The order of topics and the code snippets in these notes can differ from the lecture content. Information may be inaccurate. This should primarily be used for reviewing rather than learning course content, and as a supplement rather than a replacement for doing practice problems.

C++ Nuances and Features

These notes have code snippets written in C++. Here are brief notes about some nuances features of the C++ programming language that may be useful to know (read documentation for more details!).

C vs. C++

C++ is an extension of the C programming language. One implication of this is that any valid C program is also a valid C++ program, though the opposite is not necessarily true. However, C++ adds many useful features that can make coding easier, so these notes use C++.

Indentation

It's good practice to indent code by adding tabs/spaces indicating which indicates that certain lines of code are inside a function body, if-statement, for-loop, etc. But technically indentation isn't necessary for code to compile; it's solely to make the code more readable to humans.

Types

The type of data that is stored in a variable must be specified when that variable is declared, and the compiler more or less enforces that only that type of data is stored in the variable throughout the variable's lifetime. This can help make debugging easier and code execution faster. Some primitive data types include `bool` (true or false), `int` (integers), `double` (numbers with decimals), and `char` (characters like `'a'`).

The main Function

Every C++ program must have a function called `main` with the following signature:

```
int main(int argc, char* argv[]) {
    // ...
}
```

`main` is the entry point for every C++ program, meaning it is always the first function that runs. `argc` (short for "argument count") is the number of command-line arguments and `argv` (short for "argument values") is an array of the command-line arguments passed to the program. `main` returns 0 if the program exits normally, or should return a nonzero value otherwise.

Pre-increment vs. Post-increment Operators

The following left snippet uses the pre-increment operator, and the right uses post-increment.

```
int x = 5;
int y = ++x;
std::cout << y; // 6
```

```
int x = 5;
int y = x++;
std::cout << y; // 5
```

The pre-increment operator increments the variable first, and then returns its value. The post-increment returns the variable's value first, and then increments it. This usually only makes a difference when the variable being incremented is being used in a larger statement (as opposed to a standalone statement), such as the above case assigning a variable to the value of another variable being incremented, or when printing out the value of a variable being incremented. The following two snippets of code are behaviorally equivalent:

```
for (int i = 0; i < 3; ++i) {
    std::cout << i << " "; // 0 1 2
}
```

```
for (int i = 0; i < 3; i++) {
    std::cout << i << " "; // 0 1 2
}
```

The same ideas also apply to the pre-decrement and post-decrement operators. Separating an increment/decrement operation into its own standalone statement may make your code easier to read.

If-Statement and For/While-Loop Braces

Both of the following code snippets are behaviorally equivalent.

```
int x = 5;
if (x == 5) {
    std::cout << "x is 5";
}
```

```
int x = 5;
if (x == 5)
    std::cout << "x is 5";
```

The curly braces create a block that indicates that all of the statements inside the block are subject to the if condition. If no curly braces are included, then the compiler assumes that the single next statement, and only that statement, is subject to the if condition- even if more statements are indented that *look* like they're inside the if-statement. The same applies to loops as well; the following snippets are behaviorally equivalent.

```
for (int i = 0; i < 3; i++) {
    std::cout << i << " "; // 0 1 2
}
```

```
for (int i = 0; i < 3; i++)
    std::cout << i << " "; // 0 1 2
```

If you want more than one statement to be inside the if-statement/for-loop, then you need curly braces. In general, it's best practice to always use curly braces regardless of whether they're needed, but these notes typically don't use curly braces when they're not needed to make the code snippets more concise.

Pointers

A pointer is a variable that stores a memory address, typically the address of where some meaningful value is stored (a pointer can also be assigned `nullptr` to indicate that it's not yet pointing to a meaningful value). Both of the following snippets show valid ways to declare a pointer to an integer.

```
int x = 5;
int* ptr = &x; //&x is x's address
```

```
int x = 5;
int *ptr = &x; //&x is x's address
```

Pointers with valid memory addresses can be dereferenced to retrieve the value of what is stored at the memory address as follows.

```
int x = 5;
int* ptr = &x;
std::cout << *ptr; // 5
```

References

A reference is an alias for a variable. Both of the following snippets show valid ways to declare a reference to an integer variable.

```
int x = 5;
int& y = x;
```

```
int x = 5;
int &y = x;
```

In the following left snippet, y is another name for x, while in the right snippet, y is a copy of x but is not the same as x.

```
int x = 5;
int& y = x;
++x;
std::cout << y; // 6
```

```
int x = 5;
int y = x;
++x;
std::cout << y; // 5
```

When passing in a variable into a function, by default it's passed in by value, meaning a copy of the value of the variable is passed into the function. But function parameters can also be references, which are useful for modifying the actual variable that is passed in, or avoiding making an expensive copy of a variable whose value is really large.

```
void addOne(int x) {
    x = x + 1;
}
int x = 5;
addOne(x);
std::cout << x; // 5
```

```
void addOne(int& x) {
    x = x + 1;
}
int x = 5;
addOne(x);
std::cout << x; // 6
```

Structs vs. Classes

Structs and classes both let us define our types that contain member variables and member functions. The only difference between them is the default access level: in a struct, members are public by default (accessible from outside the type), while in a class, members are private by default. The following snippets are behaviorally equivalent.

```
struct myType {
    void incrementX() {
        ++((*this).x);
    }
private:
    int x = 0;
};
```

```
class myType {
    int x = 0;
public:
    void incrementX() {
        ++((*this).x);
    }
};
```

These notes typically use `struct` when a type only holds data (no member functions), and `class` when the type includes member functions.

The C++ Standard Template Library (STL)

The STL is a feature of C++ that provides high-quality implementations of general-purpose containers, algorithms, and utilities. These are highly optimized and should be the default choice before writing custom data structures.

To use an STL utility, the header file containing the utility must first be included like so (but the code snippets in these notes typically don't write header file inclusions to make the snippets more concise):

```
#include <vector>
std::vector<int> v;
```

You may also see code that starts with `using namespace std`, which would allow you to write `vector<int>` instead of `std::vector<int>`. However, it is generally discouraged because it brings every name from the entire `std` namespace into the global namespace, which can cause name conflicts- for example, if you define your own vector, its name may clash with the one defined in the standard library. A safer alternative is to use fully qualified names (what these notes typically do):

```
std::vector<int> v;
```

Or selectively pull in only what you need:

```
using std::vector;
vector<int> v;
```

Some useful utilities from STL are `std::string` from `<string>`, `std::cin` and `std::cout` from `<iostream>`, `std::vector` from `<vector>`, and `std::size_t` (a type that stores nonnegative integers) from `<cstddef>`.

Resize vs. Reserve

A `std::vector` can be constructed with an initial size.

```
std::vector<int> v(3); // optional- specify values, ex: std::vector<int> v(3, 1)
std::cout << v.size(); // 3
```

If no size is specified, then by default it has a size and capacity of 0. The *size* of a vector is the number of elements currently stored in it. The *capacity* of a vector is the amount of memory that has been allocated for it. If we try to push a new element to a vector whose size is equal to its capacity, then under the hood the vector will double its capacity. However, `std::vector` also has two methods to adjust how much memory it's using, `resize` and `reserve`.

```
std::vector<int> v;
v.resize(3);
std::cout << v.size(); // 3
```

```
std::vector<int> v;
v.reserve(3);
std::cout << v.size(); // 0
```

`resize(n)` changes the vector's size to `n`. If `n` is larger than the current size, the vector creates new default-initialized elements. If `n` is smaller, the vector destroys the extra elements. On the other hand, `reserve(n)` changes the vector's capacity to at least `n`, but does not change its size. No elements are created or destroyed. This is useful when you know you'll be pushing many elements and want to avoid repeated reallocations.

Iterators

An iterator is an object that acts as a generalized pointer, providing a way to access items in a container (including STL containers like `std::vector`) sequentially.

```
std::vector<int> v = {1, 2, 3};
for (std::vector<int>::iterator it = begin(v); it != end(v); ++it)
    std::cout << *it << " "; // 1 2 3
```

The compiler can also deduce the type of an iterator without us having to specify it, which we can instruct it to do using the `auto` keyword.

```
std::vector<int> v = {1, 2, 3};
for (auto it = begin(v); it != end(v); ++it)
    std::cout << *it << " "; // 1 2 3
```

Iterators are also used under the hood to enable range-based for-loops.

```
std::vector<int> v = {1, 2, 3};
for (auto num : v)
    std::cout << num << " "; // 1 2 3
```

Complexity Analysis

Suppose you have an algorithm whose input size is n and, when executing, runs $f(n)$ operations or uses $f(n)$ memory. There's notation for describing how $f(n)$ scales with n.

- $O(g(n))$ (big O of $g(n)$, or O of $g(n)$) is an asymptotic upper bound
  - More formally, $f(n)$ is $O(g(n))$ means there exists some constants $c > 0$ and n_0 such that $0 \leq f(n) \leq c * g(n)$ for all $n \geq n_0$.
- $\theta(g(n))$ (big theta of $g(n)$) is the asymptotic tightest bound
  - More formally, $f(n)$ is $\theta(g(n))$ means there exists some constants $c_1 > 0$ and $c_2 > 0$ and n_0 such that $0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$.
- $\Omega(g(n))$ (big omega of $g(n)$) is an asymptotic lower bound
  - More formally, $f(n)$ is $\Omega(g(n))$ means there exists some constants $c > 0$ and n_0 such that $0 \leq c * g(n) \leq f(n)$ for all $n \geq n_0$.

If $f(n)$ is the number of operations the algorithm runs, then $\theta(g(n))$ is the *time complexity* of the algorithm. If $f(n)$ is the amount of memory the algorithm uses, then $\theta(g(n))$ is the *space complexity* of the algorithm. $g(n)$ should only contain one highest order term because asymptotically that one term will dominate any smaller term.

$\theta(g(n))$ is also a valid $O(g(n))$ and $\Omega(g(n))$. Between the three bounds, $O(g(n))$ is the most commonly used, but usually when people ask for or provide $O(g(n))$, they really want or mean $\theta(g(n))$. For example, while it is technically true that binary search is $O(n!)$, that doesn't give us much useful information and it would be better to say that binary search is $O(log(n))$ since that is the tightest bound.

Asymptotically, $O(1) < O(log(n)) < O(n) < O(n * log(n)) < O(n^2) < O(c^n) < O(n!)$, where $c > 1$. You should aim to minimize the time and space complexities of the programs you write, though often there is a tradeoff between speed and memory, meaning that attaining faster speed requires using more memory, or using less memory causes slower speed. Also note that the worst-case, average-case, and best-case time and space complexities can differ.

Amortized Complexity

Amortized complexity is a type of worst-case complexity that is used to analyze the complexity of an operation that is sometimes expensive but usually cheap to perform. Amortized analysis is done by examining a sequence of the operation, with the sequence usually containing at least one instance of the expensive case and as many instances of the cheap case of the operation as possible. The amortized complexity then is the average cost of the operation over the sequence of operations.

An example of amortized complexity analysis is determining the time complexity of the push operation of `std::vector`. Under the hood, a vector is implemented using a C array, and the array has a fixed amount of memory allocated to it. If we have enough memory available,

pushing an item to the array is easy- we write the new item into the next free location in $O(1)$ time. But if the array is full, we must first allocate memory for a new larger array, copy the items from the old array to the new array, and then perform the push.

A key design choice is how much larger the new array should be than the old array. If the old array is size n and we decide that the new array should be a constant size c larger, then the amortized time complexity of push is:

$$\frac{(1+\theta(n))+c*\theta(1)}{c} = \theta(n)$$

But if we decide to make the new array double the size of the old array by making it n larger:

$$\frac{(1+\theta(n))+n*\theta(1)}{n} = \theta(1)$$

Doubling the array size keeps the amortized time complexity of push constant even though an individual push may occasionally require an expensive resize, which is why the STL uses this strategy.

The Master Theorem

The Master Theorem is a mathematical shortcut for finding the asymptotic time complexity of recurrence relations that match a specific format. More specifically,

given a recurrence relation of the form

- $T(n) = aT(\frac{n}{b}) + f(n)$

Where

- $T(1) = c_0$ or $T(0) = c_0$
- $a \geq 1$
- $b > 1$
- $f(n)$ is a polynomial function with degree c
  - c can be 0; a constant is a valid polynomial.

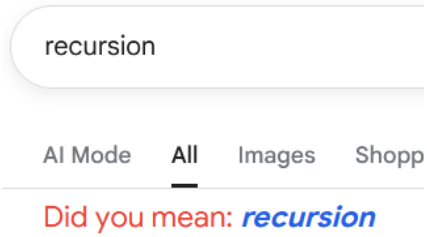
Then $T(n) \in$

- $\theta(n^{log_b a})$ if $a > b^c$
- $\theta(n^c * log(n))$ if $a = b^c$
- $\theta(n^c)$ if $a < b^c$

Recursion

A recursive function is a function that calls itself. It must have:

- At least one base case- a smallest-possible input whose result is already known, and it must be the first thing checked in the function's definition
- At least one recursive case- for an input that is not a base case, it performs some work and makes a recursive call on a strictly smaller part of the input, and then uses the result of that recursive call to compute its own result



The base case must truly be the smallest possible input, and each recursive call must receive a strictly smaller input. This is to ensure that each recursive call moves us closer to the base case so that the recursion eventually terminates and the initial call can be resolved; otherwise, we can have infinite recursion, which basically amounts to a never-ending loop.

One example of an expression that can be defined recursively is a factorial. Specifically, for a nonnegative integer n, $n!$ $=$

- 1 if $n = 0$ (base case)
- $n * (n - 1)!$ otherwise (recursive case)

Here's what that would look like translated into code.

```
size_t factorial(size_t n) {
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}
```

Tail Recursion

Every function call causes a stack frame (which stores things like the function's inputs, the function's local variables, etc.) to be created and added onto the call stack (not discussed in detail here). The stack frame is removed from the stack when the function returns. Each stack frame uses memory, and the call stack has limited memory, so there can only be so many stack frames on the call stack at any given time. Attempting to add too many stack frames onto the call stack (which can happen via very deep or infinite recursion) can result in a stack overflow.

Recursive functions can be susceptible to creating many stack frames, but sometimes can be written to be *tail recursive*, which is when there is no pending computation after the recursive call. Here's a tail-recursive implementation of factorial.

```
size_t factorialHelper(size_t n, size_t resultSoFar) {
    if (n == 0)
        return resultSoFar;
    return factorialHelper(n - 1, n * resultSoFar);
}
```

```
size_t factorial(size_t n) {
    return factorialHelper(n, 1);
}
```

Tail-recursive functions can basically “re-use” or share their caller’s stack frame, and are comparable to iteration in terms of stack usage. You should aim to make recursive functions tail-recursive when possible to improve memory usage and avoid stack overflows.

Abstract Data Types (ADTs)

An abstract data type is a theoretical concept that combines data (i.e. expected types, legal ranges of values) with a collection of valid operations (e.g. insert, remove) and their behaviors on that stored data. ADTs define interfaces, and data structures provide concrete implementations of ADTs. A data structure that stores a collection of items is called a container.

Stacks

A stack is a container ADT that supports the following operations. The stated time complexities are for an efficient implementation.

Method	Description	Time complexity
void push($item$)	Add $item$ to the top of the stack	$O(1)$ amortized
void pop()	Remove the item at the top of the stack	$O(1)$
item& top()	Return a reference to the item at the top of the stack	$O(1)$
size_t size()	Return how many items are in the stack	$O(1)$
bool empty()	Return whether the stack has any items (i.e., its size is 0)	$O(1)$

It is said that stacks use LIFO (Last In, First Out) ordering because the last item pushed onto the stack is the first item popped off of the stack. Examples of stacks are a can of Pringles, the back button in your browser, the undo feature in your text editor, and the call stack in C++. Note that unlike arrays/vectors, stacks do not need to support efficient random access; they only need to provide access to the item at the top of the stack. Still, they can be useful for algorithms such as traversing a tree and generating permutations.

Stacks can be implemented pretty easily with efficient operations using either array/vector or linked list data structures, though in practice arrays are faster. Here’s how stacks can be declared in C++.

```
#include <stack>

std::stack<Item> s;  // uses std::deque<Item> as underlying container by default
std::stack<Item, std::vector<Item>> s;  // or specify a different container
```

Queues

A queue is a container ADT that supports the following operations. The stated time complexities are for an efficient implementation.

Method	Description	Time complexity
void push($item$)	Add $item$ to the back of the queue	$O(1)$ amortized
void pop()	Remove the item at the front of the queue	$O(1)$
item& front()	Return a reference to the item at the front of the queue	$O(1)$
size_t size()	Return how many items are in the queue	$O(1)$
bool empty()	Return whether the queue has any items (i.e., its size is 0)	$O(1)$

It is said that queues use FIFO (First In, First Out) ordering because the first item pushed into the queue is the first item popped out of the queue. Examples of queues are waiting in line for office hours and adding songs to a playlist. Like stacks, queues do not need to support efficient random access; they only need to provide access to the item at the front of the queue.

Queues can be implemented with efficient operations using either deque or linked list data structures, though in practice arrays are faster.

Here's how queues can be declared in C++.

```
#include <queue>

std::queue<Item> q;  // uses std::deque<Item> as underlying container by default
std::queue<Item, std::list<Item>> q;  // or specify a different container
```

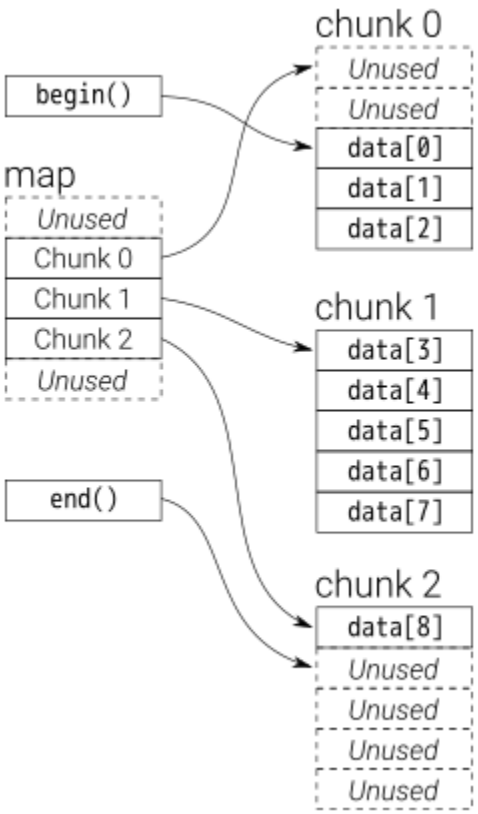
Deques

A deque (pronounced “deck”, short for double-ended queue) is a container ADT that supports the following operations. The stated time complexities are for an efficient implementation.

Method	Description	Time complexity
void push_front($item$)	Add $item$ to the front of the deque	$O(1)$ amortized
void push_back($item$)	Add $item$ to the back of the deque	$O(1)$ amortized
void pop_front()	Remove the item at the front of the deque	$O(1)$
void pop_back()	Remove the item at the back of the deque	$O(1)$
item& front()	Return a reference to the item at the front of the deque	$O(1)$
item& back()	Return a reference to the item at the back of the deque	$O(1)$
size_t size()	Return how many items are in the deque	$O(1)$
bool empty()	Return whether the deque has any items (i.e., its size is 0)	$O(1)$

Because deques can support all of the operations of a stack and a queue efficiently, they are often used to implement stacks and queues. The STL implementation of deques also overloads the [ ] operator to support efficient random access in $O(1)$ time.

To implement a deque, per StackOverflow, “[a] deque is somewhat recursively defined: internally it maintains a double-ended queue of *chunks* of fixed size. Each chunk is a vector, and the queue (“map” in the graphic [to the right]) of chunks itself is also a vector.”



Here’s how deques can be declared in C++.

```
#include <deque>

std::deque<Item> d;
```

Priority Queues

A priority queue is a container ADT that supports the following operations. The stated time complexities are for an efficient implementation. n is the number of items in the priority queue.

Method	Description	Time complexity
void push($item$)	Add $item$ to the priority queue	$O(log(n))$
void pop()	Remove the highest priority element from the priority queue	$O(log(n))$
const item& top()	Return a reference to the highest priority element in the priority queue	$O(1)$
size_t size()	Return how many items are in the priority queue	$O(1)$
bool empty()	Return whether the priority queue has any items (i.e., its size is 0)	$O(1)$

In some other programming languages, we can add pair items to a priority queue that contain an element and its explicitly defined priority. However in C++, we instead use a comparator to define how two elements should be compared to determine which element has the higher priority. If we don't define a custom comparator then `std::less` will be used by default, which causes the greater item to have a higher priority.

Examples of priority queues include hospital queues (prioritizing treating patients who most urgently need care first) and load balancing on servers.

Priority queues can be implemented using multiple different data structures, though implementing them using the heap data structure allows for efficient push and pop operations.

Here's how priority queues can be declared in C++.

```
#include <queue>

std::priority_queue<int> pq; // .top() is always the largest number
// if we use a different comparator than the default one,
// we must also specify what underlying container to use
std::priority_queue<int, std::vector<int>, std::greater<int>> pq; // .top() is
always the smallest number
```

Heaps

(A lot of the notes in this section use tree terminology from the Binary Trees section.)

A heap is a data structure where each element has a *priority* value (a notion of importance used to rank elements), and elements are arranged based on their priorities, with the element having the highest priority always being at the front/top. Thus, it allows easy access to the most extreme element. Heaps are often implemented logically as a tree because it allows for efficient implementations of heap operations, so a heap can be thought of as a tree with two properties:

- Completeness- it is a complete binary tree
- (Max) heap-ordering- the priority of each node in the tree is not greater than the priority of its parent.

A tree that maintains these properties guarantees that the highest priority element is at the root of the tree which can be inspected in $O(1)$ time. The completeness property ensures that the binary tree can be represented as an array (see the Binary Trees section for how). The heap-ordering property ensures that insert/remove/inspection operations are efficient.

Modifying Heaps

When a heap is represented as an array, we can directly modify the items in the heap. However, the modification could break the heap-ordering property, and so we may need to fix the tree. This is done differently depending on whether we increase or decrease the item in the heap.

When the item is increased, we need to keep swapping that item's node with its parent until its parent has a higher priority or it no longer has a parent (i.e., it's the root).

```
void fixUp(Item heap[], size_t k) {
    while (k > 1 && heap[k / 2] < heap[k]) {
        std::swap(heap[k], heap[k / 2]);
        k /= 2;
    }
}
```

When the item is decreased, we need to keep swapping that item's node with its highest-priority child until none of its children have a higher priority or it has no children (i.e., it's a leaf).

```
void fixDown(Item heap[], size_t heapsize, size_t k) {
    while (2 * k <= heapsize) {
        size_t j = 2 * k;
        if (j < heapsize && heap[j] < heap[j + 1])
            ++j;
        if (heap[k] >= heap[j]) break;
        std::swap(heap[k], heap[j]);
        k = j;
    }
}
```

Both `fixUp` and `fixDown` are $O(log(n))$ since each iteration moves up or down one level and a complete tree of size n has a height that is $O(log(n))$. Note that both `fixUp` and `fixDown` preserve the completeness property because they only move nodes around.

