Unit Overview
Recursion & Searching

Major Topics

✓ Introduction to recursion
✓ Search algorithms I
  ✓ Recursion vs. iteration
  ✓ Search algorithms II
  ✓ Eliminating recursion
✓ Search in the C++ STL
✓ Recursive backtracking
Review
The **Binary Search** algorithm finds a given key in a sorted list.

- Here, key = thing to search for. Often there is associated data.
- In computing, sorted means in (some specified) order.

**Procedure**

- Pick an item in the middle of the list: the **pivot**.
- Compare the given key with the pivot.
- Using this, narrow search to top or bottom half of list. Recurse.

**Example:** Use Binary Search to search for 64 in the following list.

In my illustrations, keys will be integers. In practice, a key could be just about anything that can be sorted.
Equality vs. Equivalence—may not be the same when objects being compared are not numbers.

- **Equality**: `a == b`.
- **Equivalence**: `!(a < b) && !(b < a)`.

Using equivalence instead of equality in Binary Search:
- Maintains consistency: always compare with `operator<`.
- Allows use with value types that do not have `operator==`.

---

**Using Operators**
Random-access iterators only

**Using STL Function Templates**
Works with all forward iterators
Still fast with random-access

<table>
<thead>
<tr>
<th>Using Operators</th>
<th>Using STL Function Templates</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iter += n</code></td>
<td><code>std::advance(iter, n)</code></td>
</tr>
<tr>
<td><code>iter2 - iter1</code></td>
<td><code>std::distance(iter1, iter2)</code></td>
</tr>
</tbody>
</table>
Recursion vs. Iteration
Recursion vs. Iteration
Fibonacci Again — Faster

We wrote a function that, given \( n \), returns Fibonacci number \( n \). For \( n > 40 \), our function is extremely slow. What can we do about this?

TO DO
- Rewrite the Fibonacci computation in a fast **iterative** form (using loops).

Wow! Recursion is a *lot* slower than iteration!

TO DO
- Figure out how to do a fast **recursive** Fibonacci computation. Write it.

Not necessarily.

Done. See fibo_iterate.cpp.

Done. See fibo_recurse.cpp.
Recursion vs. Iteration
Fibonacci Again — Note on Trees

Use a **tree** to represent function calls some algorithm makes.

- A box represents making a call to a function.
- A line from an *A* box down to a *B* box represents this call to function *A* making a call to function *B*.

```c
int ff(int n)
{
    return gg(n-1) + gg(n);
}

int gg(int k)
{
    if (k <= 1) return 7;
    else        return 2*gg(k-1);
}
```

Tree representing calls made by doing `ff(2)`

- `ff(2)`
- `gg(1)`
- `gg(2)`
- `gg(1)`

Same function. Different **invocations** of that function.
Choice of algorithm can make a *huge* difference in performance.

### Computing $F_6$

<table>
<thead>
<tr>
<th>Fibonacci No.</th>
<th>fibo_recurse.cpp</th>
<th>fibo_first.cpp</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F_7$</td>
<td>9 calls</td>
<td>41 calls</td>
</tr>
<tr>
<td>$F_{10}$</td>
<td>12 calls</td>
<td>177 calls</td>
</tr>
<tr>
<td>$F_{20}$</td>
<td>22 calls</td>
<td>21,891 calls</td>
</tr>
<tr>
<td>$F_{40}$</td>
<td>42 calls</td>
<td>331,160,281 calls</td>
</tr>
</tbody>
</table>
A struct can be used to return two values at once.

- Templates `std::pair` and `std::tuple` can be helpful.

The 2017 C++ Standard introduced **structured bindings**, making this more convenient.

```cpp
tuple<bignum, bignum> fibo_recurse(int n);

auto [a, b] = fibo_recurse(k);
```

Now `a` and `b` are variables of type `bignum`.
- `a` is `fibo(k-1)`.
- `b` is `fibo(k)`. 
Some algorithms have natural implementations in both recursive and iterative form.

Sometimes we have a workhorse function that does most of the processing, and a wrapper function with a convenient interface.

- Often the wrapper just calls the workhorse for us.
- This is common when we use recursion, since recursion can place restrictions on how a function is called.

We have seen this idea in another context. Recall toString and operator<< from Project 1.

```cpp
cout << p.toString();
```

If we had not written our own operator<<, then we could still do this.

```cpp
cout << p;
```

With our operator<<, we can do this. So operator<< is really just a convenient wrapper around toString.
Recursion vs. Iteration
Function-Call Internals [1/4]

To fully grasp the issues involved in recursion vs. iteration, it helps to understand how function calls are implemented.

