strategy? In finding the maximum entry of a sorted square matrix, a divide-and-conquer algorithm is sub-optimal when contrasted with a linear search. When multiplying two n-digit numbers, Karatsuba's algorithm reduces the problem to multiplying three sets of $n/2$-digit numbers, yielding a significant reduction in run time. When multiplying two $n \times n$ matrices, Strassen's algorithm reduces the problem to multiplying seven pairs of $n/2 \times n/2$ matrices. A naïve divide-and-conquer algorithm for matrix-vector multiplication reduces the product to four products of $n/2 \times n/2$ matrices with $n/2$-dimensional vectors. This does not reduce the run time—it is still $\Theta(n^2)$; however, in the special case of the Fourier transform, the matrix is such that the matrix-vector product can be reduced to two products of $n/2 \times n/2$ matrices with $n/2$-dimensional vectors resulting in a run time of $\Theta(n \ln(n))$. The master theorem allows us to write

$$T(n) = \begin{cases} 1 & n=1 \\ aT\left(\frac{n}{b}\right) + \mathbf{O}(n^k) & n>1 \end{cases} = T(b^m) = \begin{cases} 1 & n=1 \\ aT(b^{m-1}) + \mathbf{O}\left((b^k)^m\right) & n>1 \end{cases} = a^m \sum_{\ell=0}^{m} \left(\frac{b^k}{a}\right)^{\ell}$$

and then based on whether $a > b^k$, $a = b^k$ or $a < b^k$, the run times may be determined to be $\mathrm{O}\left(n^{\log_b(a)}\right)$, $\mathrm{O}\left(n^{\log_b(a)} \ln(n)\right) = \mathrm{O}\left(n^k \ln(n)\right)$, or $\mathrm{O}\left(n^k\right)$, respectively.

12.4 Dynamic programming

A divide-and-conquer algorithm is recursive. In other cases, recursive algorithms may result in result in sub-problems that are repeatedly solved. For example, in the definition of the Fibonacci numbers or in the definition of the coefficients of Newton polynomials, the naïve recursively defined functions run in exponential time. However, because so many results are repeatedly calculated, if such intermediate calculations are temporarily stored so that they may be immediately returned upon subsequent calls, the run time drops to $\Theta(n)$ and $\Theta(n^2)$, respectively. Storing intermediate results is a process called *memoization*. In these two examples, the algorithm is formulaic; in others, the recursive query may be asking to find an optimal solution for a specific sub-problem. In such cases, memoization still provides a significant speed-up. We considered matrix chain multiplications, interval scheduling with weights, and the knapsack problem. Another observation is all of these algorithms using memoization are *top-down*. It is also possible to simply determine the simplest cases and build the solution up from simpler sub-problems. This describes a bottom-up approach. Which is optimal depends on the problem at hand. One interesting observation we made is that the matrix-chain multiplication problem and the optimal triangulation problem, while appearing to be different, are very similar problems and solving one allows one to solve the other.

12.4 Dynamic programming

A divide-and-conquer algorithm is recursive. In other cases, recursive algorithms may result in result in sub-problems that are repeatedly solved. For example, in the definition of the Fibonacci numbers or in the definition of the coefficients of Newton polynomials, the naïve recursively defined functions run in exponential time. However, because so many results are repeatedly calculated, if such intermediate calculations are temporarily stored so that they may be immediately returned upon subsequent calls, the run time drops to $\Theta(n)$ and $\Theta(n^2)$, respectively. Storing intermediate results is a process called *memoization*. In these two examples, the algorithm is formulaic; in others, the recursive query may be asking to find an optimal solution for a specific sub-problem. In such cases, memoization still provides a significant speed-up. We considered matrix chain multiplications, interval scheduling with weights, and the knapsack problem. Another observation is all of these algorithms using memoization are *top-down*. It is also possible to simply determine the simplest cases and build the solution up from simpler sub-problems. This describes a bottom-up approach. Which is optimal depends on the problem at hand. One interesting observation we made is that the matrix-chain multiplication problem and the optimal triangulation problem, while appearing to be different, are very similar problems and solving one allows one to solve the other.

