Search Algorithms continued
Recursion vs. Iteration

CS 311 Data Structures and Algorithms

Lecture Slides

Monday, September 28, 2009

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
CHAPPELLG@member.ams.org

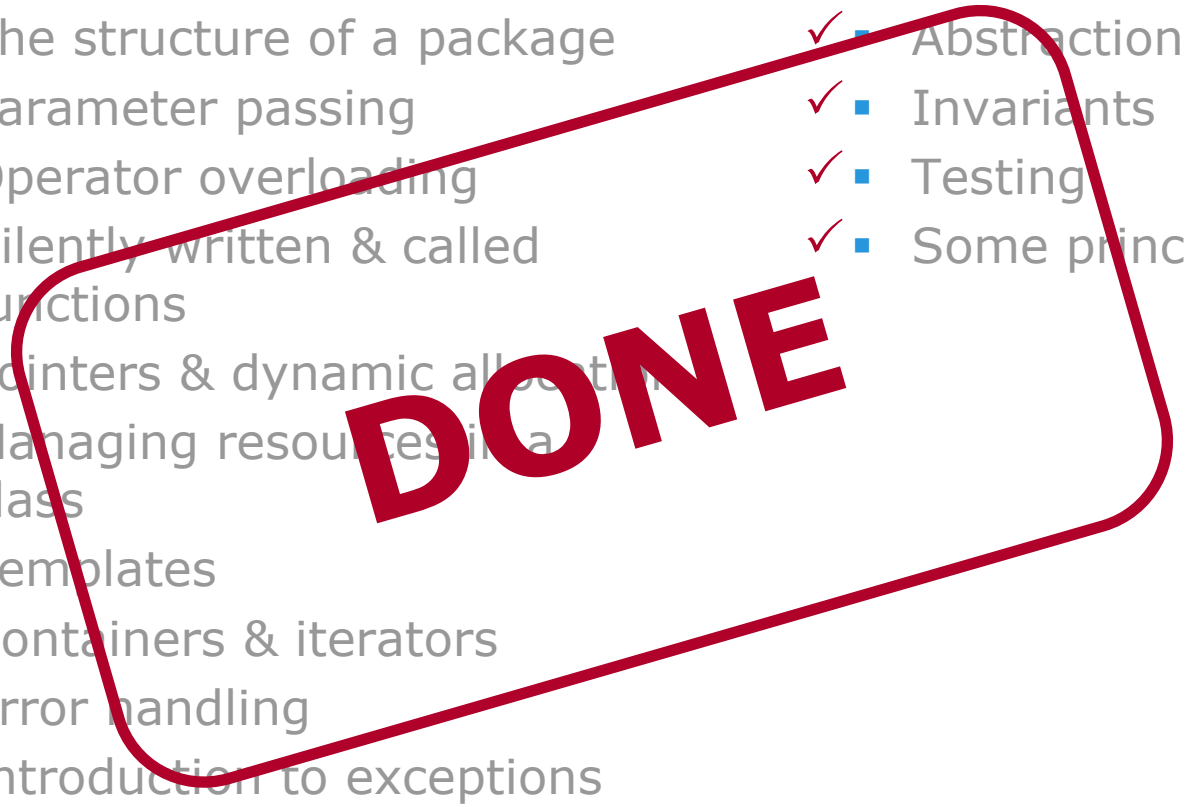
Unit Overview
Advanced C++ & Software Engineering Concepts

Major Topics: Advanced C++

- ✓ The structure of a package
- ✓ Parameter passing
- ✓ Operator overloading
- ✓ Silently written & called functions
- ✓ Pointers & dynamic allocation
- ✓ Managing resources in a class
- ✓ Templates
- ✓ Containers & iterators
- ✓ Error handling
- ✓ Introduction to exceptions
- ✓ Introduction to Linked Lists

Major Topics: S.E. Concepts

- ✓ Abstraction
- ✓ Invariants
- ✓ Testing
- ✓ Some principles



Unit Overview
Recursion & Searching

Major Topics

- ✓ Introduction to Recursion
- (part) Search Algorithms
- Recursion vs. Iteration
- Eliminating Recursion
- Recursive Search with Backtracking

Review
Introduction to Recursion

We looked at a recursive implementation of a function to return $1 + 2 + \ldots + n$, given n.

Some points to remember:

- Recursive code must have a **base case**. And every call to the recursive code must eventually reach a base case.
- **Recurrence relations** often turn naturally into recursive code.

$f(n) = f(n-1) + n$ ⟶ `return sumUpTo(n-1) + n;`

- Recursion is often not the best method. **Iteration** = loops.
- And sometimes we do not even need that.

```
return n * (n+1) / 2;  // 1 + 2 + ... + n
```

Review
Search Algorithms

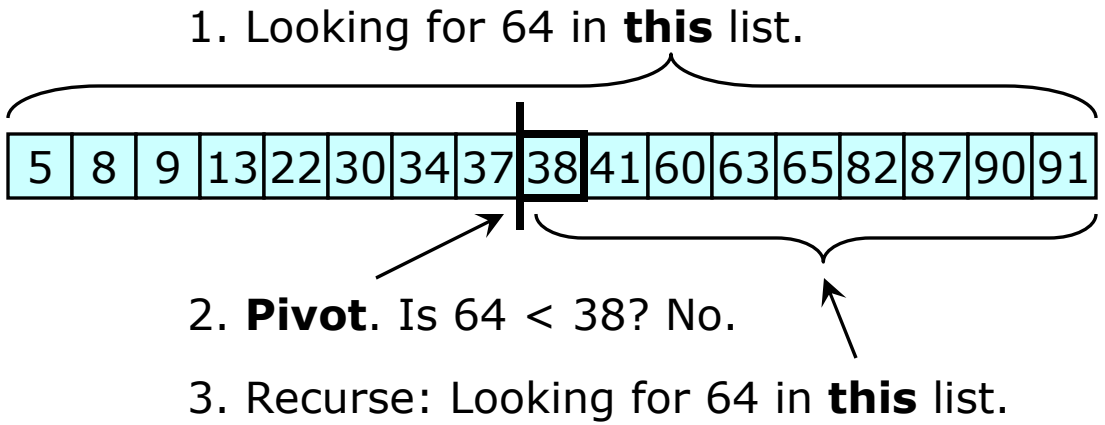
Binary Search is an algorithm to find a given **key** in a **sorted list**.