Heap Operations

When inserting an item, we must do it in a way that preserves the heap-ordering and completeness properties. This can be done by appending the new item to the end of the tree and fixing it up into a position that preserves the heap-ordering property, making insertion $O(log(n))$. Think about why this also preserves the completeness property.

```
void push(Item newItem) {
    ++heapsize;
    heap[heapsize] = newItem;
    fixUp(heap, heapsize);
}
```

Similar to how inspect can only view the highest priority element of the heap, remove can also only remove the highest priority element of the heap (in the tree structure, it's the root), and like insert it must also be implemented in a way that preserves the heap-ordering and completeness properties. The trick to accomplish this is to make the last item in the heap the new root, then delete the last item from the heap, and finally fix down the new root into a position that preserves the heap-ordering property, making removal $O(log(n))$. Think about why specifically making the last item the root helps preserve the completeness property.

```
void pop() {
    heap[1] = heap[heapsize];
    --heapsize;
    fixDown(heap, heapsize, 1);
}
```

These operations assume that the data is already a valid heap (note that an empty tree is a valid heap), but if it isn't, we can turn it into one by going backwards starting from the last internal node and repeatedly fixing down. This works in-place and in $O(n)$ time, though the proof explaining why this is $O(n)$ is a bit complicated and not included here.

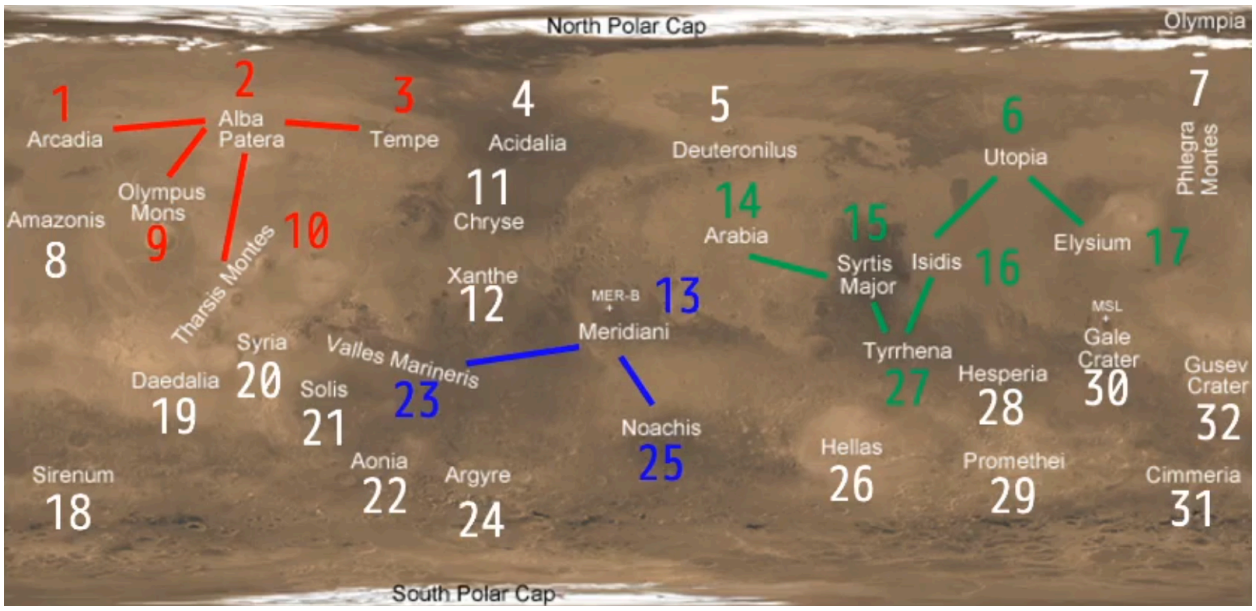
```
void heapify(Item heap[], int n) {
    for (int k = n / 2; k >= 1; --k)
        fixDown(heap, n, k);
}
```

Look at heapsort in the Sorting Algorithms section to see how we can use heaps to sort data.

Union-Find

Union-Find is a data structure that maintains a collection of disjoint sets. Each set can be thought of as containing one or more elements, with elements in the same set being connected/reachable from each other via one unique path. Thus logically each set is also a tree (so they should be thought of as such when reasoning through behavior, though below we represent them as vectors).

In the image below, the red set, blue set, and green set are disjoint. Every other number is in its own set containing just itself.



In this context, there are two operations we would like to be able to do quickly:

- Find- given an element, determine which set it belongs to. Or, given two elements, determine whether they are in the same set.
    - Example- what set does 13 (Meridiani) belong to? (Answer: the same set as 23 and 25)
    - Example- are 3 (Tempe) and 14 (Arabia) in the same set? (Answer: no)
- Union- given two elements, merge the two sets that the elements belong to into one big set.
    - Example- connect 10 (Tharsis Montes) and 23 (Valles Marineris) so that the red set and blue sets become a single unified set.

To help us implement efficient algorithms for these operations, each element will have a representative element associated with it. We can then determine whether two elements are in the same set by comparing their representatives. Think- if two basketball players have the same coach, they are on the same team.

Two elements are in the same set if their ultimate representative is the same. Each set is represented as a tree, where every node points to a representative higher up, and the root is the ultimate representative, whose representative is itself.

To start, every element is in its own set containing only itself and hence every element's representative is itself.

	0	**1**	**2**	**3**	**4**
Representative	0	1	2	3	4

Naive/Most Direct Implementation
Here's one possible implementation of the union-find data structure.

```
class UnionFind {
public:
    UnionFind(size_t n) {
        representative.resize(n);
        for (size_t i = 0; i < n; ++i)
            representative[i] = i;
    }
    size_t find(size_t x) {
        while (representative[x] != x)
            x = representative[x];
        return x;
    }
    void unite(size_t x, size_t y) {    // union is a reserved keyword in C++
        size_t repX = find(x);
        size_t repY = find(y);
        if (repX != repY)
            representative[repY] = repX;
    }
    bool connected(size_t x, size_t y) {
        return find(x) == find(y);
    }
private:
    std::vector<size_t> representative;
};
```

- Find time complexity: $O(n)$
- Union time complexity: $O(n)$

With Rank

To speed up the union operation, we can introduce rank.

```
class UnionFind {
public:
    UnionFind(size_t n) {
        representative.resize(n);
        rank.resize(n, 0);
        for (size_t i = 0; i < n; ++i)
            representative[i] = i;
    }
    size_t find(size_t x) {
        while (representative[x] != x)
            x = representative[x];
        return x;
    }
    void unite(size_t x, size_t y) {
        size_t repX = find(x);
        size_t repY = find(y);
        if (repX == repY) return;
        if (rank[repX] < rank[repY])    // rank is used here
            representative[repX] = repY;
        else if (rank[repX] > rank[repY])
            representative[repY] = repX;
        else {
            representative[repY] = repX;
            rank[repX]++;
        }
    }
    bool connected(size_t x, size_t y) {
        return find(x) == find(y);
    }
private:
    std::vector<size_t> representative;
    std::vector<int> rank;
};
```

- Find time complexity: $O(log(n))$
- Union time complexity: $O(log(n))$

Intuition behind rank: When merging two sets, instead of arbitrarily deciding which root becomes the parent, we use a notion of *rank*- an estimate of tree height- to attach the shorter tree under the taller one. Using rank to determine which tree should be attached under the other helps prevent trees from becoming tall and keeps it more balanced, keeping the height of the merged tree smaller/hierarchy shallower (thus making the path from a node to its ultimate representative/root shorter) and find operations more efficient. Because the tree is pretty balanced, its height is $O(log(n))$, where n is the size of the tree, so find is $O(log(n))$. Union is also $O(log(n))$ because union calls find.

With Rank and Path Compression

To make repeated find operations faster, we can add path compression.

```
class UnionFind {
public:
    UnionFind(size_t n) {
        representative.resize(n);
        rank.resize(n, 0);
        for (size_t i = 0; i < n; ++i)
            representative[i] = i;
    }
    size_t find(size_t x) {
        if (representative[x] != x)
            representative[x] = find(representative[x]); // path compression
        return representative[x];
    }
    void unite(size_t x, size_t y) {
        size_t repX = find(x);
        size_t repY = find(y);
        if (repX == repY) return;
        if (rank[repX] < rank[repY])
            representative[repX] = repY;
        else if (rank[repX] > rank[repY])
            representative[repY] = repX;
        else {
            representative[repY] = repX;
            rank[repX]++;
        }
    }
    bool connected(size_t x, size_t y) {
        return find(x) == find(y);
    }
private:
    std::vector<size_t> representative;
    std::vector<int> rank;
};
```

- Find time complexity: amortized $O(\alpha(n))$, where $\alpha(n)$ is the inverse Ackermann function, which is less than 5 even when n is in the billions- practically constant
- Union time complexity: same as find

Intuition behind path compression: When finding the ultimate representative of an element, we may have to traverse several intermediate nodes before reaching the root. To make future lookups faster, we can update each of those nodes to point directly to the root we just found. This way, the next time we call find on any of those elements, we reach the ultimate representative immediately (again, making the path from a node to its ultimate representative/root shorter).

Sorting Algorithms

Sorting algorithms re-order a container of items based on some specific ordering criteria. This can be done to make the container easier to search and analyze. All of the implementations of the sorting algorithms shown below sort an array of keys in ascending order.

Bubble sort, selection sort, and insertion sort are elementary sorting algorithms, meaning they're simpler but they tend to be slower. Quicksort, heapsort, and merge sort are high-performance sorting algorithms, meaning they're faster but they tend to be more complicated. Counting sort is technically one of the elementary sorting algorithms, but it is a little different from the others in that it can't always be used, and can only be used when there are just a few distinct keys that are consecutive integers or can be converted into integers.

The time complexities written below are specifically for the average-case scenarios. A sorting algorithm is considered stable if the relative ordering of elements is maintained when ties occur, which is particularly important when sorting on multiple keys.

(Adaptive) Bubble Sort

```
void bubbleSort(Item a[], size_t left, size_t right) {
    for (size_t i = left; i < right - 1; ++i) {
        bool swapped = false;
        for (size_t j = right - 1; j > i; --j)
            if (a[j] < a[j - 1]) {
                swapped = true;
                std::swap(a[j - 1], a[j]);
            }
        if (!swapped) break;
    }
}
```

- Time complexity: $O(n^2)$
- Space complexity: $O(1)$
- Stable: yes

Intuition behind bubble sort: `i` can be thought of as denoting that the `i` smallest elements are sorted and in their correct ultimate positions occupying the first `i` positions of the array- this is the sorted part of the array. Each pass of the outer loop "bubbles" the smallest unsorted element to the back of the sorted part of the array, making the sorted part of the array one element larger. Thus after the first pass of the outer loop, the minimum element should be in the correct position (i.e at index 0), making the first element sorted and the rest potentially unsorted. After the second pass, the second-smallest element should be in the correct position (i.e at index 1), making the first two elements sorted and the rest potentially unsorted. This continues until every element is in the correct position. The `swapped` flag is an optimization that makes this implementation adaptive; if the array is already sorted before the outer loop reaches the second-last element, it returns early. Bubble sort is simple but has worse performance than the other sorting algorithms.

(Adaptive) Selection Sort

```
void selectionSort(Item a[], size_t left, size_t right) {
    for (size_t i = left; i < right - 1; ++i) {
        size_t minIndex = i;
        for (size_t j = i + 1; j < right; ++j)
            if (a[j] < a[minIndex])
                minIndex = j;
        if (minIndex != i)
            std::swap(a[i], a[minIndex]);
    }
}
```

- Time complexity: $O(n^2)$
- Space complexity: $O(1)$
- Stable: no

Intuition behind selection sort: like bubble sort, `i` can be thought of as denoting that the `i` smallest elements are sorted and in their correct ultimate positions occupying the first `i` positions of the array- this is the sorted part of the array. Each pass of the outer loop selects the smallest element in the unsorted part of the array (determined using a linear search) and swaps it with the element one after the end of the sorted part of the array, making the sorted part of the array one element larger. Therefore after the first pass of the outer loop, the minimum element should be in the correct position (i.e at index 0), making the first element sorted and the rest potentially unsorted. After the second pass, the second-smallest element should be in the correct position (i.e at index 1), making the first two elements sorted and the rest potentially unsorted. This continues until every element is in the correct position. The `minIndex != i` check is an optimization that makes this implementation adaptive; If an element is already in the correct position, we don't wastefully swap it with itself. Selection sort has the easiest logic to understand but has worse performance than other sorting algorithms and isn't stable.

(Adaptive) Insertion Sort

```
void insertionSort(Item a[], size_t left, size_t right) {
    for (size_t i = right - 1; i > left; --i)
        if (a[i] < a[i - 1])
            std::swap(a[i - 1], a[i]);
    for (size_t i = left + 2; i < right; ++i) {
        Item v = a[i];
        size_t j = i;
        while (v < a[j - 1]) {
            a[j] = a[j - 1];
            --j;
        }
        a[j] = v;
    }
}
```

- Time complexity: $O(n^2)$
- Space complexity: $O(1)$
- Stable: yes

Intuition behind insertion sort: `i` can be thought of as denoting that the first `i` elements are relatively sorted, though not they're not necessarily the `i` smallest elements in the array nor are they necessarily in their correct ultimate positions. Still, the first `i` elements make up what is considered the sorted part of the array. Each pass of the outer loop inserts the first element of the unsorted part of the array into the correct position in the sorted part of the array, making the sorted part of the array one element larger. In this adaptive version of insertion sort, we first "bubble" the smallest element in the array to the front of the array (which acts as a sentinel here), which allows us later in the while loop to not have to check for whether `j` is greater than 0, because we are now guaranteed that any element in the array `v` is not less than `a[0]`. Insertion sort is the most difficult elementary sorting algorithm to understand excluding counting sort, but performs better than the other elementary sorting algorithms and is actually the best sorting algorithm for arrays that are small or almost sorted.

Counting Sort

Note: this example implementation is specifically for sorting a vector of characters, but we could use the same idea to sort a container containing only small integers or the like.

```
void countingSort(std::vector<char>& input) {
    if (input.size() <= 1) return;
    unsigned char minVal = static_cast<unsigned char>(input[0]);
    unsigned char maxVal = static_cast<unsigned char>(input[0]);
    for (char c : input) {
        unsigned char uc = static_cast<unsigned char>(c);
        if (uc < minVal)
            minVal = uc;
        if (uc > maxVal)
            maxVal = uc;
    }
    size_t range = static_cast<size_t>(maxVal - minVal + 1);
    std::vector<size_t> buckets(range, 0);
    std::vector<char> output(input.size());
    for (char c : input) {
        unsigned char charBucketIndex = static_cast<unsigned char>(c) - minVal;
        ++buckets[charBucketIndex];
    }
    for (size_t i = 1; i < range; ++i)
        buckets[i] += buckets[i - 1];
    for (size_t i = input.size(); i-- > 0; ) { // to prevent underflow
        char c = input[i];
        unsigned char charBucketIndex = static_cast<unsigned char>(c) - minVal;
        output[buckets[charBucketIndex] - 1] = c;
        --buckets[charBucketIndex];
    }
    std::swap(input, output);
}
```

- Time complexity: $O(n + k)$ where k is the number of buckets
- Space complexity: $O(n + k)$
- Stable: yes

Intuition behind counting sort: counting sort is a distribution sorting algorithm (rather than a comparison sorting algorithm like the other sorting algorithms) because it counts how many times each unique value appears rather than repeatedly comparing elements, and it uses these counts to determine where elements belong in the sorted output. In this version for sorting a

vector of characters, we first make a vector of “buckets”, one bucket for each character within the range of characters that appears in the input vector (one bucket for each letter of the alphabet would also work, though we would have to consider letter case). Next, we make the buckets store counts of how many times each character appears in the input vector. Then, we update the buckets to store one after the last index at which each character will appear in the output vector. Finally, we build the output vector by traversing the input vector from right to left (to preserve stability); for each character, we get its correct position in the output vector from its bucket, place it in the output vector, and decrement its bucket so that it stores the correct position for the next instance of that character. This is the fastest sorting algorithm when it can be used, but it can only be used when the set of possible values is small. Otherwise, too much memory would be required.

(Basic) Quicksort

```
size_t partition(Item a[], size_t left, size_t right) {
    size_t pivot = --right;
    while (true) {
        while (a[left] < a[pivot])
            ++left;
        while (left < right && a[right - 1] >= a[pivot])
            --right;
        if (left >= right) break;
        std::swap(a[left], a[right - 1]);
    }
    std::swap(a[left], a[pivot]);
    return left;
}

void quicksort(Item a[], size_t left, size_ t right) {
    if (left + 1 >= right) return;
    size_t pivot = partition(a, left, right);
    quicksort(a, left, pivot);
    quicksort(a, pivot + 1, right);
}
```

- Time complexity: $O(n * log(n))$
- Space complexity: In-place but $O(n)$ stack frames
- Stable: no

Intuition behind quicksort: this is a divide-and-conquer algorithm that works by partitioning (that is, splitting into two parts) the array around a chosen element called the pivot. The idea is that after partitioning, every element to the left of the pivot is less than the pivot, and every element

to the right of the pivot greater than or equal to the pivot (meaning that the pivot is in its correct ultimate position). We can achieve these two properties in $O(n)$ time by repeatedly using two pointers to swap an out-of-place element to the left of the pivot with an out-of-place element to the right of the pivot until all elements are in the correct position relative to the pivot. The algorithm then recursively applies this same logic to the left and right subarrays, which get smaller and smaller until each subarray has one or zero elements and is therefore sorted. The efficiency of quicksort relies heavily on picking a good pivot that will split the array into roughly equal-sized halves (so the median element or one near it is an ideal pivot; if we repeatedly pick the minimum or maximum element, we achieve the worst-case $O(n^2)$ time complexity), but since the array isn't sorted, we can't reliably pick the exact median every time. This basic implementation of quicksort picks the last element as the pivot every time. Quicksort is generally the fastest comparison-based sorting algorithm in practice for large, unsorted arrays.

Improved Quicksort

```
size_t partition(Item a[], size_t left, size_t right) {
    size_t mid = left + (right - left) / 2;
    size_t last = right - 1;
    if (a[mid] < a[left])
        std::swap(a[left], a[mid]);
    if (a[last] < a[left])
        std::swap(a[left], a[last]);
    if (a[last] < a[mid])
        std::swap(a[mid], a[last]);
    --right;
    std::swap(a[mid], a[right]);
    size_t pivot = right;
    while (true) {
        while (a[left] < a[pivot])
            ++left;
        while (left < right && a[right - 1] >= a[pivot])
            --right;
        if (left >= right) break;
        std::swap(a[left], a[right - 1]);
    }
    std::swap(a[left], a[pivot]);
    return left;
}
```

```
void quicksort(Item a[], size_t left, size_t right) {
    if (right - left < 16) {
        insertionSort(a, left, right);
        return;
    }
    size_t pivot = partition(a, left, right);
    if (pivot - left < right - pivot) {
        quicksort(a, left, pivot);
        quicksort(a, pivot + 1, right);
    }
    else {
        quicksort(a, pivot + 1, right);
        quicksort(a, left, pivot);
    }
}
```

- Time complexity: $O(n * log(n))$
- Space complexity: In-place but $O(n * log(n))$ stack frames
- Stable: no

Intuition behind improved quicksort: This version improves upon the basic quicksort by addressing some of its main weaknesses- poor pivot selection, inefficiency on small subarrays, and excessive recursion depth. Instead of always choosing a fixed pivot (like the last element), it selects the median of the first, middle, and last elements as the pivot. This helps avoid the worst-case performance that occurs when the pivot is consistently near the smallest or largest element. Additionally, when subarrays become very small (in this case, fewer than 16 elements), it switches to insertion sort, which performs better on small or nearly sorted arrays. Finally, it always quicksorts the smaller partition first (leaving the larger partition for the tail-recursive call) to further reduce the maximum recursion depth and stack usage.

Heapsort

```
void heapsort(Item a[], size_t n) {
    heapify(a, n);
    for (size_t i = n; i > 1; --i) {
        std::swap(a[1], a[i]);
        fixDown(a, i - 1, 1);
    }
}
```

- Time complexity: $O(n * log(n))$
- Space complexity: $O(1)$
- Stable: no

Intuition behind heapsort: heaps can be used not only to implement priority queues, but also to sort data. Heapsort works in two main stages. First, it performs an $O(n)$ heapify pass to make the array a valid max-heap, where each parent (rooted at `i`) is larger than its children (this implementation uses our custom-defined `heapify` function rather than `std::make_heap` because our heap's root is at index 1; see the Binary Trees section to understand why). Then, it repeatedly relocates the highest-priority element from the heap (which contains only the unsorted elements of the array) to the beginning of the sorted part of the array (which is at the back of the array), making the sorted part of the array one element larger, and restores the heap-ordering property of the unsorted part of the array with `fixDown` in $O(log(n))$ time per relocation. After n such relocations, the array is sorted in ascending order, making its worst-case time complexity $O(n * log(n))$. Though in practice it tends to be slightly slower than quicksort due to cache behavior, it's a good alternative when the recursion depth is too deep because it sorts in-place (i.e. only uses $O(1)$ space).

Merge Sort

```
void merge(Item a[], size_t left, size_t mid, size_t right) {
    size_t n = right - left;
    std::vector<Item> c(n);
    for (size_t i = left, j = mid, k = 0; k < n; ++k)
        if (i == mid)
            c[k] = a[j++];
        else if (j == right)
            c[k] = a[i++];
        else
            c[k] = (a[i] <= a[j]) ? a[i++] : a[j++];
    std::copy(begin(c), end(c), &a[left]);
}
```

```
void topDownMergeSort(Item a[], size_t left, size_t right) {
    if (right < left + 2) return;
    size_t mid = left + (right - left) / 2;
    topDownMergeSort(a, left, mid);
    topDownMergeSort(a, mid, right);
    merge(a, left, mid, right);
}

void bottomUpMergeSort(Item a[], size_t left, size_t right) {
    for (size_t size = 1; size <= right - left; size *= 2)
        for (size_t i = left; i <= right - size; i += 2 * size)
            merge(a, i, i + size, std::min(i + 2 * size, right));
}
```

- Time complexity: $O(n * log(n))$
- Space complexity: $O(n)$
- Stable: yes (when using a stable merge function, like this code and `std::stable_sort` do)

Intuition behind merge sort: Whereas quicksort is a divide-and-conquer approach, merge sort takes a more combine-and-conquer approach. The basic idea is that we can think of a one-element array (the base case) as a sorted container. Furthermore, we can merge two sorted arrays into a single larger sorted array in linear time (using a two-pointer approach). Merge sort repeatedly divides the array into halves until each subarray has just one element, and then repeatedly merges adjacent sorted subarrays back together to form larger sorted subarrays until the entire array is merged into one fully sorted array. There are two main ways to implement this- the top-down (recursive) approach, which recursively merge sorts the left and right halves of the array and then merges the halves as the recursion unwinds, and the bottom-up (iterative) approach, which starts with subarrays of size 1 and repeatedly merges adjacent subarrays of increasing size until the whole array is sorted. The top-down implementation is simpler to write and understand, but the bottom-up implementation avoids recursion and can be slightly faster and more memory-efficient. Merge sort is slower than quicksort in practice, but it's the best sorting algorithm when you need a stable sort.