A running program makes use of a structure called the call stack (there are other names, all involving the word “stack”).

A Stack is a kind of container. We will look at Stacks in detail later in the semester. For now:

- Think of a stack of plates. We can place a new plate on top, and we can pull a plate off the top. We only deal with the top of the Stack.
- Adding something on top is a push operation. Taking something off the top is a pop operation.
The items on the call stack are **stack frames**. Each stack frame corresponds to an **invocation** of a function.

- A function’s stack frame holds:
  - Its automatic variables, including parameters.
  - Its **return address**: where to go back to when it returns.
- When a function is called, a stack frame for that function is pushed.
- When the function exits, its stack frame is popped.

```c
void cat(Foo c) {
    int d;
    llama();
    ...
}

void dog(int a) {
    Foo b;
    cat(b);
}
```

At **this** point in the code (assuming `cat` was called by `dog`), the call stack looks like **this**.
Recursion vs. Iteration
Function-Call Internals [3/4]

When a function calls itself recursively, there will be multiple stack frames on the call stack corresponding to the same function—but different invocations of that function.

```cpp
void zebra(int n)
{
    if (n == 0)
    {
        cout << n << endl;
        return;
    }
    cout << n << " ";
    zebra(n-1);
}
```
Recursion vs. Iteration
Function-Call Internals [4/4]

A function call’s **recursion depth** is the greatest number of stack frames on the call stack *at any one time* as a result of the call.

When analyzing *time* usage, the total number of calls is of interest. When analyzing *space* usage, the recursion depth is of interest.
Two factors can make recursive algorithms inefficient.

- Inherent inefficiency of some recursive algorithms
  - But there are efficient recursive algorithms.
- **Function-call overhead**
  - Making all those function calls requires work: pushing and popping stack frames, saving return addresses, creating and destroying automatic variables.

And recursion has another problem.

- **Memory-management issues**
  - A high recursion depth causes the system to run out of memory for the call stack. This is **stack overflow**, and it generally cannot be dealt with using normal error-handling procedures. The result is usually a crash.
  - When we use iteration, we can manage memory ourselves. This can be more work for the programmer, but it also allows proper error handling.
Dynamic programming (which does not mean what it sounds like) can greatly speed up some recursive algorithms.

- It involves saving the results of recursive calls so that they do not need to be recomputed.
- In some contexts, this technique is called memoizing.
- Dynamic programming is covered in CS 411.

If we apply dynamic programming to fibo_first.cpp, then the recursive calls shown in blue no longer need to be made.

See fibo_dp_topdown.cpp.
There is a simple formula for $F_n$, using non-integer computations. Let $\varphi = \frac{1 + \sqrt{5}}{2} \approx 1.6180339$. (This is often called the golden ratio.)

For each nonnegative integer $n$, $F_n$ is the nearest integer to $\frac{\varphi^n}{\sqrt{5}}$.

Here is `fibo` using this formula:

```c
bignum fibo(int n)
{
    long double phi = (1.0L + sqrt(5.0L)) / 2.0L;
    long double near_fibo = pow(phi, n) / sqrt(5.0L);
    // Our Fibonacci number is the nearest integer
    return bignum(near_fibo + 0.5L);
}
```

A floating-point literal with an “L” added at the end is of type `long double`.

See `fibo_formula.cpp`. 

2020-09-18
An even faster method of computing Fibonacci numbers relies on the following facts:

- \( F_{2n-1} = (F_{n-1})^2 + (F_n)^2. \)
- \( F_{2n} = 2F_{n-1}F_n + (F_n)^2. \)

For the fast methods we mentioned earlier, computing \( F_n \) requires something like \( n \) arithmetic operations. But using the above facts, we can compute \( F_n \) using something like \( \log n \) arithmetic operations—much less, when \( n \) is large.

This allows for easy computation of Fibonacci numbers that are much larger than any C++ built-in integer type can hold. To illustrate the power of this method, I have implemented it in Python, which has a built-in arbitrarily large integer type.

See `fibo_fast.py`. 
A single problem may be solvable by many different methods.

- Different methods can have very different performance characteristics.
- It is possible that a very efficient method is not at all obvious.

Computing Fibonacci numbers is not something we need to do very often, in practice. But the above observations apply to other problems as well.

Next we will return to the problem of finding a key in a list.