12.5 Backtracking

A brute-force search tends to be very expensive. If the solutions can be hierarchically ordered, it may be possible to define a traversal which walks through the tree to find the optimal solution. Such an algorithm would, again, be expensive; however, it may be possible to determine at a specific node that all descendants from that point are unfeasible solutions; in this case, we may disregard searching that entire sub-tree. Depending on how quickly the traversal stops, it may be possible to significantly reduce the number of partial solutions that need to be checked to find an optimal solution. We considered problems such as Sudoku, the *n*-queens problem, the knight's tour problem, and parsing a string with respect to a grammar.

12.6 Stochastic algorithms

In some cases, it may be reasonable to use either random or pseudo-random to find or approximate solutions to problems. First, we discuss random number generation, specifically the 48-bit class of random number generators including drand48(), lrand48() and mrand48(). We then discuss how random numbers that follow a standard normal can be calculated. This is followed by an example of Monte Carlo integration, and how randomly assigning errors to circuit components can be useful in determine if a circuit is robust; that is, if small errors do not cause significant differences in the output. Finally, we discuss skip lists, a data structure that combines the best characteristics of arrays and linked lists.

13. Theory of computation

We will now look at some of the theoretical aspects of algorithms to answer questions such as: What can we solve? Is there anything we cannot solve? What can we solve efficiently?

13.1 Theory of computation

We have investigated many abstract data types (ADTs) and considered many implementations of these ADTs. We have also investigated many problems that may be posed

13.2 Turing completeness

To answer the question "what can we solve?", Alan Turing devised a simple machine that maintained a state from a finite set of states, that had a head that could read or write to a single tape divided into cells, and where each cell could contain a character from a finite alphabet. The head can also move the tape one cell to the left or right. This machine then looks up the state and the character currently located in the cell under the head, and from this reads off an instruction that indicates the new state, the letter to be written to the cell, and whether the head should remain over the current cell, move one cell to the right, or move one cell to the left. Turing showed that this machine could be used to implement a broad spectrum of algorithms and the Church-Turing thesis says that anything that can be computed can be computed by a Turing machine. While this has not been proven, no counter examples have yet been discovered. Any machine that can perform at least the operations of a Turing machine is said to be *Turing complete*. Essentially, any computer with a register, linear memory, and instructions allowing both to be changed to reasonably arbitrary values is Turing complete.

13.3 Decidability and the halting problem

Are there questions we cannot decide? To answer this in the affirmative, we pose the halting problem: "Is it possible to, given a function and arguments to be passed to that function, determine whether or not that function will terminate at some finite time in the future?" If we assume that such a halting function exists, then we can create a function that, no matter what such a halting function returns, that returned value will be incorrect; consequently, showing that we cannot find such a function. Therefore, yes, there are some questions which we will never be able to compute.

13.4 Decidability and the halting problem

Are there questions we cannot decide? To answer this in the affirmative, we pose the halting problem: "Is it possible to, given a function and arguments to be passed to that function, determine whether or not that function will terminate at some finite time in the future?" If we assume that such a halting function exists, then we can create a function that, no matter what such a halting function returns, that returned value will be incorrect; consequently, showing that we cannot find such a function. Therefore, yes, there are some questions which we will never be able to compute.

14. Other topics

Other topics include a discussion on sparse matrices, searching algorithms and the standard template library (STL). We will look only at the first.

14.1 Sparse matrices

A sparse $M \times N$ matrix is generally one that contains $o(MN)$ entries, and usually $\Theta(M + N)$ entries. The old Yale sparse matrix format stores the entries in row-major order. This allows us to store the n entries in an array of size n, with an indication of the column in another array of size n, together with an array of size $M + 1$ indicating the position in the array where the k^{th} row begins. Access time is now $\Theta(\ln(n/M))$.