Bonus- `std::sort` and custom comparators

In real projects, if you need to sort, you should use `std::sort` or `std::stable_sort` because they have very optimized (and correct!) implementations. However, implementing counting sort can be a better idea if the data you want to sort fits the criteria for when counting sort can be used.

Here's some pseudocode showing a rough implementation of `std::sort` so you can get an idea of how it works. This function is named introsort because it's shorthand for introspective sort, which is a hybrid sorting algorithm that takes advantage of the best parts of multiple other sorting algorithms by determining which of them would be the best for the current situation.

```
Algorithm introsort(a[], n):
      if (n is small)
            insertionSort()
      else if (quicksort.recursionDepth is large)
            heapsort()
      else
            quicksort()
```

When using `std::sort`, if you want to customize/specify how a container is sorted (for example, maybe you want to sort by an element's id attribute or in descending order), you can pass a custom comparator as an optional third argument that determines the ordering of elements by specifying how to compare elements. The comparator should return true if the first input should appear before (to the left of) the second input in the sorted order, and false otherwise. Here are three examples of how you can use custom comparators to sort in descending order.

```cpp
bool descendingFunctionComparator(int a, int b) {
    return a > b;
}

struct DescendingFunctorComparator {
    bool operator()(int a, int b) const {
        return a > b;
    }
};

std::sort(begin(v), end(v), descendingFunctionComparator);
// descendingLambdaComparator
std::sort(begin(v), end(v), [](int a, int b) {
    return a > b;
});
std::sort(begin(v), end(v), DescendingFunctorComparator());
```

Hashing

A dictionary is an ADT that stores key-value pairs (called items) and supports the fast insertion of key-value pairs, removal of key-value pairs, and retrieval of the value associated with a key in average $O(1)$ time. To implement the data structure for this ADT, we need a way to quickly convert (or *hash*) an arbitrary key into a unique index indicating where the key-value pair is stored in the table (called a *hash table*). More formally, we need:

- Translation- converts a search key into an integer
- Compression- limits an integer to a valid index
    - using modular arithmetic is common here
- Collision Resolution- resolves search keys that hash to the same table index
    - having some collisions is inevitable, but ideally we want to have as few collisions as possible so we don't need to frequently resolve collisions, because resolving collisions can be expensive

To achieve translation and compression, we can define a hash function $h(key)$ that informally works as follows:

$$h(key) \Rightarrow c(t(key)) \Rightarrow \ table\ index$$

Where:

- $t(key) \Rightarrow \ hashed\ integer$ is a translation function that converts the key into an integer
- $c(hashed\ integer) \Rightarrow table\ index$ is a compression function that maps the translated integer into the range [0, M)
    - M is the size of the hash table. Ideally M is prime and is around the number of elements expected

The benefits of a hash table depend on having a "good" hash function that:

- Must be easy to compute
    - Will compute a hash for every key
    - Will compute the same hash for the same key
- Should distribute keys evenly in the hash table
    - Will minimize collisions

Here's how we can declare hash tables in C++.

```
std::unordered_map<std::string, size_t> monthToDays;
monthToDays["January"] = 31;
auto it = monthToDays.find("January");
if (it != end(monthtodays))
    std::cout << it->first << "- " << it->second << \n"; // January- 31
```

We could also retrieve how many days January has using `monthToDays["January"]`, but the square brackets operator would create an item for "January" if it didn't already exist. Therefore, it's recommended to use `.find()` or `.at()` instead since they don't modify the hash table.

Collision Resolution

We know we have a collision when, after hashing a key, there is an item at the resulting index in the hash table, and the item's key doesn't match the key we hashed (this is why we must store both the key and value in the hash table, rather than just the value). There are several ways to resolve collisions.

Separate Chaining

In separate chaining, the hash table stores linked lists of items (we could use different containers such as vectors or trees, but we focus on linked lists here), where each linked list contains items whose keys hash to the same index in the hash table.

In a table with N keys and M linked lists, we have load factor $\alpha = \frac{N}{M}$ which is the average number of items in each linked list, and it can be greater than 1. With a good hash function:

- Insert time complexity: $O(1)$ if duplicate keys are allowed, $O(\alpha)$ otherwise
- Search time complexity: $O(\alpha)$
- Remove time complexity: $O(\alpha)$

We want to choose $N \approx M$, because then $O(\alpha) \approx O(1)$.

Linear Probing

Linear probing is one of three different open addressing approaches to collision resolution that we'll discuss here, and one of two that uses probing. Open addressing is resolving collisions by using empty locations in the hash table. Probing is searching for the location of an item in the hash table by its key. A Probe can result in:

- Empty- no data found at the probed location
- Hit- there is data at the probed location, and it contains an item whose key matches the search key
- Full- there is data at the probed location, but the key of the item there doesn't match the search key
- Deleted- there used to be data at the probed location, but not anymore. A search considers this empty, and an insertion considers this full.

If a probe results in full, then we probe the table at the "next higher location" until the probe results in hit or empty. Which result we want depends on what operation we're performing. For example, if we're inserting an item, we'll want to probe until we get empty, and if we're updating an item, we'll want to probe until we get hit.

There are different ways we can probe to determine what the next higher location is. Linear probing hashes the search key to get the initial table index of the location to probe. If there is a collision, we probe the next index in the hash table (or wrap around if at the end of the hash table) until the search key or an open address is found. More precisely, probe location:

$(t(key) + j) \ mod \ M$ for $j = 0, 1, 2,...$

until successful. This is simple to understand, but is prone to clustering, which is when lots of contiguous locations in the hash table have items, and can lead to bad performance when trying to find an empty location in the hash table. Knowing the exact time complexity of a search isn't too important here, but just know that as the load factor α increases, so does the search time, especially so for unsuccessful searches.

Quadratic Probing

Quadratic probing is very similar to linear probing. The only difference is how we determine what the next higher location is- if there is a collision, we probe locations in the hash table that are increasingly far away (at a quadratic rate) from the initial probe until the search key or an open address is found. To be more precise, probe location:

$$(t(key) + c_1 * j + c_2 * j^2) \bmod M \text{ for } j = 0, 1, 2,...$$

until successful. The intuition behind choosing this method of probing over linear probing is that it should help prevent the formation of clusters. Picking good constants c_1, c_2, and M is important to get good performance out of this probing method.

Double Hashing

Double hashing is another open addressing approach to collision resolution. If there is a collision, hash the search key again through a different hashing function, and use that to determine the next location to probe. To be more precise, probe location:

$$(t(key) + j * t'(key)) \bmod M \text{ for } j = 0, 1, 2,... \text{ where } t'(key) = q - (t(key) \bmod q)$$

until successful. Like quadratic probing, this should lead to less clustering than linear probing. Pick a constant q that is prime and less than M.

Dynamic Hashing

As the number of keys in a hash table increases, search performance degrades. Dynamic hashing helps address this issue by increasing the size of the hash table when it reaches some pre-determined load factor.

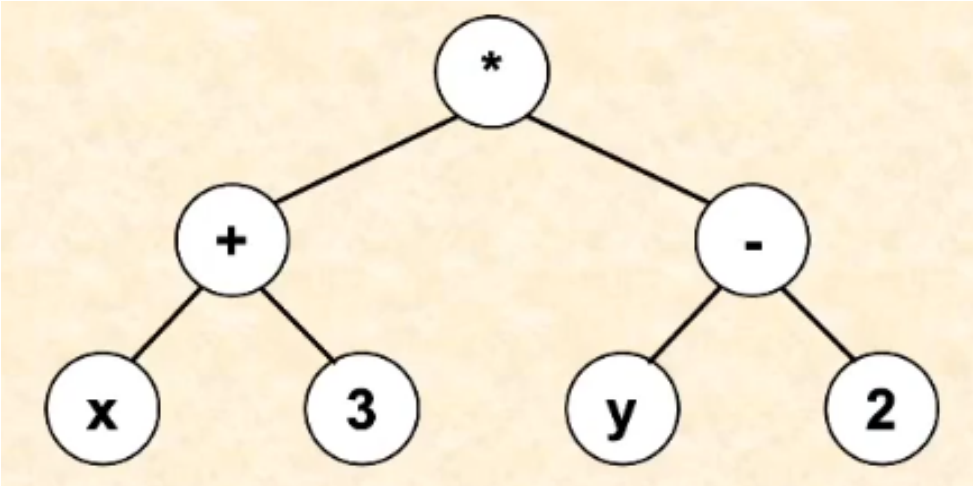
Simple dynamic hashing doubles the size of the hash table when it "fills up"- you could define that as when α exceeds 0.5. This is expensive to do because each item that was already in the hash table needs to be re-hashed into the new hash table, but it's infrequent, so the amortized time complexity for insertion is still $O(1)$.

Binary Trees

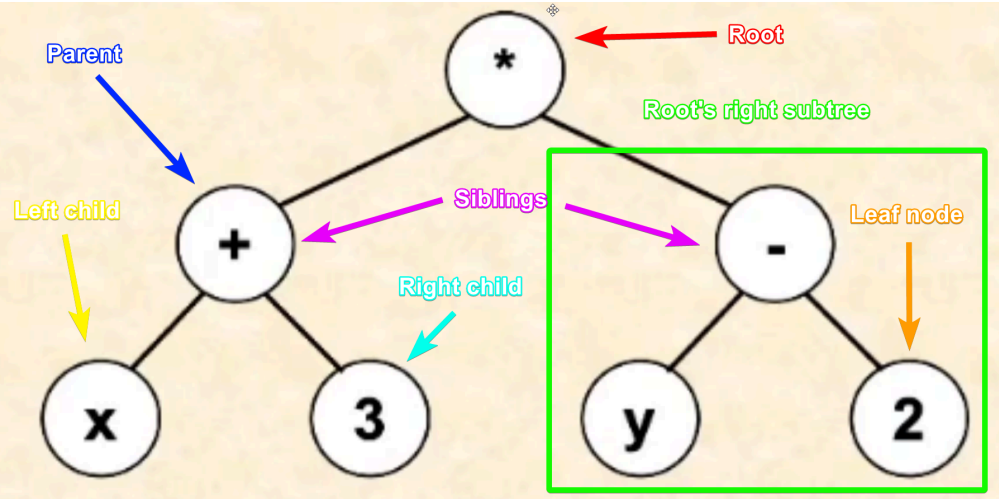
A graph is a data structure consisting of a collection of nodes which hold data, and edges that connect nodes (see the Graphs section for more detail). A tree is a specific kind of graph that has no cycles, which equivalently means there is exactly one unique shortest path between any two nodes in the graph. Here's some common tree terminology.

- Root- the "topmost" node in the tree. A tree may be referred to by its root.
- Parent- immediate predecessor of a node.
- Child- immediate successor of a node.
- Siblings- children of the same parent.
- Ancestor- parent of a parent (closer to the root).
- Descendant- child of a child (further from the root).
- Internal node- a node with children.
- External node- a node without children. Also called a leaf node.
- Left/right subtree- the tree rooted at the left/right child of a binary tree. Often used interchangeably with the left/right child.
- $size(node) = sum(size\ of\ child\ for\ child\ in\ children) + 1$, where $size(empty) = 0$. Equivalently, the number of nodes in the tree rooted at $node$.
- $height(node) = max(height\ of\ child\ for\ child\ in\ children) + 1$, where $height(node) = 0$ if $node$ has no children (i.e. it's empty or a leaf). Equivalently, the maximum number of edges from $node$ to a leaf.
- $depth(node) = depth(parent) + 1$, where $depth(root) = 0$. Equivalently, the number of edges from the root to $node$.
- $level(node) = depth(node) + 1$. A tree has the same number of levels as its height, and a particular level consists of all nodes at the same depth in the tree.

You can think of a tree as having a single root node which may have children nodes, which themselves may be the parent to their own children if they have any. An ordered tree is a tree where there is a linear ordering for the children of each node, so basically the order of children is fixed and matters. A binary tree is an ordered tree in which every node has at most two children, a left child and a right child. Here's an example of a binary tree.



This tree has a size of 7 and a height of 3. Here's the same tree but with some of its node relationships labelled.



A complete binary tree is a binary tree that is completely filled except for maybe the last level, though any nodes that are in the last level must be as leftmost as possible. The example above is a complete binary tree, and would still be a complete binary tree if 3, y, and 2 were removed, or if only y and 2 were removed, or if only 2 were removed.

A binary tree can be represented as an array where:

- the root is at index 1
- For a node at index `i`:
    - Its parent is at `i / 2` (excluding the root, which does not have a parent)

    - Its left child is at `2 * i`, if it has a left child
    - Its right child is at `2 * i + 1`, if it has a right child

For example, the tree above can be stored as `[null, *, +, -, x, 3, y, 2]`. You could instead put the root at index 0, but we put it at index 1 because it makes calculating parent/child indices easier. Typically we only use arrays to represent binary trees that are complete, because a binary tree that isn't complete would have empty gaps in an array representation and it would use a lot more memory than necessary. Another binary tree representation is pointer-based. Unlike the array representation, parents are not easily accessible to children.

```
template <typename KEY> // writing "class" instead of "typename" is equivalent
struct Node {
    KEY key;
    Node* left = nullptr;
    Node* right = nullptr;
    Node(const KEY& k) : key{k} {}
};
```

Depth First Search (DFS)

Depth first search (DFS) is a method for traversing a tree that explores as far down a branch as possible before backtracking. It's called depth-first because it prioritizes exploring deeper nodes before moving sideways to other branches. There are three main types of DFS for binary trees:

- Preorder- visit node, then recursively visit left subtree, then recursively visit right subtree.
    - In the example above, `[*, +, x, 3, -, y, 2]`.
- Inorder- recursively visit left subtree, then visit node, then recursively visit right subtree.
    - In the example above, `[x, +, 3, *, y, -, 2]`.
- Postorder- recursively visit left subtree, then recursively visit right subtree, then visit node.
    - In the example above, `[x, 3, +, y, 2, -, *]`.

Preorder DFS can be implemented using recursion. Recursion often goes naturally with trees because trees themselves are recursive structures, consisting of a root node and smaller subtrees that are equally valid trees (you can think of arrays similarly).

```
void visit(Node* p) {
    std::cout << p.key << " ";
}

void preorder(Node* root) {
    if (root == nullptr) return;
    visit(root->key);
    preorder(root->left);
    preorder(root->right);
}
```

Preorder DFS can also be implemented iteratively using a stack.

```
void dfs(Node* root) {
    if (root == nullptr) return;
    std::stack<Node*> s;
    s.push(root);
    while (!s.empty()) {
        Node* node = s.top();
        s.pop();
        visit(node);
        if (node->right != nullptr)
            s.push(node->right);
        if (node->left != nullptr)
            s.push(node->left);
    }
}
```

Breadth First Search (BFS)

Breadth first search (BFS) is a method for traversing a tree that explores all nodes at the current depth before moving on to the next level. It's called breadth-first because it searches level by level, visiting all neighbors of a node before going deeper.

- Level order- visit nodes in order of increasing depth in the tree.
    - In the example above, `[*, +, -, x, 3, y, 2]`.

It's difficult to implement a level order BFS using recursion, but it can be implemented iteratively using a queue. Note that in both DFS and BFS, we want the left child to be popped from the container before the right child- otherwise, the traversal would be in reverse order.

```
void bfs(Node* root) {
    if (root == nullptr) return;
    std::queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* node = q.front();
        q.pop();
        visit(node);
        if (node->left != nullptr)
            q.push(node->left);
        if (node->right != nullptr)
            q.push(node->right);
    }
}
```

Binary Search Trees (BSTs)

A binary search trees (BST) is a binary tree with the property that the key of every node in the tree is greater than the keys of all the nodes in its left subtree, and is less than or equal to the keys of all the nodes in its right subtree (note: code snippets may not allow duplicates for simplicity). Therefore an inorder traversal of a BST should traverse the tree in sorted order. Because BSTs order data in a structured way, we can search for a node more efficiently than needing to potentially traverse the entire tree.

```
Node* BST_search(Node* x, KEY k) {
    while (x != nullptr && x->key != k)
        if (x->key > k)
            x = x->left;
        else
            x = x->right;
    return x;
}
```

Inserting a node in a BST is similarly easy. We need to perform a search to insert it in the correct place so that we maintain the BST property. This can also be implemented iteratively.

```
void BST_insert(Node*& root, KEY k) {
    if (root == nullptr)
        root = new Node(k);
    else if (root->key > k)
        BST_insert(root->left, k);
    else if (root->key < k)
        BST_insert(root->right, k);
    // else k == root->key and so k already exists in the BST; do nothing
}
```

In a BST search, the maximum number of nodes we may have to traverse is the height of the BST, because with each traversal we move one level down. If a tree of size n is a stick, meaning its height is also n, then a search is $O(n)$, which is no better than a complete traversal of a non-BST binary tree. You can get a stick tree by repeatedly inserting the smallest or largest remaining element into the tree. To really get the efficiency benefits of using a BST, we want to minimize the height of the BST, which is the case when the tree is balanced and so its height is $O(log(n))$ (read more about what a balanced tree is in the AVL Trees section. For now, you can think of it as being close to a complete tree and the opposite of a stick tree).

Removing a node in a BST in a way that maintains the BST property can be more difficult than insertion, and is done differently depending on whether the node being removed has 0, 1, or 2 children. After searching for the node to remove, if the node has no children, it's easy; just replace the parent's pointer to the node with nullptr. If the node has 1 child, replace the node with its child. If the node has 2 children, replace the node's key with the key of the smallest node

in its right subtree, which is called the node's inorder successor (because it would be the node traversed right after during an inorder traversal), and then delete the inorder successor node from the right subtree. Think about why this maintains the BST property.

```
void BST_remove(Node*& root, const KEY& val) {
    if (root == nullptr) return;
    else if (root->value > val)
        BST_remove(root->left, val);
    else if (root->value < val)
        BST_remove(root->right, val);
    else {  //root->value == val
        Node* nodeToDelete = root;
        if (root->left == nullptr) {
            root = root->right;
            delete nodeToDelete;
        }
        else if (root->right == nullptr) {
            root = root->left;
            delete nodeToDelete;
        }
        else {
            Node* inorderSuccessor = root->right;
            while (inorderSuccessor->left != nullptr)
                inorderSuccessor = inorderSuccessor->left;
            nodeToDelete->value = inorderSuccessor->value;
            BST_remove(root->right, inorderSuccessor->value);
        }
    }
}
```

Adelson-Velsky and Landis (AVL) Trees

Earlier it was mentioned that BST operations are more efficient when the BST is balanced. An AVL tree is a self-balancing BST. More specifically, AVL trees have the height balance property, which states that for every internal node v of tree T, the heights of the children of v differ by at most 1. Height imbalances are corrected using rotations to maintain the height balance property (more on this later). When a binary tree maintains this property, it is a valid AVL tree whose worst-case search and insert time complexities are $O(log(n))$ because the height of a balanced tree is $O(log(n))$ (there is a proof for this that is not included here).

Search for AVL trees is the same as it is for BSTs because AVL trees are BSTs and searches do not modify the tree. However, insertions and removals are different because they modify the tree which can cause the tree to become unbalanced, so we need to be able to determine whether the tree is balanced and be able to adjust the tree to make it balanced again if necessary.

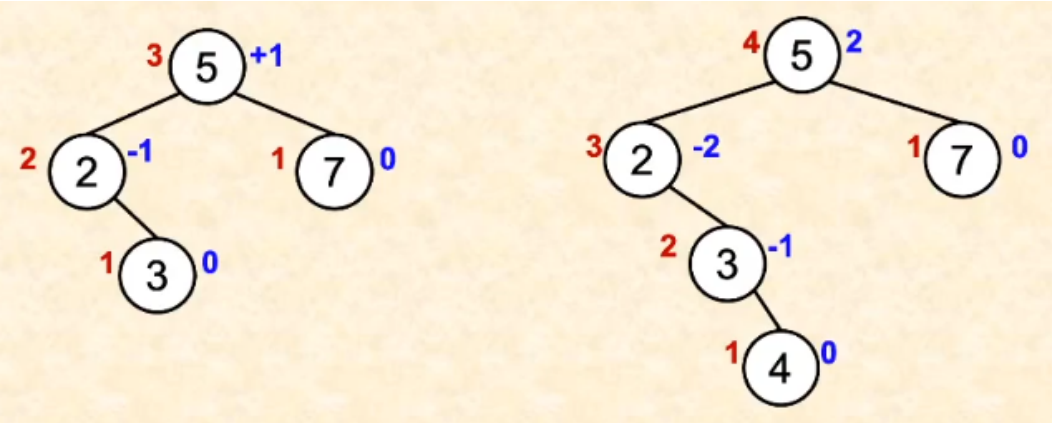
To help us determine whether the tree is balanced or not, we keep track of the height of every node in the tree. We can then define and calculate a balance factor for nodes using the height data.

$$balance(node) = height(left\ subtree) - height(right\ subtree)$$

We can draw conclusions about the balance state of the node based on the result of this computation.

- $balance(node) = 0$ means both subtrees are the same height.
- $balance(node) = 1$ means the left subtree is one node taller than the right subtree.
- $balance(node) = -1$ means the right subtree is one node taller than the left subtree.
- Otherwise, and most importantly, $|balance(node)| > 1$ means that $node$ is unbalanced, which we'll need to fix. A tree is unbalanced if any node in the tree is unbalanced.

The following image shows an example of a balanced tree on the left and an unbalanced tree on the right, with the height of each node in red and the balance factor of each node in blue. The heights shown in this image are all 1 more than their actual heights, but it doesn't really matter since the balance factors are still correct.



Here's code to compute the height and balance factor of a node.

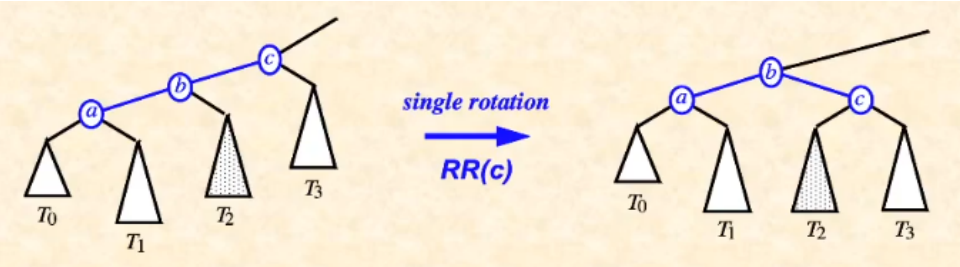
```
int height(Node* n) {
    return n != nullptr ? n->height : 0;
}

int balanceFactor(Node* n) {
    return n != nullptr ? height(n->left) - height(n->right) : 0;
}
```

Rotations

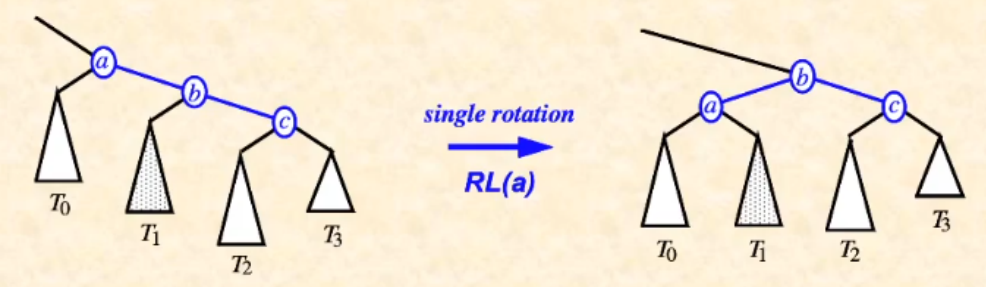
A rotation is a local change involving only three pointers and two nodes (hence constant time complexity) that is used to fix imbalances in a tree while maintaining the BST property.

There are two fundamental single rotations:

- Right rotation $RR(node)$- interchange $node$ with its left child, and then copy the right child of the left child of $node$ ("the middle subtree") to be the new left child of the old parent $node$. Doing a right rotation around $node$ brings $node$ down and to the right, so it's a good idea to perform a right rotation on the unbalanced node at the top of a left stick.

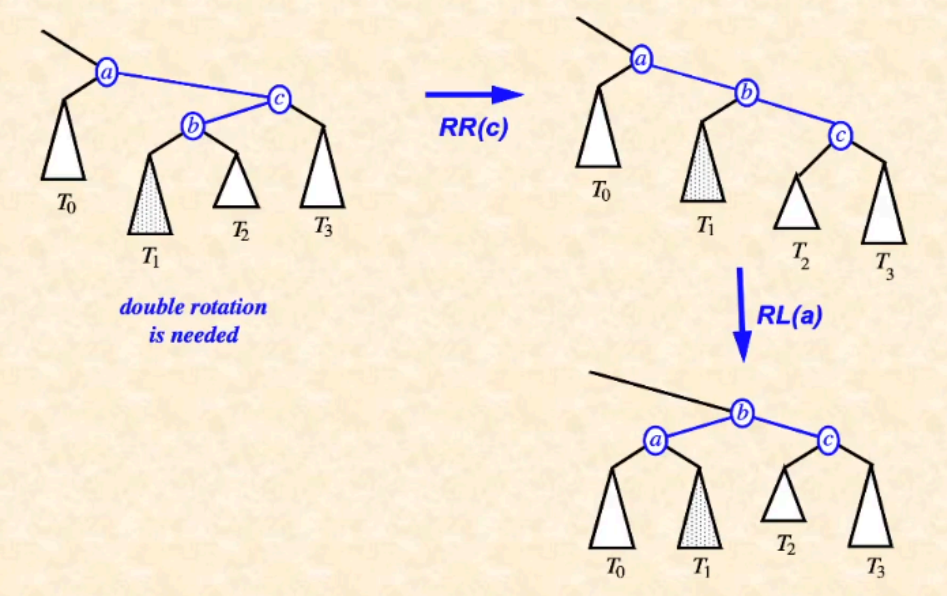


- Left rotation $RL(node)$- interchange $node$ with its right child, and then copy the left child of the right child of $node$ ("the middle subtree") to be the new right child of the old parent $node$. Doing a left rotation around $node$ brings $node$ down and to the left, so it's a good idea to perform a left rotation on the unbalanced node at the top of a right stick.

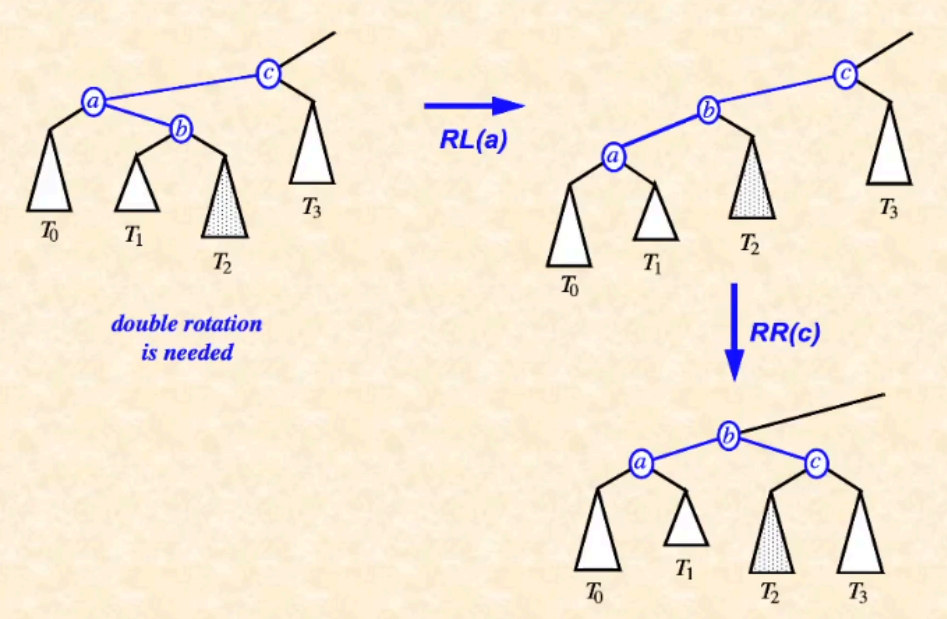


When we have a stick-like structure shown in the two examples above, a single rotation is enough to fix the imbalance. But when we have a zig-zag structure, we need to perform two single rotations, called a double rotation, to balance the tree.

- A right-left zig-zag with an unbalanced node at the top requires a right rotation around the node in the middle of the zig-zag (the right child child of the unbalanced node), and then it's a normal single left rotation on the unbalanced node at the top of the right stick.



- A left-right zig-zag with an unbalanced node at the top requires a left rotation around the node in the middle of the zig-zag (the left child of the unbalanced node), and then it's a normal single right rotation on the unbalanced node at the top of the left stick.



Here's code implementing rotations.

```
void updateHeight(Node* n) {
    if (n != nullptr)
        n->height = std::max(height(n->left), height(n->right)) + 1;
}
```

```
Node* rotateRight(Node* c) {
    Node* b = c->left;
    Node* middleSubtree = b->right;
    b->right = c;
    c->left = middleSubtree;
    updateHeight(c);
    updateHeight(b);
    return b;  // new root
}

Node* rotateLeft(Node* a) {
    Node* b = a->right;
    Node* middleSubtree = b->left;
    b->left = a;
    a->right = middleSubtree;
    updateHeight(a);
    updateHeight(b);
    return b;  // new root
}

Node* checkAndBalance(Node* n) {
    updateHeight(n);
    int bf = balanceFactor(n);
    if (bf > 1) {
        if (balanceFactor(n->left) < 0)
            n->left = rotateLeft(n->left);
        return rotateRight(n);
    }
    if (bf < -1) {
        if (balanceFactor(n->right) > 0)
            n->right = rotateRight(n->right);
        return rotateLeft(n);
    }
    return n;
}
```

AVL Tree Operations

Now that we can re-balance unbalanced trees, we can implement AVL Tree Operations.

When inserting a node into an AVL tree, we first insert the node the same way as we would in a normal BST (see the BSTs section to review and compare it with AVL tree insert), but now after each recursive call the algorithm calls `checkAndBalance` on the current node, ensuring that balance is restored (and node heights are updated) as the recursion unwinds from the inserted leaf back to the root. Even though we check and balance the newly inserted node and all its ancestors, at most only 1 fix (constituting a single or double rotation) will be necessary to re-balance the tree.

```
void AVL_insert(Node*& root, const KEY& k) {
    if (root == nullptr)
        root = new Node(k);
    else if (root->key > k)
        AVL_insert(root->left, k);
    else if (root->key < k)
        AVL_insert(root->right, k);
    root = checkAndBalance(root);
}
```

When removing a node from an AVL tree, we first remove the node the same way as we would in a normal BST (see the BSTs section to review and compare it with AVL tree remove), but now the after each recursive call the algorithm calls `checkAndBalance` on the current node, ensuring that balance is restored (and node heights are updated) as the recursion unwinds from the parent of the removed node back to the root. The removal can cause an imbalance, and when we re-balance a node after a removal we may inadvertently unbalance ancestor nodes, so in the worst-case we must perform $O(log(n))$ fixes, but this doesn't affect the $O(log(n))$ time complexity of remove.

```
void AVL_remove(Node*& root, const KEY& val) {
    if (root == nullptr) return;
    if (root->key > val)
        AVL_remove(root->left, val);
    else if (root->key < val)
        AVL_remove(root->right, val);
    else {  // root->key == val
        Node* nodeToDelete = root;
        if (root->left == nullptr) {
            root = root->right;
            delete nodeToDelete;
        }
        else if (root->right == nullptr) {
            root = root->left;
            delete nodeToDelete;
        }
        else {
            Node* inorderSuccessor = root->right;
            while (inorderSuccessor->left != nullptr)
                inorderSuccessor = inorderSuccessor->left;
            root->key = inorderSuccessor->key;
            AVL_remove(root->right, inorderSuccessor->key);
        }
    }
    if (root != nullptr)
        root = checkAndBalance(root);
}
```

Graphs

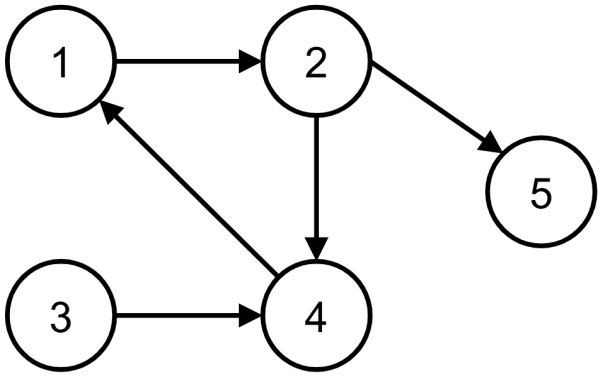
Formally, a graph $G = (V, E)$ is a set of vertices $V = \{v_1, v_2, ..\}$ (also called nodes) together with a set of edges $E = \{e_1, e_2, ..\}$ that connect vertices. Edges are often represented as pairs, and can be either:

- Directed- the edge has direction (i.e. is one-way), so the order of the pair matters. For example, a directed edge $e_m = (v_s, v_t)$ is an edge *from* vertex v_s to vertex v_t, and if drawn would be an arrow to indicate direction. It does not imply the existence of an edge from v_t to v_s. Graphs with directed edges are called directed graphs or digraphs.
- Undirected- the edge has no direction, so the order of the pair doesn't matter. For example, an undirected edge $e_m = (v_s, v_t)$ is an edge *between* vertex v_s and vertex v_t, and if drawn would be a line to indicate no direction. The connection is symmetric, so it could also be written as $e_m = (v_t, v_s)$. e_m could also be written as a set (i.e., $e_m = \{v_t, v_s\}$) rather than a pair to indicate that order doesn't matter. Graphs with undirected edges are called undirected graphs.

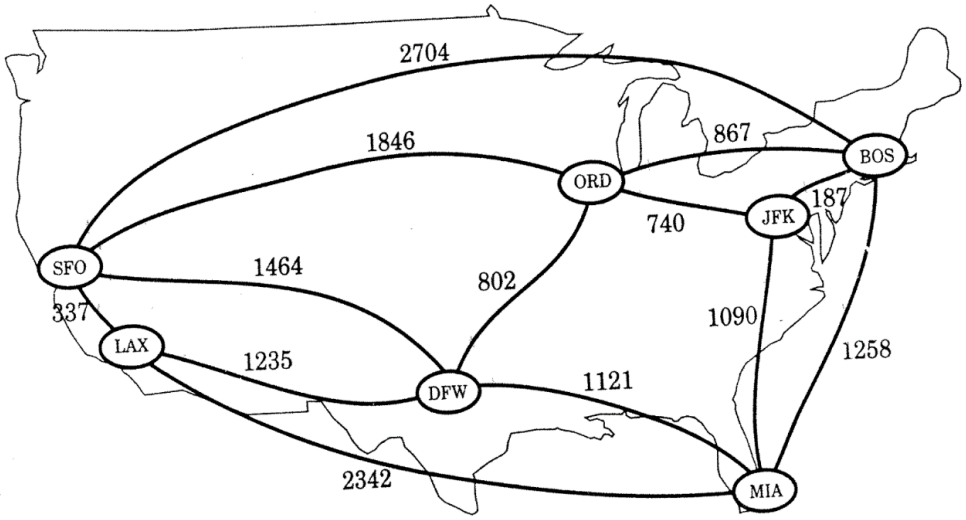
Vertices that are connected by an edge are said to be neighbors or adjacent to each other. Graphs can have edges that start and end at the same vertex, which are called self-loops (i.e., $e_m = (v_s, v_s)$). Graphs can also have multiple edges connecting the same two vertices (i.e., $e_m = (v_s, v_t)$ and $e_n = (v_s, v_t)$), which are called parallel edges. Graphs that do not have self-loops or parallel edges are called simple graphs. Assume that we're talking about simple graphs unless explicitly specified.

A simple path is a sequence of edges leading from one vertex to another vertex with no vertex in the sequence appearing more than once. A connected graph is a graph where a simple path exists between every pair of vertices. A cycle is a path where the first and final vertices are the same. A tree is a connected graph with no cycles.

Here's an example of a directed graph that qualifies as a connected graph and has a cycle $C = \{(1, 2), (2, 4), (4, 1)\}$.



Edges can also be weighted, and when drawn weights can be indicated by labelling the edges with numbers. Think of weight as the distance between vertices, or the cost to traverse the edge. Graphs with weighted edges are called weighted graphs, and graphs with unweighted edges are called unweighted graphs. Graph algorithms often search an unweighted graph for a shortest path, or a weighted graph for a minimum cost path. For example, here's a weighted undirected graph, where vertices are airports, edges are flight routes, and the weight of each edge is the cost to fly that route. It would be useful to be able to determine the cheapest route between any two locations.



Representing Graphs

When representing graphs in code, how many edges it has relative to how many vertices it has matters a lot for choosing how we'll represent it- we don't want to choose a representation that is unnecessarily inefficient or uses an unnecessarily high amount of memory.

- A complete graph is a graph that has an edge between every pair of vertices, which is all possible edges ($|E| = |V| * \frac{|V-1|}{2} \approx |V|^2$, where $|E|$ is the size of the set of edges E).
- A dense graph is a graph that has many edges, which is either a complete graph or an "almost" a complete graph ($|E| \approx |V|^2$). A dense graph can be represented using an adjacency matrix.
- A sparse graph is a graph that has few edges ($|E| << |V|^2$ or $|E| \approx |V|$). Trees are always sparse graphs because for trees $|E| = |V| - 1$. A sparse graph can be represented using an adjacency list.

This course discusses two representations for graphs, one using an adjacency matrix and another using an adjacency list. An adjacency matrix uses $O(|V|^2)$ memory, where a 1 indicates an edge between the two vertices exists, and a 0 indicates it doesn't. Since the flight graph has weighted edges, we could instead write the weight of each edge that exists, or ∞ for edges that don't exist (`std::numeric_limits<double>::infinity()` in code. It will become more apparent later why this is a good choice). In an undirected graph like this one, the matrix is symmetrical across the diagonal.

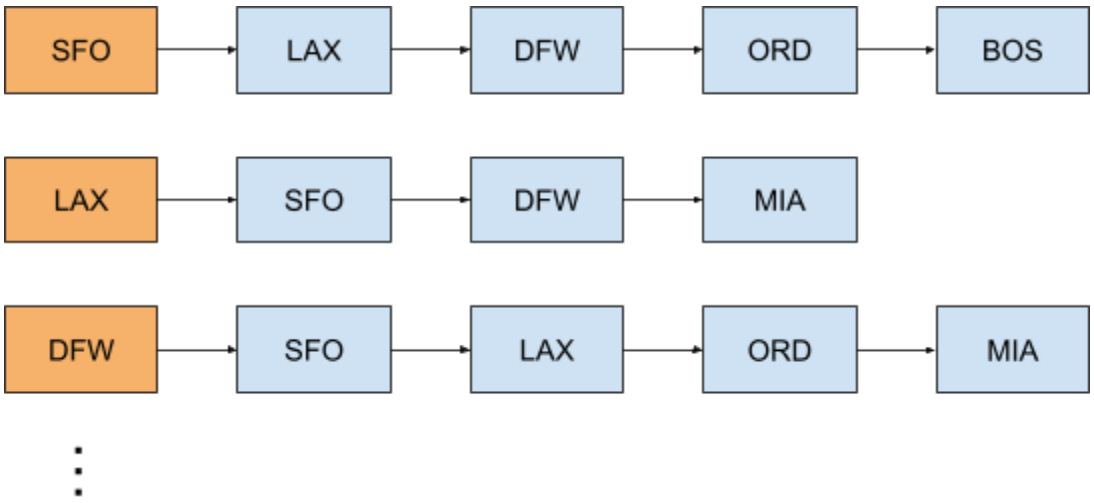
	SFO	LAX	DFW	ORD	MIA	JFK	BOS
SFO	0	1	1	1	0	0	1
LAX	1	0	1	0	1	0	0
DFW	1	1	0	1	1	0	0
ORD	1	0	1	0	0	1	1
MIA	0	1	1	0	0	1	1
JFK	0	0	0	1	1	0	1
BOS	1	0	0	1	1	1	0

In code, it's generally easier to refer to vertices by an integer index instead of a string, so in the flight graph example, SFO would be vertex 0, LAX would be vertex 1, DFW would be vertex 2, and so on. Here's a class-based implementation of an undirected graph using an adjacency matrix representation. It stores the edge weights in `adjMat`, and it conveniently provides member functions to safely read from and write to `adjMat`.

```
class Graph {
public:
    Graph(size_t V)
        : V(V),
          adjMat(V, std::vector<double>(
              V, std::numeric_limits<double>::infinity()
          )) {}
    void addEdge(size_t u, size_t v, double weight) {
        if (!validVertex(u) || !validVertex(v)) return;
        adjMat[u][v] = weight;
        adjMat[v][u] = weight;
    }
    void removeEdge(size_t u, size_t v) {
        if (!validVertex(u) || !validVertex(v)) return;
        adjMat[u][v] = std::numeric_limits<double>::infinity();
        adjMat[v][u] = std::numeric_limits<double>::infinity();
    }
    double getEdgeWeight(size_t u, size_t v) const {
        if (!validVertex(u) || !validVertex(v))
            return std::numeric_limits<double>::infinity();
        return adjMat[u][v];
    }
```

```
    void printEdgeWeights() const {
        std::cout << "    ";
        for (size_t j = 0; j < V; ++j)
            std::cout << j << "    ";
        std::cout << "\n";
        for (size_t i = 0; i < V; ++i) {
            std::cout << i << " ";
            for (size_t j = 0; j < V; ++j) {
                double w = adjMat[i][j];
                if (w == std::numeric_limits<double>::infinity())
                    std::cout << " inf ";
                else
                    std::cout << " " << w << " ";
            }
            std::cout << "\n";
        }
    }
private:
    size_t V;
    std::vector<std::vector<double>> adjMat;
    bool validVertex(size_t u) const {
        return u < V;
    }
};
```

Graphs can also be represented using an adjacency list. An adjacency list only stores edges that exists, so it uses $O(|V| + |E|)$ memory, which can be much less than an adjacency matrix if the graph doesn't have many edges. In the image below, we can start at an orange node (which is accessible in $O(1)$ time) and traverse the linked list starting at that orange node. Each blue node that appears in the linked list indicates that there is an edge connecting the orange node and the blue node. Each linked list has, on average, about $\frac{|E|}{|V|}$ nodes, so iterating through all the neighbors of a vertex is $O(1 + \frac{|E|}{|V|})$ time (whereas it would be $O(|V|)$ using an adjacency matrix). In weighted graphs, we can also include the weight of each edge in the blue nodes.



Here's a class-based implementation of an undirected graph using an adjacency list representation. It stores the edge weights in `adjList`, and it conveniently provides member functions to safely read from and write to `adjList`.

```
class Graph {
public:
    Graph(size_t V)
        : V(V), adjList(V) {}
    void addEdge(size_t u, size_t v, double w) {
        if (!validVertex(u) || !validVertex(v)) return;
        adjList[u].push_back({v, w});
        adjList[v].push_back({u, w});
    }
    void removeEdge(size_t u, size_t v) {
        if (!validVertex(u) || !validVertex(v)) return;
        removeEdgeHelper(u, v);
        removeEdgeHelper(v, u);
    }
    double getEdgeWeight(size_t u, size_t v) const {
        if (!validVertex(u) || !validVertex(v))
            return std::numeric_limits<double>::infinity();
        for (const ListEdge& e : adjList[u])
            if (e.neighbor == v)
                return e.weight;
        return std::numeric_limits<double>::infinity();
    }
```

```
    void printEdgeWeights() const {
        for (size_t u = 0; u < V; ++u) {
            std::cout << u << ": ";
            for (const ListEdge& e : adjList[u])
                std::cout << "(" << e.neighbor << " , " << e.weight << ") ";
            std::cout << "\n";
        }
    }
private:
    struct ListEdge {
        size_t neighbor;
        double weight;
    };
    size_t V;
    std::vector<std::list<ListEdge>> adjList;
    bool validVertex(size_t u) const {
        return u < V;
    }
    void removeEdgeHelper(size_t u, size_t v) {
        if (!validVertex(u)) return;
        auto& edges = adjList[u];
        for (auto it = begin(edges); it != end(edges); )
            if (it->neighbor == v)
                it = edges.erase(it);
            else
                ++it;
    }
};
```

Some algorithms/operations are faster using an adjacency matrix (for example, removing the edge between vertices X and Y), while others are faster using an adjacency list (for example, iterating through all edges that start from vertex X). When using an adjacency list, we can easily add new vertices to the graph after initialization, but that functionality is not implemented here to keep a consistent interface with the adjacency matrix implementation example.

The Shortest Path Problem

For the flight graph above, suppose we were asked to “determine the cheapest path from LAX to BOS”. This is an example of the shortest path problem, which is “given an edge-weighted graph $G = (V, E)$ and two vertices $v_s \in V$ and $v_d \in V$, find the path that starts at v_s and ends at

v_d that has the smallest weighted path length", where the weighted path length of a particular path is the sum of all the weights of the edges in that path. In our case, $v_s = LAX$ and $v_d = BOS$.

A brute-force algorithm would explore all possible paths, which in the worst-case is $O(|V|!)$ time, so that approach is not feasible. A different way we can try to solve this problem is to implement an iterative stack DFS algorithm for general graphs much like we did earlier for binary trees.

```
Algorithm GraphDFS
      Mark source as visited
      Push source to top of (empty) Stack
      While Stack is not empty
            Get/Pop candidate from top of Stack
            For each child/neighbor of candidate
                  If child is unvisited
                        Mark child visited
                        Push child to top of Stack
                        If child is goal
                              Return success
      Return failure
```

This DFS algorithm is pretty similar to the version for binary trees, but there are a couple of key differences- one, we must mark vertices as visited to prevent an infinite loop because the graph can have cycles, and two, a vertex can have more than two children/neighbors. Unfortunately, DFS won't always return the cheapest path to the goal, but it will find a path if one exists. We can also try to do a BFS.

```
Algorithm GraphBFS
      Mark source as visited
      Push source to back of (empty) Queue
      While Queue is not empty
            Get/Pop candidate from front of Queue
            For each child/neighbor of candidate
                  Mark child visited
                  Push child to back of Queue
                  If child is goal
                        Return success
Return failure
```

Unlike DFS, BFS will first search vertices that are closer to the source before searching farther away, so it will find the path to the goal that traverses the fewest vertices, which is the cheapest path in unweighted graphs. Unfortunately, BFS won't always return the cheapest path to the goal in weighted graphs. Both DFS and BFS run in $O(|V|^2)$ time when the graph is represented using an adjacency matrix and $O(|V| + |E|)$ time when the graph is represented using an

adjacency list (the discrepancy is due to how we iterate through all of the children/neighbors of a vertex in each graph representation).

Dijkstra's Algorithm
(You may want to look at Prim's algorithm first; it's similar to Dijkstra's algorithm, but easier.)
DFS and BFS don't solve the shortest path problem, but Dijkstra's algorithm does. Dijkstra's algorithm is a greedy algorithm that solves the shortest path problem by first solving the single-source shortest path problem, which is "given an edge-weighted graph $G = (V, E)$ and a vertex $v_s \in V$, find the shortest path from v_s to every other vertex in V". Dijkstra's algorithm assumes the graph doesn't have negative edge weights.

The general strategy is at each step, pick the unvisited vertex with the shortest known path from the source, visit it so that its shortest known path is finalized, and then update its unvisited neighbors' shortest known path from the source until we visit the destination. To do this, for each vertex $v \in V$, we need to keep track of three pieces of information:

- k_v - do we already know the shortest path from v_s to v? Equivalently, have we already visited v? (initially false for all $v \in V$)
- d_v - what is the length/distance/cost of the shortest path from v_s to v discovered so far? (initially ∞ for all $v \in V$, except for v_s where $d_{v_s} = 0$)
- p_v - what vertex precedes (i.e. is the parent of) v on the shortest path from v_s to v? (initially unknown for all $v \in V$)

We can represent how we track this information as a table that initially looks like this for our flight graph.

v	k_v	d_v	p_v
SFO	F	∞	
BOS	F	∞	
ORD	F	∞	
DFW	F	∞	
LAX	F	0	-
MIA	F	∞	
JFK	F	∞	

We start by writing down the distance $v_s = 0$ (v_s is 0 away from v_s). In this example, v_s = LAX so we set $d_{LAX} = 0$.

Next, we pick the vertex u with k_u = F (i.e. u is unvisited) that has the smallest d_u (i.e. u has the shortest known path from the source v_s) and set k_u = T (i.e. we visit u, at which point we know we have the shortest path to u). Then for each neighbor w of u, if the sum of d_u plus the weight of the edge (u, w) is less than the current d_w, we update d_w with the sum (because we now know a new shorter path from v_s to w, which is the shortest path to u plus the edge (u, w)), and set $p_w = u$. For example, after the first iteration of this algorithm where u = LAX (u will always be v_s for the first iteration) because it initially has the smallest d_u among all unvisited vertices, our table looks like this.

v	k_v	d_v	p_v
SFO	F	337	LAX
BOS	F	∞	
ORD	F	∞	
DFW	F	1235	LAX
LAX	T	0	-
MIA	F	2342	LAX
JFK	F	∞	

For the next iteration we choose u = SFO (u will always be the closest vertex to v_s for the second iteration) because it now has the smallest d_u among all unvisited vertices.

v	k_v	d_v	p_v
SFO	T	337	LAX
BOS	F	3041	SFO
ORD	F	2183	SFO
DFW	F	1235	LAX
LAX	T	0	-
MIA	F	2342	LAX
JFK	F	∞	

We repeat these steps until we visit v_d, which in this example is BOS, at which point we will have found the shortest path from LAX to BOS. We can then re-construct the shortest path backwards by starting at BOS, going to p_{BOS}, then $p_{p_{BOS}}$, and so on until we get back to LAX. For this problem, we find that the cheapest path is LAX, DFW, ORD, BOS, and the cost of that path is $d_{BOS} = 1235 + 802 + 867 = 2904$.

Alternatively, we can terminate the algorithm after visiting every vertex, at which point we will have solved the single-source shortest path problem.

Dijkstra's algorithm can be implemented in $O(|V|^2)$ time where for each iteration, a linear search is performed to find the unvisited vertex u that has the smallest d_u. Here's an implementation for a graph represented using an adjacency list.

```
struct TableEntry {
    bool visited = false;
    double dist = std::numeric_limits<double>::infinity();
    size_t parent = SIZE_MAX; // sentinel value
};

void printPath(const std::vector<TableEntry>& table, size_t dest) {
    std::stack<size_t> path;
    size_t v = dest;
    while (v != SIZE_MAX) {
        path.push(v);
        v = table[v].parent;
    }
    std::cout << "Path: ";
    while (!path.empty()) {
        std::cout << path.top();
        path.pop();
        if (!path.empty())
            std::cout << " -> ";
    }
    std::cout << "\nTotal cost/distance: " << table[dest].dist << "\n";
}
```

```
void findShortestPath(
    const std::vector<std::list<ListEdge>>& adjList,
    size_t source, size_t dest
) {
    size_t V = adjList.size();
    std::vector<TableEntry> table(V);
    table[source].dist = 0;
    while (true) {
        size_t u = SIZE_MAX;
        double minDist = std::numeric_limits<double>::infinity();
        for (size_t v = 0; v < V; ++v)
            if (!table[v].visited && table[v].dist < minDist) {
                minDist = table[v].dist;
                u = v;
            }
        if (u == SIZE_MAX) break;
        table[u].visited = true;
        if (u == dest) {
            printPath(table, dest);
            return;
        }
        for (const ListEdge& edge : adjList[u]) {
            size_t w = edge.neighbor;
            double newDist = table[u].dist + edge.weight;
            if (newDist < table[w].dist) {
                table[w].dist = newDist;
                table[w].parent = u;
            }
        }
    }
    std::cout << dest << " is unreachable from " << source << ".\n";
}
```

But we could also determine the unvisited vertex u that has the smallest d_u by using a priority queue to store vertices, where the highest priority vertex v in the priority queue is the vertex that has the smallest d_v. This is $O(|E|log(|V|))$ time, which is about $O(|V^2|log(|V|))$ for a dense graph (slower than before) and $O(|V|log(|V|))$ for a sparse graph (faster than before).

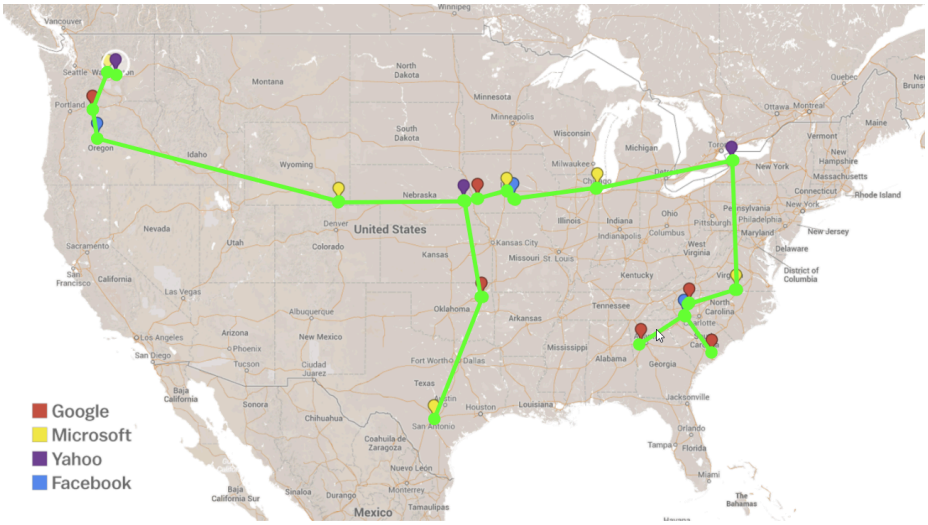
```
using Pair = std::pair<double, size_t>; // (distance, vertex)

void findShortestPath(
    const std::vector<std::list<ListEdge>>& adjList,
    size_t source, size_t dest
) {
    size_t V = adjList.size();
    std::priority_queue<Pair, std::vector<Pair>, std::greater<Pair>> pq;
    std::vector<TableEntry> table(V);
    table[source].dist = 0;
    pq.push({0, source});
    while (!pq.empty()) {
        auto [distU, u] = pq.top();
        pq.pop();
        if (table[u].visited) continue;
        table[u].visited = true;
        if (u == dest) {
            printPath(table, dest);
            return;
        }
        for (const ListEdge& edge : adjList[u]) {
            size_t w = edge.neighbor;
            double newDist = table[u].dist + edge.weight;
            if (newDist < table[w].dist) {
                table[w].dist = newDist;
                table[w].parent = u;
                pq.push({newDist, w});
            }
        }
    }
    std::cout << dest << " is unreachable from " << source << ".\n";
}
```

Note that the priority queue can contain visited vertices because after pushing a (d_v, v) pair to the priority queue for a new known shortest distance d_v to vertex v, we may later find the actual shortest distance d_v' to v and push (d_v', v) to the priority queue. Since $d_v' < d_v$, we'll pop (d_v', v) from the priority queue before (d_v, v) and visit v. Later, we'll pop the stale (d_v, v) that was added to the priority queue before (d_v', v), but at this point v is already visited. Therefore, after

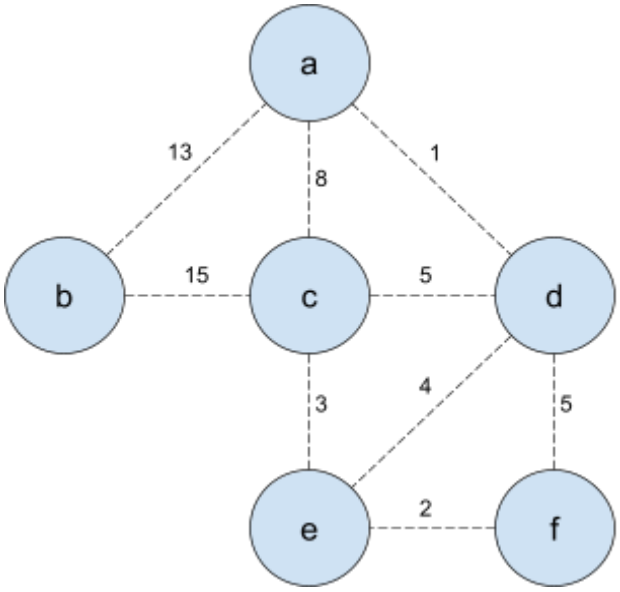
we pop a vertex from the priority queue, we check whether the vertex was already visited and move onto the next iteration if it was.

The Minimum Spanning Tree (MST) Problem

The minimum spanning tree (MST) problem is "given a weighted, undirected graph $G = (V, E)$, find a subgraph $T = (V, E')$ where $E' \subseteq E$ such that all vertices are pair-wise connected (i.e. reachable from every other vertex) and the sum of all edge weights in T is minimal". T is called a minimum spanning tree (MST) of G, and is guaranteed to be a tree. G may have multiple valid MSTs. Solving this problem can be useful, for example, if we wanted to use the least amount of expensive fiber-optic cables possible to make a group of data centers reachable from every other data center. Here's an example of an MST, where the vertices are data centers and the edges are fiber-optic cables connecting them.



We'll look at the following simpler graph to explain algorithms to solve this problem, where vertices are referred to by characters/letters.



Prim's Algorithm

Prim's algorithm is a greedy algorithm that solves the MST problem by greedily selecting edges one-by-one and adding them to a growing subgraph. It starts with two sets of vertices:

- *Innies*- vertices that have already been visited and hence are in T (initially empty)
- *Outies*- vertices that have not yet been visited and hence are not in T (initially V)

The general strategy is at each step, make the outie that is closest to any innie an innie until there are no more outies. To do this, for each vertex $v \in V$, we need to keep track of three pieces of information:

- k_v- is v an innie? Equivalently, have we already visited v? (initially false for all $v \in V$)
- d_v- what is the weight of the smallest/minimum weight edge connecting v to any innie? (initially ∞ for all $v \in V$, except for for the first innie v_r called the root, where $d_{v_r} = 0$)
- p_v- what vertex precedes (i.e. is the parent of) v? (initially unknown for all $v \in V$)

We can represent how we track this information as a table. Here's what it would initially look like for the graph above.

v	k_v	d_v	p_v
a	F	0	-
b	F	∞	
c	F	∞	
d	F	∞	
e	F	∞	
f	F	∞	

We start by picking any vertex to be the root v_r. It doesn't matter which, so we'll just pick $v_r =$ a, and then set $d_0 = 0$.

Next, we pick the vertex u with $k_u =$ F (i.e. u is an outie) that has the smallest d_u (i.e. u is the closest to any innie) and set $k_u =$ T (i.e. we make u an innie). Then for each neighbor w of u, if $k_w =$ F (i.e. w is an outie) and the weight of the edge (u, w) is less than the current d_w, we update d_w to be the weight of (u, w) (because we now know a new smallest/minimum weight edge from w to any innie, which is the edge between w and the new innie u), and set $p_w = u$.

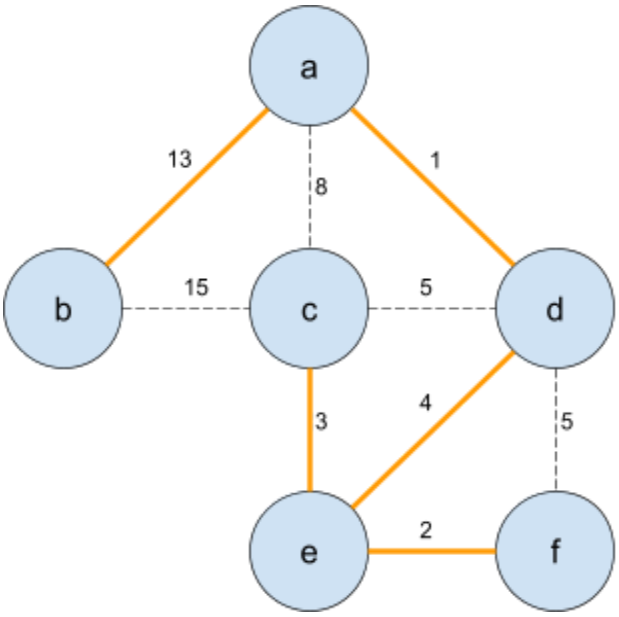
For example, after the first iteration of this algorithm where u = a (u will always be v_r for the first iteration) because it initially has the smallest d_u among all outies, our table looks like this.

v	k_v	d_v	p_v
a	T	0	-
b	F	13	a
c	F	8	a
d	F	1	a
e	F	∞	
f	F	∞	

For the next iteration we choose u = d (u will always be the closest vertex to v_r for the second iteration) because it now has the smallest d_u among all unvisited vertices.

v	k_v	d_v	p_v
a	T	0	-
b	F	13	a
c	F	5	d
d	T	1	a
e	F	4	d
f	F	5	d

We run these steps a total of $|V|$ times until all vertices are innies, at which point we will have found an MST. We can then re-construct the MST by adding every (p_v, v) edge in our table to the MST. For our example graph, we get the following MST with edges (a, d), (d, e), (e, f), (e, c), and (a, b).



Prim's algorithm can be implemented in $O(|V|^2)$ time where for each iteration, a linear search is performed to find the outie u that has the smallest d_u. Here's an implementation for a graph represented using an adjacency list.

```
struct TableEntry {
    bool innie = false;
    double dist = std::numeric_limits<double>::infinity();
    size_t parent = SIZE_MAX;
};

void printMST(const std::vector<TableEntry>& table, size_t root) {
    double totalWeight = 0;
    std::cout << "Edges in MST:\n";
    for (size_t v = 0; v < table.size(); ++v)
        if (v != root && table[v].parent != SIZE_MAX) {
            std::cout << "(" << table[v].parent << " , " << v << ") "
                      << "[weight = " << table[v].dist << "]\n";
            totalWeight += table[v].dist;
        }
    std::cout << "Total weight of MST: " << totalWeight << "\n";
}
```

```cpp
void findMST(const std::vector<std::list<ListEdge>>& adjList, size_t root) {
    size_t V = adjList.size();
    std::vector<TableEntry> table(V);
    table[root].dist = 0;
    for (size_t i = 0; i < V; ++i) {
        size_t u = SIZE_MAX;
        double minDist = std::numeric_limits<double>::infinity();
        for (size_t v = 0; v < V; ++v)
            if (!table[v].innie && table[v].dist < minDist) {
                minDist = table[v].dist;
                u = v;
            }
        table[u].innie = true;
        for (const ListEdge& edge : adjList[u]) {
            size_t w = edge.neighbor;
            double weight = edge.weight;
            if (!table[w].innie && weight < table[w].dist) {
                table[w].dist = weight;
                table[w].parent = u;
            }
        }
    }
    printMST(table, root);
}
```

But we could also determine the outie u that has the smallest d_u by using a priority queue to store vertices, where the highest priority vertex v in the priority queue is the vertex that has the smallest d_v. This runs in $O(|E|log(|V|))$ time, which is about $O(|V|^2 log(|V|))$ for a dense graph (slower than before) and $O(|V|log(|V|))$ for a sparse graph (faster than before).

```
void findMST(const std::vector<std::list<ListEdge>>& adjList, size_t root) {
    std::priority_queue<Pair, std::vector<Pair>, std::greater<Pair>> pq;
    std::vector<TableEntry> table(adjList.size());
    table[root].dist = 0;
    pq.push({0, root});
    while (!pq.empty()) {
        auto [weightU, u] = pq.top();
        pq.pop();
        if (!table[u].innie) {
            table[u].innie = true;
            for (const ListEdge& edge : adjList[u]) {
                size_t w = edge.neighbor;
                double weight = edge.weight;
                if (!table[w].innie && weight < table[w].dist) {
                    table[w].dist = weight;
                    table[w].parent = u;
                    pq.push({weight, w});
                }
            }
        }
    }
    printMST(table, root);
}
```

Note that the priority queue can contain innies because after pushing a (d_v, v) pair to the priority queue for a new smallest edge weight d_v of an edge connecting v to any innie, we may later find the actual smallest edge weight d_v' of an edge connecting v any innie and push (d_v', v) to the priority queue. Since $d_v' < d_v$, we'll pop (d_v', v) from the priority queue before (d_v, v) and make v an innie. Later, we'll remove the stale (d_v, v) that was added to the priority queue before (d_v', v), but at this point v is already an innie. Therefore, after we pop a vertex from the priority queue, we check whether the vertex is already an innie and move onto the next iteration if it is.

Kruskal's Algorithm

Kruskal's algorithm is another algorithm that finds an MST, where the general strategy is to repeatedly add the smallest-weight edge that doesn't create a cycle to the MST until all vertices are in the MST.

To do this, we sort the edges of the graph by weight in ascending order in $O(|E|log(|E|))$ time. Then for each edge $e \in E$ from smallest to largest weight, we check whether adding e to the growing MST would create a cycle by using the union-find data structure, where each vertex starts in its own disjoint set. Union-find lets us check whether two vertices u and w connected by edge e are already in the same set (which would mean adding e to the MST creates a cycle) in $O(\alpha(|V|))$ time (practically constant). If u and w are in different sets, we add e to the MST and union their sets, also in $O(\alpha(|V|))$ time. Overall, this makes Kruskal's algorithm run in $O(|E|log(|E|))$ time, which is $O(|V|^2 log(|V|))$ for dense graphs and $O(|V|log(|V|))$ for sparse graphs, so it should only be run with sparse graph inputs (use the $O(|V|^2)$ implementation of Prim's algorithm for dense graphs).

Here's an implementation of Kruskal's algorithm. We can iterate through all edges in an adjacency list to get a vector of edges, where each edge stores the two vertices that the edge connects and its weight.

```
struct Edge {
    size_t u;
    size_t w;
    double weight;
};

std::vector<Edge> getEdges(const std::vector<std::list<ListEdge>>& adjList) {
    std::vector<Edge> edges;
    size_t n = adjList.size();
    std::vector<std::vector<bool>> seen(n, std::vector<bool>(n, false));
    for (size_t u = 0; u < n; ++u)
        for (const ListEdge& e : adjList[u]) {
            size_t w = e.neighbor;
            if (!seen[u][w] && !seen[w][u]) {
                edges.push_back({u, w, e.weight});
                seen[u][w] = true;
                seen[w][u] = true;
            }
        }
    return edges;
}
```

```
void printMST(const std::vector<Edge>& mst) {
    double totalWeight = 0;
    std::cout << "Edges in MST:\n";
    for (const Edge& e : mst) {
        std::cout << "(" << e.u << " , " << e.w << ") "
                  << "[weight = " << e.weight << "]\n";
        totalWeight += e.weight;
    }
    std::cout << "Total weight of MST: " << totalWeight << "\n";
}

void findMST(const std::vector<std::list<ListEdge>>& adjList) {
    std::vector<Edge> edges = getEdges(adjList);
    std::sort(begin(edges), end(edges), [](const Edge& a, const Edge& b) {
        return a.weight < b.weight;
    });
    UnionFind uf(adjList.size());
    std::vector<Edge> mst;
    for (const Edge& edge : edges)
        if (!uf.connected(edge.u, edge.w)) {
            uf.unite(edge.u, edge.w);
            mst.push_back(edge);
        }
    printMST(mst);
}
```

Greedy Algorithms

A greedy algorithm is an algorithm that makes a sequence of decisions, always choosing the locally optimal decision at each step, and it never reconsiders decisions that have been made. Greedy algorithms may run significantly faster than brute-force algorithms, but for a greedy algorithm to work, it must be shown that locally optimal decisions lead to a globally optimal solution by proving that the following two properties hold:

- Optimal substructure- the optimal solution is the first action plus the optimal solution for the remaining subproblem (note the recursion)
- Greedy choice property- the first action can be chosen greedily without invalidating the optimal solution

Examples of greedy algorithms include Dijkstra's algorithm, Prim's algorithm, and Kruskal's algorithm.

Counting Change

The counting change problem is "given a set $P = \{p_1, p_2, \ldots, p_n\}$ of coins worth $D = \{d_1, d_2, \ldots, d_n\}$ consisting of pennies (1 cent), nickels (5 cents), dimes (10 cents), and quarters (25 cents), return the smallest set of coins whose values sum to a given number A (i.e. the change)."

A brute-force algorithm can determine every one of the 2^n possible subsets of P, and for every subset examines each coin in the subset to determine whether the subset is one that sums to the correct number and is the smallest such subset. This is extremely inefficient, running in $O(n * 2^n)$ time.

A greedy algorithm can select the largest coin p_i whose value d_i is less than the amount owed remaining A. We update A to be $A - d_i$, and repeat until A is 0. In other words, we always pick the best option possible- we pick a quarter if possible, and if not then we pick a dime if possible, and if not then pick a nickel if possible, and if not then pick a penny, which is always possible when $A > 0$. This ends up solving the problem in worst-case $O(n)$ time. However, it is important to note that while this algorithm works for U.S. coins, it doesn't always give the optimal solution for arbitrary coin systems. For example, if the U.S. coin system did not have nickels and we were asked to give change for 30 cents, the algorithm would select a quarter and five pennies, when the optimal solution is three dimes.

Sorting

The sorting problem is "given a random array of numbers $A = [a_1, a_2, \ldots, a_n]$, re-arrange the elements in A such that for all $i < n$, $a_i \leq a_{i+1}$."

A brute-force algorithm can determine every one of the $n!$ possible orderings of A (ignoring duplicates), and for every ordering examine each integer in the ordering to determine whether the ordering is one that is sorted. This algorithm is called bogo sort and is extremely inefficient, running in $O(n!)$ time.

A greedy algorithm can find the smallest item and move it to the first location. Then we can find the next smallest item and move it to the second location. We perform this a total of $n - 1$ times until A is sorted. In other words, we always pick the best option possible, which is the smallest item that we have not yet sorted. This algorithm is called selection sort and runs in worst-case $O(n^2)$ time. Bubble sort and insertion sort are also greedy algorithms.

Divide-and-conquer and Combine-and-conquer Algorithms

A divide-and-conquer algorithm is an often-recursive top-down algorithm that divides a problem solution into two (or more) smaller non-overlapping subproblems, preferably of equal size. Divide-and-conquer algorithms are often efficient and elegant, but recursive calls can be

expensive and sometimes are dependent upon the initial state of subdomains. Examples of divide-and-conquer algorithms include binary search and quicksort.

A combine-and-conquer algorithm is a bottom-up algorithm that starts with the smallest subdomain possible, then combines increasingly larger subdomains until size $= n$. Merge sort is an example of a combine-and-conquer algorithm.

Backtracking Algorithms
(The notes in this section are long because this topic is challenging.)
Constraint satisfaction problems are problems that define a set of constraints and ask for complete solutions that satisfy all of the constraints (or in other words, don't violate any of the constraints). They can be solved using backtracking algorithms, which are algorithms that systematically consider all possible outcomes of each decision, but prune searches that violate a constraint.

When implementing a backtracking algorithm, we typically:

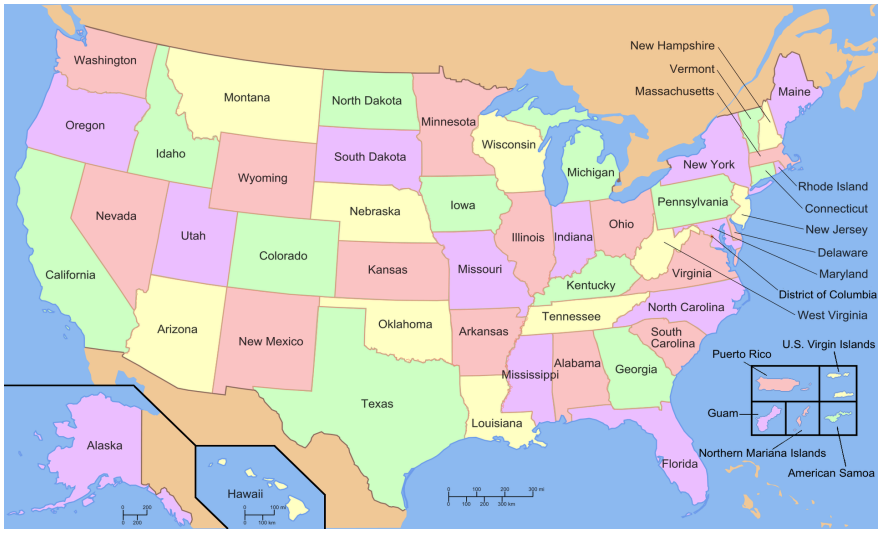
- Check that a candidate (or partial) solution is promising, meaning it doesn't violate a constraint (this is usually the hardest part of implementing a backtracking algorithm)
- If a candidate solution is promising, check whether it's a complete solution, meaning we're done searching (often, this entails checking that a candidate solution is long enough or contains all elements of the input to be a complete solution to the problem; this is basically the "base case")
- If a candidate solution is promising but it's not a complete solution, we're not done, so we try extending the candidate solution to get closer to a complete solution

We prune by not extending candidate solutions that aren't promising, thus reducing the search space. If we reach a point where a promising candidate solution cannot be extended any further, then we know there are no complete solutions that begin with that candidate solution, in which case we undo the most recent step- or *backtrack*- and continue searching. Examples of problems that can be solved using backtracking algorithms include finding a path through a maze and solving a sudoku puzzle.

Backtracking algorithms often have the same time complexity as brute-force algorithms (can be exponential), and are not guaranteed to be the most efficient algorithm for a given problem, but they're still better than brute-force and a good fallback we can rely on to solve constraint satisfaction problems.

The m-coloring Problem

The m-coloring problem is “given a graph with n vertices and m colors, assign each vertex in the graph one of the m colors such that no two adjacent vertices (i.e. vertices that are connected by an edge) are the same color.” Solving this problem can be useful, for example, to color a graph in a way such that no two states or countries are the same color. Here’s an example of a 4-colored map of the United States of America. You can think of the states as vertices, with states that share a border having an edge connecting them.



This problem can be solved using a backtracking algorithm. When examining a candidate solution, which is a coloring of a subset of the graph’s vertices, we consider it promising if no two adjacent vertices are the same color. If it’s promising but not all vertices have been colored, we try to extend the candidate solution by coloring one more vertex to get closer to a complete solution. If all vertices have been colored, then we’ve found a complete solution, in which case we print the solution and return. If we cannot extend a promising candidate solution any further because the current coloring makes it impossible to assign a valid color to the next vertex without violating the coloring constraint, we backtrack by undoing the most recent coloring decision and trying a different color for that vertex.

```
bool promising(
    const std::vector<std::vector<int>>& adjMat,
    const std::vector<size_t>& currentColoring, size_t vertexIndex
) {
    for (size_t j = 0; j < vertexIndex; ++j)
        if (
            adjMat[vertexIndex][j] == 1
            && currentColoring[vertexIndex] == currentColoring[j]
        )
            return false;
    return true;
}
```

```
bool colorVertex(
    const std::vector<std::vector<int>>& adjMat, size_t m,
    std::vector<size_t>& currentColoring, size_t numColored
) {
    size_t n = adjMat.size();
    if (numColored == n) {
        for (size_t c : currentColoring)
            std::cout << c << " ";
        std::cout << "\n";
        return true;
    }
    for (size_t color = 0; color < m; ++color) {
        currentColoring[numColored] = color;
        if (promising(adjMat, currentColoring, numColored))
            if (colorVertex(adjMat, m, currentColoring, numColored + 1))
                return true;
    }
    currentColoring[numColored] = SIZE_MAX;
    return false;
}

void m_coloring(const std::vector<std::vector<int>>& adjMat, size_t m) {
    size_t n = adjMat.size();
    std::vector<size_t> currentColoring(n, SIZE_MAX);
    if (!colorVertex(adjMat, m, currentColoring, 0))
        std::cout << "No valid " << m << "-coloring exists.\n";
}
```

This algorithm runs in $O(m^n)$ time, which is slow but in practice is faster than brute-force because of pruning.

Bonus- Generate Parentheses

The generate parentheses problem is “given a positive integer n, generate all combinations of n pairs of parentheses that are well-formed (i.e. balanced).” A well-formed parentheses string is one where every opening parenthesis `'('` has a corresponding closing parenthesis `')'` that comes after it, and at no point are there more closing parentheses than opening parentheses. For example, when $n = 3$, the valid combinations are:

```
((()))
(()())
(())()
()(())
```

```
()()()
```

This problem can be solved using a backtracking algorithm. A candidate solution promising if:

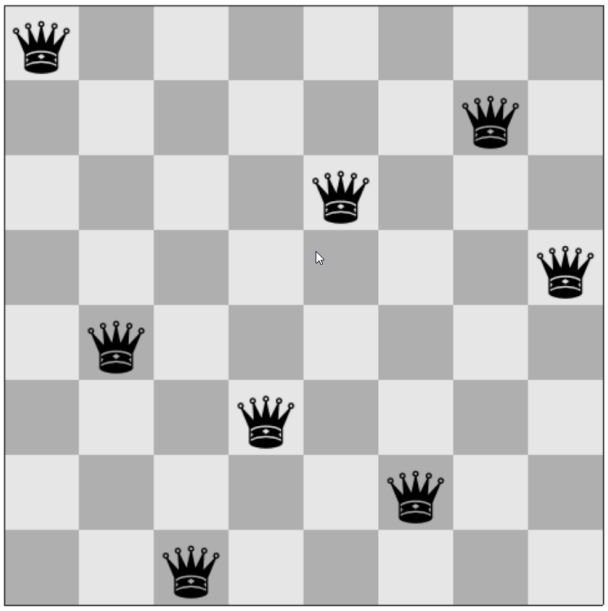
- the number of opening parentheses it has is less than or equal to n (because every complete solution needs exactly n opening parentheses)
- the number of opening parentheses it has is greater than or equal to the number of closing parentheses it has (because every opening parenthesis needs exactly one corresponding closing parenthesis after it)

In the following implementation, we only extend a candidate solution if doing so results in a promising candidate solution. Therefore we can extend candidate solutions that haven't used n opening parentheses yet with an open parenthesis, and we can extend candidate solutions that have more opening parentheses than closing parentheses with a closing parenthesis. If we follow these rules, then any candidate solution with $2 * n$ parentheses is a well-formed combination of n pairs of parentheses, and thus is a complete solution, so we print it. After finding all combinations that begin with a particular sequence of parentheses, we backtrack to find more combinations that begin with a different sequence.

```cpp
void addParenthesis(
    size_t n,
    std::string& currentComb, size_t numOpen, size_t numClose
) {
    if (currentComb.size() == 2 * n) {
        std::cout << currentComb << "\n";
        return;
    }
    if (numOpen < n) {
        currentComb.push_back('(');
        addParenthesis(n, currentComb, numOpen + 1, numClose);
        currentComb.pop_back();
    }
    if (numOpen > numClose) {
        currentComb.push_back(')');
        addParenthesis(n, currentComb, numOpen, numClose + 1);
        currentComb.pop_back();
    }
}

void generateParentheses(size_t n) {
    std::string currentComb;
    addParenthesis(n, currentComb, 0, 0);
}
```

n Queens Problem

The n queens problem is "determine if we can place n queens on an $n \times n$ chessboard in a way that none of the queens threaten each other" (in other words, no queen is in a position where it can be attacked/taken by another queen). A queen in chess can move horizontally, vertically, and diagonally, so this means that there should be no two queens in the same row, column, or left diagonal, or right diagonal. The following image shows how for $n = 8$, we can place 8 queens on an 8×8 board in a way that none of the queens threaten each other.



A brute-force algorithm for $n = 8$ would check the $1.785 * 10^{14}$ possible board configurations, which is extremely slow. Luckily, this problem can be solved using backtracking. We consider a candidate solution promising if for every queen on the board, there is no other queen in the same row, column, left diagonal, or right diagonal. If it's promising but not all n queens have been placed on the board, we try to extend the candidate solution by placing one more queen on the board to get closer to a complete solution. If all the queens have been placed, then we've found a complete solution, in which case we print the solution and return. If we cannot extend a promising candidate solution any further because the queens are placed in a way that makes it impossible to add a new queen to the board without threatening an existing queen, then we backtrack by undoing the most recent queen placement and trying a different position for that queen.

When implementing the promising check for the n-queens problem, the most direct approach is to represent the chessboard as an $n \times n$ 2D array, where each element indicates whether a queen is placed on that tile. Then, to determine if we can place a queen on the tile at (r, c), we can iterate over the entire row r, column c, and both diagonals passing through (r, c) to check for existing queens. This implementation is simple to understand but inefficient- for each placement, it may require scanning up to $O(n)$ tiles per direction. A more efficient approach relies on the observations that in any complete solution to the problem:

- each row will have exactly one queen (because there are n queens that need to be placed in n rows, and two queens cannot be in the same row)
- each column will have exactly one queen (because there are n queens that need to be placed in n columns, and two queens cannot be in the same column)
- each left diagonal will have *at most* one queen (because there are n queens that need to be placed in $2n - 1$ left diagonals, and two queens cannot be in the same left diagonal, so n left diagonals will have one queen and $n - 1$ left diagonals will have none)
- each right diagonal will have *at most* one queen (because there are n queens that need to be placed in $2n - 1$ right diagonals, and two queens cannot be in the same right diagonal, so n right diagonals will have one queen and $n - 1$ right diagonals have none)

Because these conditions must hold for any complete solution, we don't need to model the chessboard as a 2D array. Instead, we can:

- use a 1D array that we index into using a row number, and the corresponding value is the column that a queen is placed in that row (if there is no queen placed in that row, then the corresponding value has no meaning)
- use a 1D array that we index into using a column number, and the corresponding value is a boolean indicating whether there is a queen in that column
- use a 1D array that we index into using a left diagonal number (which is equal to row number plus column number), and the corresponding value is a boolean indicating whether there is a queen in that left diagonal
- use a 1D array that we index into using a right diagonal number (which is equal to row number minus column number plus n minus 1), and the corresponding value is a boolean indicating whether there is a queen in that right diagonal

Now we can determine whether a queen can be placed on a tile at a specific row number and column number in $O(1)$ time. The following class-based implementation places queens row by row, trying each column position that doesn't violate a constraint. If it reaches a dead end, it backtracks and tries a different position until a queen has been placed in every row, in which case we have found a complete solution, so we print it and return.

```
class NQueens {
public:
    NQueens(size_t n) :
        n(n),
        columnInRow(n),
        column(n, false),
        leftDiagonal(2 * n - 1, false),
        rightDiagonal(2 * n - 1, false)
    {
        if (!placeQueen(0))
            std::cout << "No solution for n=" << n << ".\n";
    }
```

```
private:
    size_t n;
    std::vector<size_t> columnInRow;
    std::vector<bool> column;
    std::vector<bool> leftDiagonal;
    std::vector<bool> rightDiagonal;
    bool promising(size_t row, size_t col) {
        size_t leftDiagonalIndex = row + col;
        size_t rightDiagonalIndex = row - col + (n - 1);
        return (
            !column[col]
            && !leftDiagonal[leftDiagonalIndex]
            && !rightDiagonal[rightDiagonalIndex]
        );
    }
    bool placeQueen(size_t row) {
        if (row == n) {
            printSolution();
            return true;
        }
        for (size_t col = 0; col < n; ++col)
            if (promising(row, col)) {
                size_t leftDiagonalIndex = row + col;
                size_t rightDiagonalIndex = row - col + (n - 1);
                columnInRow[row] = col;
                column[col] = true;
                leftDiagonal[leftDiagonalIndex] = true;
                rightDiagonal[rightDiagonalIndex] = true;
                if (placeQueen(row + 1))
                    return true;
                column[col] = false;
                leftDiagonal[leftDiagonalIndex] = false;
                rightDiagonal[rightDiagonalIndex] = false;
            }
        return false;
    }
```

```
    void printSolution() {
        std::cout << "Solution found for n=" << n << ". Place queens at:\n";
        for (size_t row = 0; row < n; ++row)
            std::cout << "(" << row << " , " << columnInRow[row] << ")\n";
    }
};
```

This algorithm runs in $O(n!)$ time, which is slow but still faster than brute-force. Note that the n queens problem has a solution for all positive integers n except for $n = 2$ and $n = 3$.

Generate Permutations

The generate permutations problem is “given an array of distinct items, generate all possible permutations of those items.” A permutation is an ordering of the items. For example, if the input array is `[1, 2, 3]`, then all of the possible permutations of that array are:

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 1, 2]
[3, 2, 1]
```

This problem can be solved using a backtracking algorithm. We start by swapping an item with the item in the first position of the array, denoted as *fixing* that first position because we have determined what item goes into the first position, whereas we still need to determine where items go in the remaining positions. We’ll repeat fixing the first position with every item in turn since we must consider every possible permutation. Then, at each step, we recursively permute the remaining items by fixing the next unfixed position, and then fixing the next unfixed position after that, and so on. If all of the positions have been fixed, then we’ve found a complete permutation, in which case we print the permutation. After finding all permutations that begin with a particular sequence of items, we backtrack to find more permutations that begin with a different sequence.

```
void genPermsHelper(std::vector<int>& nums, size_t permLength) {
    if (permLength == nums.size()) {
        for (int n : nums)
            std::cout << n << " ";
        std::cout << "\n";
        return;
    }
    for (size_t i = permLength; i < nums.size(); ++i) {
        std::swap(nums[permLength], nums[i]);
        genPermsHelper(nums, permLength + 1);
        std::swap(nums[permLength], nums[i]);
    }
}

void genPerms(std::vector<int> nums) {
    genPermsHelper(nums, 0);
}
```

In a permutation of n distinct items, any one of the n items can occupy the first position in the permutation, any of the remaining $n - 1$ items can occupy the second position, any of the

remaining $n - 2$ items can occupy the third position, and so on. As a result, there are $n * (n - 1) * (n - 2) *... = n!$ possible permutations, so this algorithm runs in $O(n!)$ time. This algorithm can be difficult to visualize, so try running the code with print statements to see how it works.

Bonus- Generate Subsets

The generate subsets problem is "given an array of distinct items, generate all possible subsets of those items." A subset is a selection of items in the array, and because a subset is a set, the ordering of the items in a subset doesn't matter. For example, if the input array is `[1, 2, 3]`, then all of the possible subsets of that array are:

```
{1, 2, 3}
{1, 2}
{1, 3}
{1}
{2, 3}
{2}
{3}
{}
```

This problem can be solved using a backtracking algorithm. At each step, we consider whether to include (i.e. select) or exclude the current item. We then recursively consider the remaining items by considering the next item, and then the next item after that, and so on. If all of the items have been considered for inclusion or exclusion, then we've found a complete subset, in which case we print the subset. After finding all subsets that, for the previous considerations, include the current item and all subsets that, for the previous considerations, exclude the current item, we backtrack to find more subsets with different previous considerations.

```
void genSubsetsHelper(
    const std::vector<int>& nums,
    std::vector<int>& currentSelection, size_t i
) {
    if (i == nums.size()) {
        for (int n : currentSelection)
            std::cout << n << " ";
        std::cout << "\n";
        return;
    }
    currentSelection.push_back(nums[i]);
    genSubsetsHelper(nums, currentSelection, i + 1);
    currentSelection.pop_back();
    genSubsetsHelper(nums, currentSelection, i + 1);
}
```

```
void genSubsets(const std::vector<int>& nums) {
    std::vector<int> currentSelection;
    genSubsetsHelper(nums, currentSelection, 0);
}
```

When making a subset from a list of n items, we must make one of two choices (include or exclude) for each item. As a result, there are $2 * 2 * 2 *\ldots = 2^n$ possible subsets, so this algorithm runs in $O(2^n)$ time. This algorithm can be difficult to visualize, so try running the code with print statements to see how it works.

Branch-and-bound (B&B) Algorithms

Optimization problems are constraint satisfaction problems that aim to find the *best* solution (as opposed to just any solution) that satisfies all of the constraints. The best solution is defined as any complete solution that:

- doesn't violate a constraint (i.e. is a valid solution)
- minimizes an objective function that takes a solution as input and outputs the cost of that solution (this is how we quantify how "good" a solution is, with smaller-cost solutions being better)

Optimization problems can be solved using branch-and-bound algorithms, which are algorithms that systematically explore all possible outcomes of each decision, but prune candidate solutions that:

- violate a constraint (just like backtracking algorithms)
- cannot be better than the best solution found so far, because the smallest possible cost of any complete solution that starts with the candidate solution- the *lower bound* of that candidate solution- is greater than or equal to the cost of the best solution found so far- the *upper bound*

Branch-and-bound algorithms can be viewed as an extension of backtracking algorithms from solving constraint satisfaction to solving optimization problems. When the algorithm starts, no solution has been found yet, so we initialize the upper bound to be infinity. This ensures that the cost of the first complete solution found will automatically become the new upper bound.

When implementing a branch-and-bound algorithm, we typically:

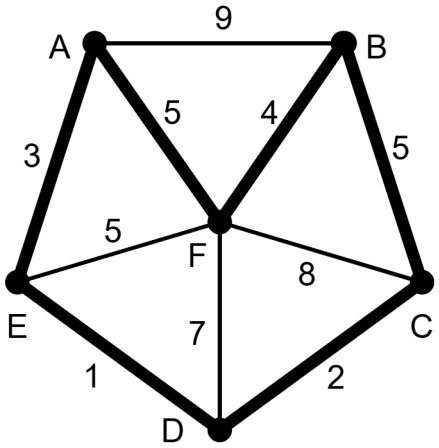
- For a candidate solution, calculate a lower bound estimating the smallest cost of a complete solution that starts with the candidate solution. The lower bound must be an underestimate of (i.e. strictly less than or equal to) the actual smallest cost (so that we don't accidentally prune away any potential better solutions, but a good lower bound is ideally close to the actual smallest cost so that we can prune more branches while searching), and it should be easier to calculate than the exact smallest cost (or else we're not saving any work)
- Check that a candidate solution is promising, meaning that:
    - it doesn't violate a constraint
    - its lower bound is less than the upper bound
- If a candidate solution is promising, check whether it's a complete solution. If it is, update the best solution found so far and the upper bound
- If a candidate solution is promising but it's not a complete solution, try extending the candidate solution so that we get closer to a complete solution

As before, we don't extend unpromising candidate solutions, and we backtrack if we cannot extend a promising candidate solution any further. Once the algorithm runs to completion, the best solution found will be the globally minimal solution. Note that we must consider all complete solutions, because a complete solution that we haven't considered could be better than the best solution we've found so far. Also note that branch-and-bound can similarly be used to find the

globally maximal solution. Examples of problems that can be solved using branch-and-bound algorithms include the traveling salesman problem and the knapsack problem.

Like backtracking algorithms, branch-and-bound algorithms often have the same time complexity as brute-force algorithms (can be exponential), and are not guaranteed to be the most efficient algorithm for a given problem, but they're still better than brute-force and a good fallback we can rely on to solve optimization problems.

Traveling Salesman Problem (TSP)
The traveling salesman problem is "given a graph, find the smallest-weight Hamiltonian cycle in that graph". A Hamiltonian cycle is a cycle that traverses every vertex in the graph exactly once, except for the first/last vertex which are the same. Solving this problem can be useful, for example, for a traveling salesman to find the shortest-distance tour through a set of cities that starts and ends at the same city and visits every city exactly once. Here's an example of the smallest-weight Hamiltonian cycle in a graph with six vertices.



A brute-force algorithm for TSP would generate every possible permutation of the vertices, compute the total cost of the cycle for each, and return the smallest one. Since there are $(n-1)!$ possible Hamiltonian cycles- which we call *tours*- this approach is extremely slow for large n. However, the TSP can be solved more efficiently using a branch-and-bound algorithm.

To start the algorithm, it works much like a brute-force algorithm- we must consider every possible tour, so we generate permutations of the vertices, where the first vertex in every permutation is the first vertex in the adjacency matrix (because the starting location of the tour is fixed). It's just like the `genPerms` function from earlier, but now we prune branches that aren't promising.

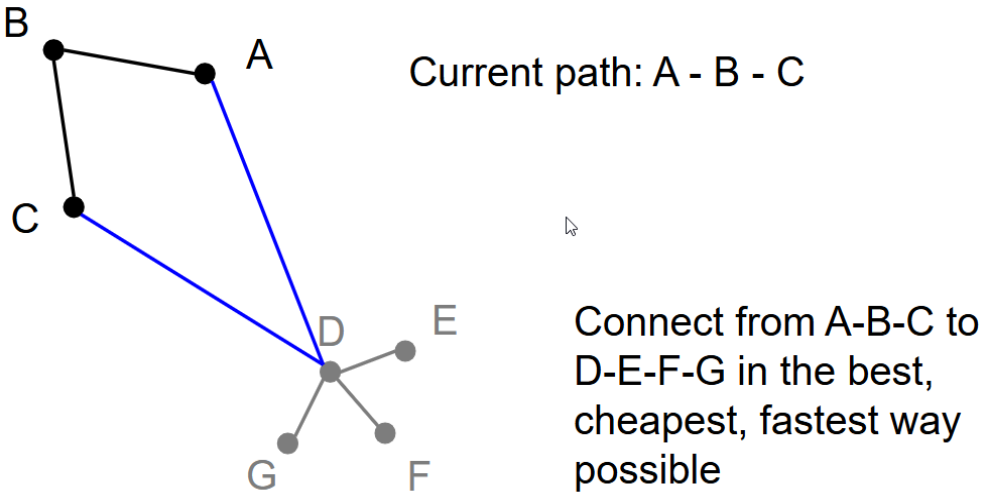
We generate the permutations in a way that we only examine valid tours that don't violate any constraints, so the only reason that a candidate solution- which we call a *partial path*- could be considered unpromising is if the cost of the partial path is already too high compared to the cost of the best complete tour found so far. More specifically, we prune a partial path if we know that

any complete tour starting with that partial path would have a higher cost than the cost of the best complete tour found so far.

To implement the promising check then, we need an efficient way to calculate a good underestimate of the minimum cost of a complete tour starting with a given partial path- the lower bound of that given partial path. We can do this by creating an MST of the unvisited vertices. The key idea is that the cost of any complete tour extending from the current partial path must be at least the sum of:

- the cost of the already chosen edges (the current partial path)
- the cost of the MST connecting all of the unvisited vertices
- the weight of the smallest edge connecting the last visited vertex to any unvisited vertex
- the weight of the smallest edge connecting any unvisited vertex back to the starting vertex

This becomes easier to visualize and understand with an example.



Note that this MST estimation does not satisfy all of the constraints of the problem, but that's fine- it's only meant to be an estimation, not a solution.

Importantly, making an MST of the unvisited vertices and connecting it to the current partial path can be done in $O(n^2)$ time using Prim's algorithm, which is much faster than the $O(n!)$ time it would take to find the actual minimum-cost complete tour starting with the partial path.

Below is an implementation of TSP using these ideas. It takes an adjacency matrix as input and generates permutations of the vertices, where the fixed vertices correspond to the current partial path and the remaining vertices are unvisited. For each partial path, it uses Prim's algorithm to construct an MST of the unvisited vertices and uses that MST to estimate a lower bound on the cost of any complete tour starting with that partial path. This lower bound is then compared with the cost of the best complete tour found so far- the upper bound- to determine whether the current partial path is unpromising and should be pruned. When the algorithm finishes, it will have found the globally optimal complete tour.

```cpp
double pathCost(
    const std::vector<std::vector<double>>& adjMat,
    const std::vector<size_t>& path, size_t pathLength
) {
    if (pathLength == 0) return 0;
    double cost = 0;
    for (size_t i = 0; i < pathLength - 1; ++i)
        cost += adjMat[path[i]][path[i + 1]];
    return cost;
}

double MSTCost(
    const std::vector<std::vector<double>>& adjMat,
    const std::vector<size_t>& unvisited
) {
    size_t n = unvisited.size();
    if (n == 0) return 0;
    std::vector<double> dist(n, std::numeric_limits<double>::infinity());
    std::vector<bool> innie(n, false);
    dist[0] = 0;
    double cost = 0;
    for (size_t i = 0; i < n; ++i) {
        size_t u = n;  // n is a sentinel here
        double minDist = std::numeric_limits<double>::infinity();
        for (size_t v = 0; v < n; ++v)
            if (!innie[v] && dist[v] < minDist) {
                minDist = dist[v];
                u = v;
            }
        innie[u] = true;
        cost += dist[u];
        for (size_t w = 0; w < n; ++w) {
            double weight = adjMat[unvisited[u]][unvisited[w]];
            if (!innie[w] && weight < dist[w])
                dist[w] = weight;
        }
    }
    return cost;
}
```

```cpp
bool promising(
    const std::vector<std::vector<double>>& adjMat,
    double upperBound,
    const std::vector<size_t>& currentPath, size_t currentPathLength
) {
    double partialPathCost = pathCost(adjMat, currentPath, currentPathLength);
    std::vector<size_t> unvisited(
        begin(currentPath) + currentPathLength,
        end(currentPath)
    );
    double unvisitedMSTCost = MSTCost(adjMat, unvisited);
    size_t lastVertexIndex = currentPathLength > 0
        ? currentPath[currentPathLength - 1]
        : currentPath[0];
    double minLastVisitedToUnvisited = std::numeric_limits<double>::infinity();
    double minUnvisitedToStart = std::numeric_limits<double>::infinity();
    for (size_t v : unvisited) {
        minLastVisitedToUnvisited = std::min(
            minLastVisitedToUnvisited,
            adjMat[lastVertexIndex][v]
        );
        minUnvisitedToStart = std::min(
            minUnvisitedToStart,
            adjMat[v][currentPath[0]]
        );
    }
    double lowerBound = partialPathCost +
                        unvisitedMSTCost +
                        minLastVisitedToUnvisited +
                        minUnvisitedToStart;
    return lowerBound < upperBound;
}
```

```cpp
void TSPHelper(
    const std::vector<std::vector<double>>& adjMat,
    std::vector<size_t>& bestTour, double& upperBound,
    std::vector<size_t>& currentPath, size_t currentPathLength
) {
    size_t n = adjMat.size();
    if (currentPathLength == n) {
        std::vector<size_t> tour = currentPath;
        tour.push_back(currentPath[0]);
        double completeTourCost = pathCost(adjMat, tour, tour.size());
        if (completeTourCost < upperBound) {
            bestTour = tour;
            upperBound = completeTourCost;
        }
        return;
    }
    for (size_t i = currentPathLength; i < n; ++i) {
        std::swap(currentPath[currentPathLength], currentPath[i]);
        if (promising(adjMat, upperBound, currentPath, currentPathLength + 1))
            TSPHelper(
                adjMat,
                bestTour, upperBound,
                currentPath, currentPathLength + 1
            );
        std::swap(currentPath[currentPathLength], currentPath[i]);
    }
}

std::pair<std::vector<size_t>, double> TSP(
    const std::vector<std::vector<double>>& adjMat
) {
    size_t n = adjMat.size();
    std::vector<size_t> bestTour;
    double upperBound = std::numeric_limits<double>::infinity();
    std::vector<size_t> currentPath(n);
    std::iota(begin(currentPath), end(currentPath), 0);  // google std::iota
    TSPHelper(adjMat, bestTour, upperBound, currentPath, 0);
    return {bestTour, upperBound};
}
```

This algorithm has the same $O(n!)$ time complexity as the brute-force approach, but is much faster than brute-force in practice because of pruning. If you're short on time and a good-enough solution will suffice, then there are also more efficient algorithms called *heuristic algorithms* that can find a complete tour whose cost is less than or equal to two times the optimal cost in $O(n^2)$ time, or find a complete tour whose cost is less than or equal to 1.5 times the optimal cost in $O(n^2 * log(n))$ time.

Dynamic Programming (DP) Algorithms

A dynamic programming algorithm is an algorithm that remembers the solutions of subproblems when subproblems are not independent, meaning their calculations overlap/are related (if subproblems are independent, use a divide-and-conquer or combine-and-conquer algorithm instead). It solves smaller subproblems first, stores their solutions, and looks them up later when needed, so it trades off extra memory for increased speed. The process of memorizing past solutions to look up later for future use is called *memoization*, and the container storing the solutions is called the *memo*. Dynamic programming can allow for amazing speedups (on the order of exponential $O(2^n)$ or even factorial $O(n!)$ to polynomial $O(n^c)$) but can be conceptually difficult to grasp. There are both bottom-up and top-down approaches.

To implement memoization for a top-down approach, first write a recursive function that solves the problem (albeit inefficiently). Then modify the function so that right before it returns, the current inputs and the corresponding solution are written to the memo. Also modify the function so that right after entering, it checks the memo to see if for the current inputs, the corresponding solution is already stored. If it is, the solution can be retrieved and returned instead of spending time re-computing it. Often, we can implicitly store the inputs by using them to index into the memo at which location the corresponding solution is stored if it was previously computed.

Calculating Fibonacci Numbers

The Fibonacci sequence is a sequence of numbers where the first Fibonacci number in the sequence $F_0 = 0$, the second Fibonacci number $F_1 = 1$, and for $n > 1$, the n'th Fibonacci number is the sum of the previous two Fibonacci numbers, $F_n = F_{n-1} + F_{n-2}$. The sequence starts with `[0, 1, 1, 2, 3, 5, 8, 13, ...]`. F_n could be computed as follows:

```
size_t fibonacci(size_t n) {
    if (n <= 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

But since each call to `fibonacci` spawns two more calls, this runs in $O(2^n)$ time (because the subproblems overlap, the tightest bound is actually ~$\theta(1.618^n)$). For example, to calculate F_{25}:

$$F_{25} = F_{24} + F_{23}$$

$$= (F_{23} + F_{22}) + (F_{22} + F_{21})$$
$$= (F_{22} + F_{21}) + (F_{21} + F_{20}) + (F_{21} + F_{20}) + (F_{20} + F_{19})$$
$$= \ldots$$

The key observation here is that there's a lot of repeated work. For example, the subproblems F_{24} and F_{23} aren't independent because both depend on the solution to the subproblem F_{22} in their calculations. It would be better to calculate each Fibonacci number only once, store its value, and then the next time we need that Fibonacci number in a calculation, we can simply retrieve the stored value- this is memoization. Here's a top-down dynamic programming implementation where we first check whether the Fibonacci number we want to calculate is already stored (for this specific problem, `SIZE_MAX` works as a sentinel value indicating that we have not yet calculated a Fibonacci number because there is no Fibonacci number is equal to `SIZE_MAX`). If it is, we retrieve and return it; otherwise, we recursively compute it, store it in our memo, and then return it.

```
size_t fibonacciHelper(size_t n, std::vector<size_t>& memo) {
    if (n <= 1)
        return n;
    if (memo[n] != SIZE_MAX)
        return memo[n];
    memo[n] = fibonacciHelper(n - 1, memo) + fibonacciHelper(n - 2, memo);
    return memo[n];
}

size_t fibonacci(size_t n) {
    std::vector<size_t> memo(n + 1, SIZE_MAX);
    return fibonacciHelper(n, memo);
}
```

And here's a bottom-up dynamic programming implementation where we first compute the smallest Fibonacci numbers and use those to iteratively compute larger ones.

```
size_t fibonacci(size_t n) {
    if (n <= 1)
        return n;
    std::vector<size_t> memo(n + 1);
    memo[0] = 0;
    memo[1] = 1;
    for (size_t i = 2; i <= n; ++i)
        memo[i] = memo[i - 1] + memo[i - 2];
    return memo[n];
}
```

These dynamic programming implementations improve the time complexity of the simple recursive implementation from $O(2^n)$ all the way down to $O(n)$.

Calculating Binomial Coefficients

A binomial is a mathematical expression with two terms connected by an addition or subtraction operator, such as $x + y$. If we raise the binomial to the power of n, i.e. $(x + y)^n$, the resulting expansion will have $n + 1$ terms (if either x or y are constant, there may be like terms that can be combined to simplify the expression). For particular integers $k \geq 0$ and $n \geq k$, the binomial coefficient is the positive integer constant that the k'th term is multiplied by after expanding the binomial raised to the power of n (assume no simplification). Equivalently, the binomial coefficient is also the number of ways to choose a subset of k items (or in other words, a combination of size k) from a larger set of n distinct items. Mathematically, this is written and calculated as (read as "n choose k"):

$$^nC_k = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Computing n choose k in code looks easy at first. The most direct approach is to compute $n!$, $k!$, and $(n - k)!$, and then simply multiply and divide from there. However, factorials become very large very quickly, which can be problematic- 13! overflows 32-bit integers, 21! overflows 64-bit integers, and 35! overflows 128-bit integers. This approach would leave us unable to compute relatively small values that can be stored in memory like 52 choose 5 (the number of possible poker hands, which ends up being only around 2.5 million) because the intermediate computations overflow, so we need to find a different approach.

Luckily, there is a recursive definition for n choose k:

- Base cases ($n \geq 0$)
  - $\binom{n}{0} = 1$ (1 way to choose 0 items from a set of n items)
  - $\binom{n}{n} = 1$ (1 way to choose n items from a set of n items)
- Recursive case ($n > k \geq 1$)
  - $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$

The recursive definition translated into code is elegant.

```
uint64_t binomialCoefficient(uint32_t n, uint32_t k) {
    if (k == 0 || k == n)
        return 1;
    return binomialCoefficient(n - 1, k - 1) + binomialCoefficient(n - 1, k);
}
```

Unfortunately, it is also very inefficient, having an exponential time complexity. But if we expand the calculation into the sum of its parts like we did with Fibonacci numbers, we'll see that the recursive cases overlap, so we can use dynamic programming to speed up execution. Because calculating a binomial coefficient takes two inputs, we can create a 2D memo that is indexed into using those two inputs to store the results of intermediate binomial coefficient computations (for this specific problem, 0 works as a sentinel value indicating that we have not yet calculated n choose k because for valid values of n and k, n choose k is never 0). Here's a top-down implementation.

```
uint64_t binomHelper(
    uint32_t n, uint32_t k,
    std::vector<std::vector<uint64_t>>& memo
) {
    if (k == 0 || k == n)
        return 1;
    if (memo[n][k] > 0)
        return memo[n][k];
    memo[n][k] = binomHelper(n - 1, k - 1, memo) + binomHelper(n - 1, k, memo);
    return memo[n][k];
}

uint64_t binomialCoefficient(uint32_t n, uint32_t k) {
    std::vector<std::vector<uint64_t>> memo(n + 1, std::vector<uint64_t>(k + 1));
    return binomHelper(n, k, memo);
}
```

And here's an iterative bottom-up implementation.

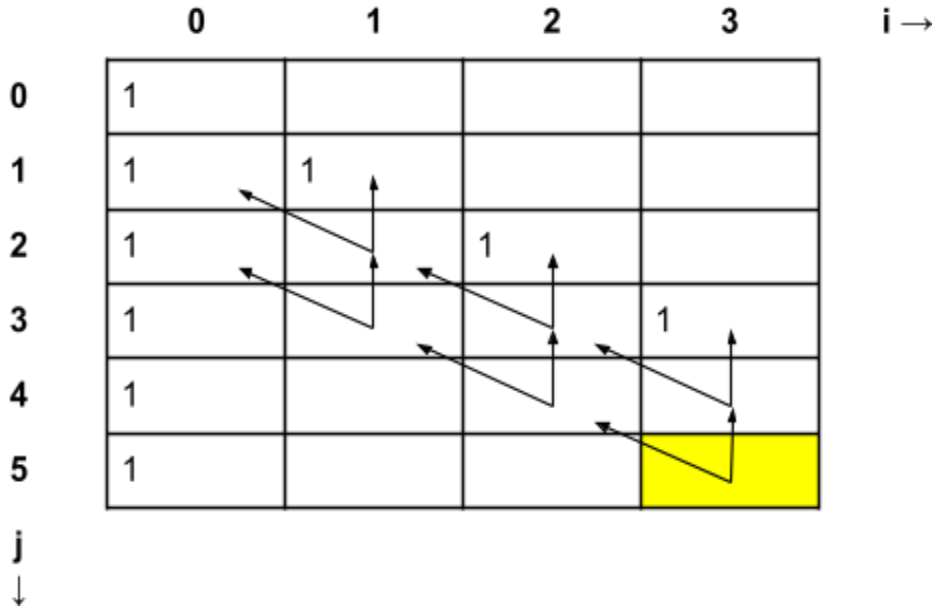
```
uint64_t binomialCoefficient(uint32_t n, uint32_t k) {
    std::vector<std::vector<uint64_t>> memo(n + 1, std::vector<uint64_t>(k + 1));
    for (size_t i = 0; i <= k; ++i)
        for (size_t j = i; j <= n; ++j)
            if ((i == j) || (i == 0))
                memo[j][i] = 1;
            else
                memo[j][i] = memo[j - 1][i - 1] + memo[j - 1][i];
    return memo[n][k];
}
```

It's now easier to see that the dynamic programming implementations run in $O(n * k)$ time. To help visualize how this works, imagine the memo as a grid of tiles, each storing some value, and

we want the value of a specific tile. For any tile that doesn't represent a base case, its value depends on:

- The value of the tile one row up and one column left (`memo[j - 1][i - 1]`)
- The value of the tile one row up (`memo[j - 1][i]`)

These two tiles represent the solutions of the subproblems for the current tile, and those tiles may themselves depend on earlier tiles, and so on. This chain continues as we move left and upward through the grid until we eventually reach the base-case tiles whose values are fixed and do not depend on any other tiles. For example, for 5 choose 3:



From this, we can see that 2 choose 1 is the first subproblem that isn't a base case whose value we can calculate (by adding the base cases 1 choose 0 and 1 choose 1). We can then calculate 3 choose 1 and 3 choose 2, then 4 choose 2 and 4 choose 3, and then finally 5 choose 3.

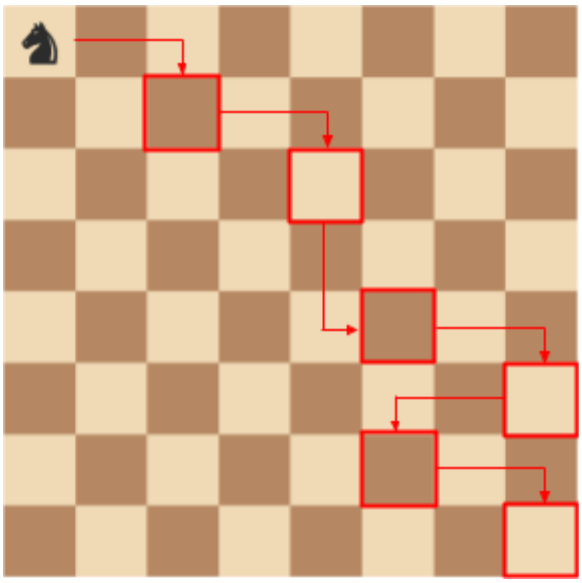
In general, the top-down approach is generally preferred because it's an easier mechanical transformation of the "natural" (recursive) solution. Furthermore, we don't need to determine which subproblems need to be solved, or what order the subproblems should be calculated in, because the algorithm will take care of those for us. In contrast, the bottom-up approach computes the solutions of all subproblems whether they're needed or not, and the programmer must carefully write the code to solve the subproblems in the correct order. Any dynamic programming problem that can be solved with a top-down approach can also be solved with a bottom-up approach.

Keep in mind that dynamic programming algorithms can use a lot of memory, especially if the problem takes in many inputs, or a top-down implementation requires many stack frames.

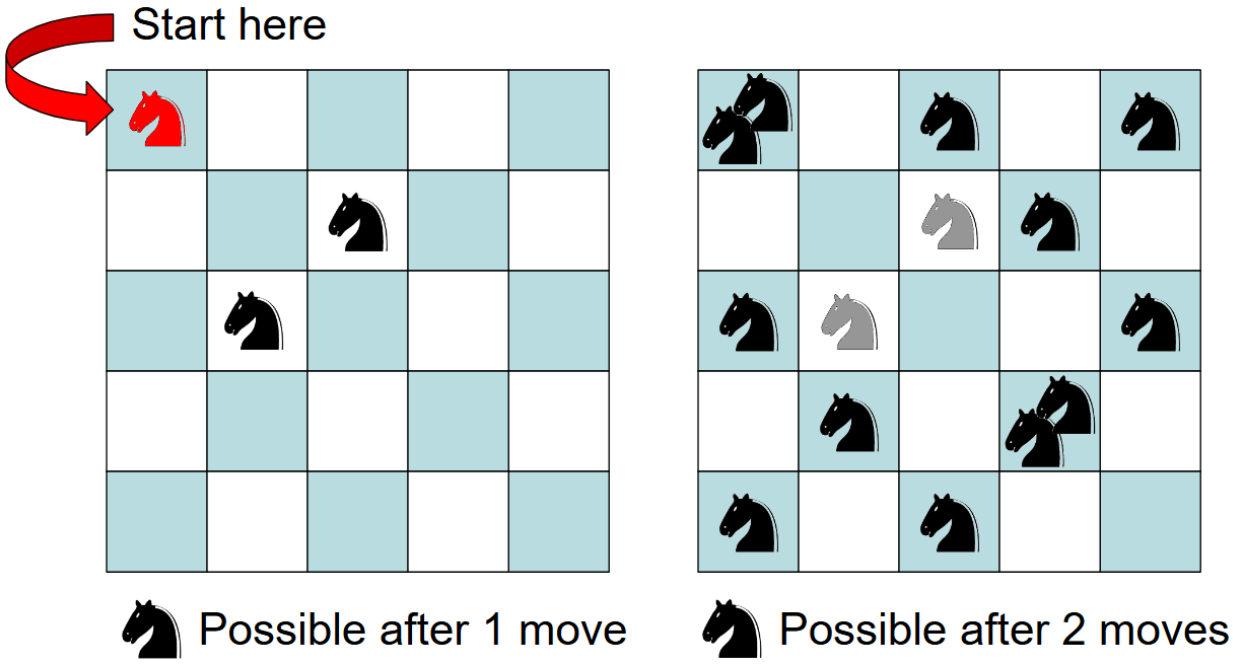
Knight Moves Problem

(Not another chess problem 😭)

The knight moves problem is “determine the number of different ways a knight can move from the top-left tile to the bottom-right tile of a standard 8×8 chessboard in 6 moves”. A knight in chess can go two tiles in one direction and one tile perpendicular to that direction in a single move. Moving a knight from the top-left tile to the bottom-right tile of a chessboard can be done in a minimum of six moves; the following image shows one way how.



This problem can be solved using dynamic programming. To make it clearer why, the following shows the tiles a knight can be on after one and two moves respectively.

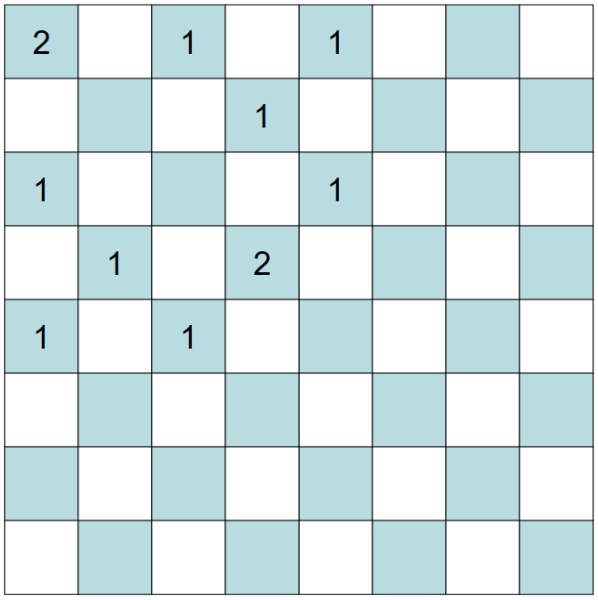


Importantly, the knight can move to tile (0, 0) or tile (3, 3) via two different move sequences, indicating that there are overlapping subproblems. Regardless of which initial 2-move partial path the knight takes to reach (3, 3), calculating the number of 4-move partial paths from (3, 3) to the final tile (7, 7) could be useful for finding the number of 6-move complete paths that begin with the two initial partial paths.

The key to solving this problem is:

- we know how many ways the knight can move to each tile in 0 moves (1 way for the top-left tile, 0 for everything else; this is the base-case)
- if we know the number of different ways a knight can move to each tile in $k - 1$ moves, then we can use that to determine the number of different ways a knight can move to each tile in k moves (this is the recursive case)

Therefore we will use a 3D memo, where the first index is the number of moves, the second index is the row, the third index is the column, and the corresponding value is the number of different ways a knight can move to the tile at the given row and column in the given number of moves. For example, `memo[2]` would look like this (empty tiles contain 0. Compare with the chessboard from above):

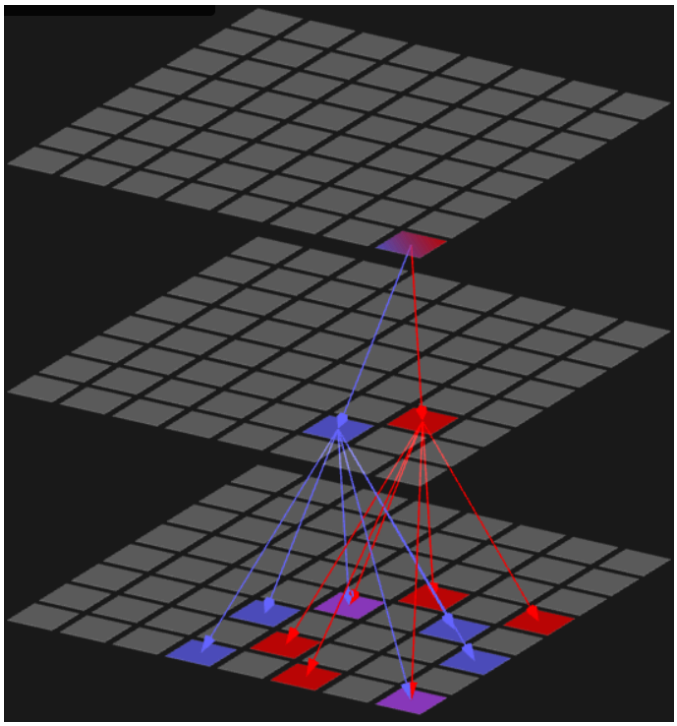


We know the values of `memo[0]`, which means we can calculate the values of `memo[1]`, which means we can calculate the values of `memo[2]`, and so on. Once we're done computing, the answer should be stored in `memo[6][7][7]`.

Below is a bottom-up implementation that is generalized to work for an arbitrary chessboard size and number of moves. For each number of moves from 1 to 6, we iterate through every tile on the board. For each tile, we examine the up to eight tiles from which a knight can reach it in one move, and we add their counts from the previous move number to the current tile's count for the current move number.

```cpp
size_t knightMoves(size_t n=8, size_t k=6) {
    std::vector<std::vector<std::vector<size_t>>> memo(
        k + 1,
        std::vector<std::vector<size_t>>(n, std::vector<size_t>(n, 0))
    );
    memo[0][0][0] = 1;
    std::vector<int> rowDiff = {2, 2, -2, -2, 1, 1, -1, -1};
    std::vector<int> colDiff = {1, -1, 1, -1, 2, -2, 2, -2};
    for (size_t numMoves = 1; numMoves <= k; ++numMoves)
        for (size_t row = 0; row < n; ++row)
            for (size_t col = 0; col < n; ++col)
                for (size_t i = 0; i < rowDiff.size(); ++i) {
                    int prevRow = row + rowDiff[i];
                    int prevCol = col + colDiff[i];
                    if (
                        prevRow >= 0 && prevRow < n &&
                        prevCol >= 0 && prevCol < n
                    )
                      memo[numMoves][row][col] += memo[numMoves - 1][prevRow][prevCol];
                }
    return memo[k][n - 1][n - 1];
}
```

We can try to visualize the memo dependencies like we did when calculating binomial coefficients, though it's more complicated since the memo is 3D. Here's a visualization of how `memo[6]` (the top layer) depends on `memo[5]`, and how `memo[5]` depends on `memo[4]` (the rest of `memo` is omitted for simplicity. The different colors are only meant to help distinguish tiles and arrows from other tiles and arrows).



(Note: for a large positive integer input `n` that can fit in a `size_t` but would overflow an `int`, this code would break- this is the danger of type casting. Such an input would be unfeasible anyway since that would require creating a massive memo that cannot fit into memory, but it should still be noted. The best practice would be to check that the user is passing in a valid positive integer that fits in an `int`, but the code snippets in these notes don't do that to make the snippets shorter. However, for reasonably small and realistic positive integer `n`, this code will work.)

For completeness, here's a top-down implementation.

```
std::vector<int> rowDiff = {2, 2, -2, -2, 1, 1, -1, -1};
std::vector<int> colDiff = {1, -1, 1, -1, 2, -2, 2, -2};

size_t knightMovesHelper(
    const int n,
    int numMoves, int row, int col,
    std::vector<std::vector<std::vector<size_t>>>& memo
) {
    if (row < 0 || row >= n || col < 0 || col >= n)
        return 0;
    if (numMoves == 0) {
        if (row == 0 && col == 0)
            return 1;
        return 0;
    }
    if (memo[numMoves][row][col] != SIZE_MAX)
        return memo[numMoves][row][col];
    size_t numWays = 0;
    for (size_t i = 0; i < rowDiff.size(); ++i)
        numWays += knightMovesHelper(
            n,
            numMoves - 1, row + rowDiff[i], col + colDiff[i],
            memo
        );
    memo[numMoves][row][col] = numWays;
    return memo[numMoves][row][col];
}

size_t knightMoves(size_t n=8, size_t k=6) {
    std::vector<std::vector<std::vector<size_t>>> memo(
        k + 1,
        std::vector<std::vector<size_t>>(n, std::vector<size_t>(n, SIZE_MAX))
    );
    return knightMovesHelper(n, k, n - 1, n - 1, memo);
}
```

The code can be optimized a lot more, but both implementations will find the correct solution, which ends up being 108, and run in $O(n^2 * k)$ time.

Bonus- Longest Increasing Subsequence

The longest increasing subsequence problem is “given an array of integers, find the length of the longest increasing subsequence in that array”. A subsequence is a sequence that can be derived from the given array by deleting some or no elements without changing the relative order of the remaining elements. For example, if the given array is `[3, 8, 5, 7, 4, 9]`, then the longest increasing subsequence is `[3, 5, 7, 9]`, so the length of the longest increasing subsequence is 4.

This problem is different from the previous problems because it’s an optimization problem rather than an enumeration problem (discussed more when solving the knapsack problem), but it can be solved with dynamic programming. Here’s a bottom-up implementation that stores the length of the longest increasing subsequence ending at `nums[i]` in `memo[i]`.

```
size_t longestIncreasingSubsequence(const std::vector<int>& nums) {
    size_t n = nums.size();
    if (n == 0) return 0;
    std::vector<size_t> memo(n, 1);
    size_t maxLen = 1;
    for (size_t i = 1; i < n; ++i) {
        for (size_t j = 0; j < i; ++j)
            if (nums[j] < nums[i] && memo[j] + 1 > memo[i])
                memo[i] = memo[j] + 1;
        if (memo[i] > maxLen)
            maxLen = memo[i];
    }
    return maxLen;
}
```

This code runs in $O(n^2)$ time and uses $O(n)$ memory.

If we were asked to return the actual longest increasing subsequence instead of its length, we could do that as follows.

```cpp
std::vector<int> longestIncreasingSubsequence(const std::vector<int>& nums) {
    size_t n = nums.size();
    if (n == 0) return {};
    std::vector<size_t> memo(n, 1);
    std::vector<size_t> parent(n, SIZE_MAX);
    size_t maxLen = 1;
    size_t maxIdx = 0;
    for (size_t i = 1; i < n; ++i) {
        for (size_t j = 0; j < i; ++j) {
            if (nums[j] < nums[i] && memo[j] + 1 > memo[i]) {
                memo[i] = memo[j] + 1;
                parent[i] = j;
            }
        }
        if (memo[i] > maxLen) {
            maxLen = memo[i];
            maxIdx = i;
        }
    }
    std::vector<int> lis(maxLen);
    for (size_t i = maxLen - 1, j = maxIdx; j != SIZE_MAX; --i, j = parent[j])
        lis[i] = nums[j];
    return lis;
}
```

Knapsack Problem

The knapsack problem is "given n items each with a size (can be a volume or weight) and value, and a knapsack whose capacity is m that the items can be placed into (capacity permitting), find the maximum value that can be packed into the knapsack". The standard version is the 0-1 knapsack problem, where there is exactly one copy of each item, and for every item our only options are to either steal the whole item (1) or leave the whole item behind (0). Variants include allowing fractional items or having access to infinitely many copies of each item. For example, given a knapsack with size $m = 11$ and the following $n = 5$ items:

	0	**1**	**2**	**3**	**4**
Size	1	2	5	6	7
Value	1	6	18	22	28

We can maximize the value packed into the knapsack by stealing items 2 and 3, whose sizes 5 + 6 ≤ 11 and whose values 18 + 22 = 40. When it comes to solving this problem, there are several approaches we can take.

Brute-force Approach

A brute-force approach can generate all 2^n possible subsets of the items. For each subset, it computes the total size and value of the selected items, discarding any subset whose size exceeds m. Among the valid subsets, it keeps track of the one with the greatest value and returns it. This always finds the correct solution, but it runs in $O(n * 2^n)$ time, which is extremely slow for large n.

Greedy Approach

With a greedy approach, we have three options:

- steal the highest value items first- but it might not work if those items are very large
- steal the smallest items first- but it might not work if those items have a very low values
- steal the highest value-density (i.e. $\frac{value}{size}$) items first - but it might not work if those items are very large

A greedy approach can first sort the items by the best value/size/value-density, and then repeatedly steal the best remaining item that fits in the knapsack. If we sort the items by the highest value density, we can always find the optimal solution to the fractional knapsack problem. However, the greedy approach can't always find the optimal solution to the general knapsack problem because the greedy choice property doesn't hold. Nevertheless, it can often find a pretty good solution in $O(n * log(n))$ time.

Dynamic Programming Approach
(This is the most difficult algorithm to understand in the course.)
The previous problems we solved with dynamic programming- calculating Fibonacci numbers, calculating binomial coefficients, and knight moves- are enumeration problems calculating the value of an expression or counting the number of solutions that don't violate any constraints. In contrast, the knapsack problem is an optimization problem that tries to find the best solution, but we can still try to solve it using dynamic programming. For optimization problems, it's usually easier to reason through a bottom-up approach.

A dynamic programming approach can find the optimal solutions for subproblems, and then use those to calculate the optimal solutions for larger problems.

To help demonstrate this, suppose a master thief tasks an us, an apprentice thief, with stealing items from a safe. Helpfully, the master thief provides the safe's password, a list of items in the safe, and a knapsack to pack those items in. Furthermore, he provides a special table that lists the maximum value that can be packed into the knapsack, and along the way he also found the maximum value that could be packed into the knapsack if the knapsack was smaller, or if only some number of the first items were in the safe (this is basically a memo!). But then when we open the safe, we're surprised to see one additional item that wasn't in the list of items! We're now faced with a choice- should we steal the new item or leave it behind?

If the new item is too large to fit in the knapsack, we have no choice but to leave it behind. Otherwise, it can fit, so we must determine:

- What is the maximum value that can be packed into the knapsack if we steal the new item?
- Is that greater than the maximum value achievable without stealing the new item?

Now looking at the example list of items given above, suppose the list the master thief provided had the first 4 items (i.e. items 0-3), but item 4 was unaccounted for.

	0	**1**	**2**	**3**	**4**
Size	1	2	5	6	7
Value	1	6	18	22	28

The master thief also provided the knapsack of size 11 and the following table, but because he didn't know that item 4 was in the safe, row 5 is empty.

first `i` items in safe ↓	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
knapsack size `j` →												
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	1
2	0	1	6	7	7	7	7	7	7	7	7	7
3	0	1	6	7	7	18	19	24	25	25	25	25
4	0	1	6	7	7	18	22	24	28	29	29	40
5												

The cell at a given row `i` and column `j` stores the maximum value that can be packed into a knapsack of size `j` if only the first `i` items, which are items 0 to `i` - 1, inclusive, are in the safe. You may notice then that row 0 stores the maximum value that can be packed if there are no items in the safe, which is obviously 0 every time. Likewise, column 0 stores the maximum value that can be packed if the knapsack is size 0, which is also 0 every time. Think of these as our base cases.

As far as the master thief knows, the maximum value that can be packed into the knapsack is 40, because that's the value in row 4 (i.e. the first 4 items are in the safe) and column 11 (i.e. using a knapsack of size 11). 40 is attained by stealing items 2 and 3 (we'll explain later how to determine that those items are the ones whose values sum to 40). We want to calculate the value stored at row 5 and column 11, because now we can potentially steal item 4, and maybe that'll allow us to pack even more value into the knapsack. Luckily, we can do that using the values that were previously calculated. Here's how (an explanation will follow).

```
struct Item {
    size_t size;
    double value;
};
```

```cpp
std::vector<bool> reconstructKnapsack(
    const std::vector<Item>& items, size_t m,
    const std::vector<std::vector<double>>& memo
) {
    size_t n = items.size();
    size_t j = m;
    std::vector<bool> stolen(n, false);
    for (size_t i = n; i > 0; --i) // memo should be of size n + 1
        if (items[i - 1].size > j)
            continue;
        else if (
            memo[i][j] <= items[i - 1].value + memo[i - 1][j - items[i - 1].size]
        ) {
            stolen[i - 1] = true;
            j -= items[i - 1].size;
        }
    return stolen;
}

std::pair<std::vector<bool>, double> knapsack(
    const std::vector<Item>& items, size_t m
) {
    size_t n = items.size();
    std::vector<std::vector<double>> memo(n + 1, std::vector<double>(m + 1, 0));
    for (size_t i = 1; i < n + 1; ++i)
        for (size_t j = 0; j < m + 1; ++j)
            if (items[i - 1].size > j)
                memo[i][j] = memo[i - 1][j];
            else
                memo[i][j] = std::max(
                    items[i - 1].value + memo[i - 1][j - items[i - 1].size],
                    memo[i - 1][j]
                );
    std::vector<bool> stolen = reconstructKnapsack(items, m, memo);
    return {stolen, memo[n][m]};
}
```

First, let's start by discussing the `knapsack` function, which is mostly just a couple of loops and an if/else statement. The outer loop iterates through the rows of the table (starting at row 1,

because row 0 is already filled with 0's), and the inner loop iterates through the columns, so the function runs in $O(n * m)$ time.

Recall that `memo[i][j]` is the maximum value that can be packed into a knapsack of size `j` if only the first `i` items are in the safe. `memo[i][j]` is much like `memo[i - 1][j]`, except that we now have the option to steal `items[i - 1]`. But if `items[i - 1]` (whose size is `items[i - 1].size`) cannot fit in the knapsack (whose size is `j`), then `items[i - 1]` cannot be stolen, so we set `memo[i][j] = memo[i - 1][j]`, which is the maximum value achievable with the same knapsack size but without stealing `items[i - 1]`.

Else, `items[i - 1]` can fit. If we pack `items[i - 1]` (whose size is `items[i - 1].size`) into the knapsack (whose size is `j`), then there will be `j - items[i - 1].size` space remaining in the knapsack to pack other items. We would like to pack the maximum value that we can into that remaining space, but behold- we already calculated it earlier! The maximum value that can be packed into a knapsack of size `j - items[i - 1].size` without stealing `items[i - 1]` is stored in `memo[i - 1][j - items[i - 1].size]`. Thus the maximum value achievable if we steal `items[i - 1]` is `items[i - 1].value + memo[i - 1][j - items[i - 1].size]`.

We then compare that value to the maximum value achievable without stealing `items[i - 1]`, which again is `memo[i - 1][j]`. We assign `memo[i][j]` to the larger of the two values, reflecting our choice to either steal or leave the item behind.

(If we wanted to draw arrows to visualize how the value stored in `memo[i][j]` depends on the values in `memo[i - 1][j - items[i - 1].size]` and `memo[i - 1][j]`, we would see that it always depends on cells that are above and/or left of it. Because we iterate through the rows of the table from top to bottom and iterate through the columns from left to right, the values in those cells should already have been calculated earlier. When writing a bottom-up dynamic programming implementation, it's important that subproblems are solved in the correct order to ensure that any subproblems that the current cell's calculation depends on have already been solved.)

For our example, we need to calculate `memo[5][11]`, which considers stealing item 4 with a knapsack of size 11. Item 4 has a size of 7, so if we steal it then we'll have 11 - 7 = 4 space remaining in the knapsack. Item 4 has a value of 28, and the maximum value we can pack into a knapsack of size 4 without stealing item 4 is `memo[4][4]` = 7, so the maximum value achievable if we steal item 4 is 28 + 7 = 35. The maximum value achievable without stealing item 4 is `memo[4][11]` = 40, which is larger. Thus we set `memo[5][11]` = 40, concluding that we should leave item 4 behind and that the maximum value that can be packed is still 40.

Now that we know the maximum value that can be packed into the knapsack, we should figure out what items were stolen in order to attain that maximum value. This is where `reconstructKnapsack` comes in. The loop iterates through the rows in reverse, so the function runs in $O(n)$ time.

We start at `memo[n][m]` and work backwards. When examining `memo[i][j]`, where `j` is the amount of space remaining in the knapsack, we are trying to determine whether `items[i - 1]` was stolen. But if `items[i - 1]` (whose size is `items[i - 1].size`) cannot fit in the knapsack (which has `j` space remaining), then `items[i - 1]` could not have been stolen, so we move onto the next item.

Else, recall that if we stole `items[i - 1]`, then we set `memo[i][j] = items[i - 1].value + memo[i - 1][j - items[i - 1].size]`. So if `memo[i][j] <= items[i - 1].value + memo[i - 1][j - items[i - 1].size]`, then we know `items[i - 1]` was stolen, so we mark it as stolen and pack it into the knapsack, reducing the knapsack's remaining space by `items[i - 1].size`. We use <= instead of == because there can be multiple valid solutions.

For our example, we start at `memo[5][11]` = 40. The best value achievable if we stole item 4 (whose size is 7) is `items[4].value + memo[4][11 - 7]` = 28 + 7 = 35, which isn't as good as 40, so item 4 wasn't stolen. Next we move to `memo[4][11]` = 40. The best value achievable if we stole item 3 (whose size is 6) is `items[3].value + memo[3][11 - 6]` = 22 + 18 = 40, which is as good as 40, so item 3 was stolen and now have 11 - 6 = 5 space remaining in the knapsack. Next we move to `memo[3][5]` = 18. The best value achievable if we stole item 2 (whose size is 5) is `items[2].value + memo[2][5 - 5]` = 18 + 0 = 18, which is as good as 18, so we item 2 was stolen and now have 5 - 5 = 0 space remaining in the knapsack. The loop would run for a couple more iterations, but at this point no more items can fit in the knapsack because there's no space remaining, so the stolen items are items 2 and 3.

Branch-and-bound Approach
It was previously mentioned that the knapsack problem is an optimization problem, and we know that we can solve optimization problems using a branch-and-bound approach. But unlike the travelling salesman problem that aims to minimize cost, the knapsack problem aims to maximize value.

To start the algorithm, it works much like a brute-force algorithm- we must consider every possible selection of items to steal, so we generate subsets of the items (just like `genSubsets` from earlier), but now we prune branches that aren't promising.

When examining a candidate solution, which is a selection of items, we consider it promising if:

- the selection's size is less than or equal to the knapsack's size (i.e. doesn't violate the size constraint)
- the upper bound of the selection's value is at least the total value of the best selection found so far- the lower bound

To implement the promising check then, we need an efficient way to calculate a good overestimate of the maximum value of a (potentially larger) selection that contains all of the items in a given (potentially smaller) selection- the upper bound of the given selection. We can do this by adding:

- the given selection's value
- the fractional knapsack solution for the remaining space in the knapsack using the unselected items- solved by greedily stealing the unselected items in order of highest value-density

Here's a branch-and-bound implementation (you wouldn't be asked to write this during an exam).

```
double fractionalKnapsackValue(
    const std::vector<Item>& items, size_t m,
    size_t i,
    size_t selectionSize, double selectionValue
) {
    size_t spaceRemaining = m - selectionSize;
    if (spaceRemaining == 0)
        return selectionValue;
    size_t n = items.size();
    std::vector<std::pair<double, size_t>> valueDensities(n - i);
    for (size_t k = i; k < n; ++k)
        valueDensities[k - i] = {items[k].value / items[k].size, k};
    std::sort(begin(valueDensities), end(valueDensities), [](auto& a, auto& b) {
        return a.first > b.first;
    });
    double upperBound = selectionValue;
    for (auto& vd : valueDensities) {
        size_t itemIndex = vd.second;
        if (items[itemIndex].size <= spaceRemaining) {
            upperBound += items[itemIndex].value;
            spaceRemaining -= items[itemIndex].size;
        }
        else {
            upperBound += vd.first * spaceRemaining;
            break;
        }
    }
    return upperBound;
}
```

```cpp
bool promising(
    const std::vector<Item>& items, size_t m,
    double lowerBound,
    const std::vector<bool>& currentSelection, size_t i,
    size_t selectionSize, double selectionValue
) {
    if (selectionSize > m) return false;
    double upperBound = fractionalKnapsackValue(
        items, m,
        i,
        selectionSize, selectionValue
    );
    return upperBound > lowerBound;
}
```

```cpp
void knapsackHelper(
    const std::vector<Item>& items, size_t m,
    std::vector<bool>& bestSelection, double& lowerBound,
    std::vector<bool>& currentSelection, size_t i,
    size_t selectionSize, double selectionValue
) {
    if (i == items.size()) {
        if (selectionValue > lowerBound) {
            bestSelection = currentSelection;
            lowerBound = selectionValue;
        }
        return;
    }
    if (
        promising(
            items, m,
            lowerBound,
            currentSelection, i,
            selectionSize, selectionValue
        )
    ) {
        currentSelection[i] = true;
        knapsackHelper(
            items, m,
            bestSelection, lowerBound,
            currentSelection, i + 1,
            selectionSize + items[i].size, selectionValue + items[i].value
        );
        currentSelection[i] = false;
        knapsackHelper(
            items, m,
            bestSelection, lowerBound,
            currentSelection, i + 1,
            selectionSize, selectionValue
        );
    }
}
```

```cpp
std::pair<std::vector<bool>, double> knapsack(
    const std::vector<Item>& items, size_t m
) {
    size_t n = items.size();
    std::vector<bool> bestSelection(n, false);
    double lowerBound = 0;
    std::vector<bool> currentSelection(n, false);
    knapsackHelper(
        items, m,
        bestSelection, lowerBound,
        currentSelection, 0,
        0, 0
    );
    return {bestSelection, lowerBound};
}
```

Branch-and-bound will find the optimal solution, but it runs in $O(2^n * n * log(n))$, much slower than dynamic programming (though faster than brute force in practice because of pruning), since it generates up to 2^n selections of items and sorts each one in $O(n * log(n))$ time. But it uses only $O(n)$ memory, which is less than dynamic programming.