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

Procedure

- Pick an item in the middle: the **pivot**.
- Use this to narrow search to top or bottom half of list. Recurse.

Example: Binary Search for 64 in the following list.



Binary Search Example — Four Questions

How can you define the problem in terms of a smaller problem of the same type?

- Look at the middle of the list. Then recursively search the top or bottom half, as appropriate.

How does each recursive call diminish the size of the problem?

- It cuts the size of the list in half (roughly).

What instance of the problem can serve as the base case?

- List size is 0 or 1.

As the problem size diminishes, will you reach this base case?

- Yes.
  - A list cannot have negative size.

Suppose we wish to write a function to do Binary Search.

- How should the list to search in be given?
  - Parameters: Two iterators, one pointing to first item and one pointing just past last item.
- Should there be any other parameters?
  - Yes, the value to search for.
- What should the function return?
  - There are several options: just **`true`** or **`false`** (found or not), an iterator to the value found, an iterator to the first equal value in the list, two iterators specifying the range of equal values, etc.
  - Our function will return a **`bool`** indicating whether the value was found.

Binary Search Example — Do It

TO DO

- Write a function **`binSearch`** to do Binary Search, as discussed.

Done. See **`binsearch1.cpp`**, *on the web page.*

Function **`binSearch`** only determines **whether** an item is in a list. It does not tell **where** it is in the list.

All less-than/greater-than comparisons on objects of the value type are performed with **`operator<`**.

Document "requirements on types", to indicate which types the function can be used with.

- Iterators given (**`RAIter`**) must be random-access iterators.
  - Recall: "Random-access iterator" is a standard *iterator category*.
- Dereferencing an **`RAIter`** must give a **`ValueType`**.
- Type **`ValueType`** needs copy ctor, **`operator<`**, **`operator==`**, dctor.
  - To think about: Can we make do with less than this?

We also need pre- & postconditions.

- Useful idea: A **valid range** is one in which, if we start at the beginning and increment repeatedly, we will eventually reach the end, and all the data we pass through in the interim, are valid.

Search Algorithms
Binary vs. Sequential Search [1/3]

We have discussed **Binary Search**. Another algorithm is **Sequential Search**.

- Also called "Linear Search".
- In *Sequential Search*, we search a list from beginning to end, looking at each item until we either find what we are searching for, or we reach the end of the list.

Sequential Search is applicable to more situations than Binary Search. To do a Binary Search efficiently, the list should be:

- Sorted (required for Binary Search)
- Random-Access (for efficiency)

So, why do we like Binary Search?

Search Algorithms
Binary vs. Sequential Search [2/3]

We like Binary Search better than Sequential Search, because it is **much faster** …

Number of Items in List	Look-Ups: Binary Search* (worst case)	Look-Ups: Sequential Search (worst case)
1	1	1
2	2	2
4	3	4
100	8	100
10,000	15	10,000
1,000,000	21	1,000,000
10,000,000,000	35	10,000,000,000
n	*Roughly* $\log_2 n$	n

*Using our version of the algorithm. Other variations may differ slightly, but the main point of this table remains valid.

Search Algorithms
Binary vs. Sequential Search [3/3]

… and, therefore, the amount of data it can process in the **same** amount of time is **much greater**.

Number of Look-Ups We Have Time to Perform	Maximum Allowable List Size: Binary Search	Maximum Allowable List Size: Sequential Search
1	1	1
2	2	2
3	4	3
4	8	4
10	512	10
20	524,288	20
40	549,755,813,888	40
k	*Roughly* 2^k	*k*

"The fundamental law of computer science: As machines become more powerful, the efficiency of algorithms grows more important, not less."
— Nick Trefethen

Better Binary Search — Equality vs. Equivalence

Let's improve our Binary Search function.

First, some ideas.

If we use **`operator<`** to search for an item, then we probably should also use **`operator<`** to make sure we have the correct item.

- This allows us to handle types that:
  - Do not have **`operator==`**.
  - Have **`operator==`**, but do not define it in a way that is consistent with **`operator<`**.
- "**`a == b`**" is **equality**.
- "**`!(a < b) && !(b < a)`**" is **equivalence**.

Random-access iterators can do pointer-style arithmetic:

- Adding Integers
  - `iter1 = iter2 + 3;`
  - `iter1 += 3;`
- Difference
  - `n = iter1 - iter2;`

In general, iterators cannot do all this.

However, we can get the same results for more general **forward** iterators, using **`std::advance`** and **`std::distance`**.

- These are defined in **`<iterator>`**.
- "**`std::advance(iter, n)`**" is like "**`iter += n`**".
- "**`std::distance(iter1, iter2)`**" is like "**`iter2 - iter1`**".
- These two functions are fast for random-access iterators; they may be slower for other iterators.
- See the STL doc's.

Better Binary Search — Do It

TO DO

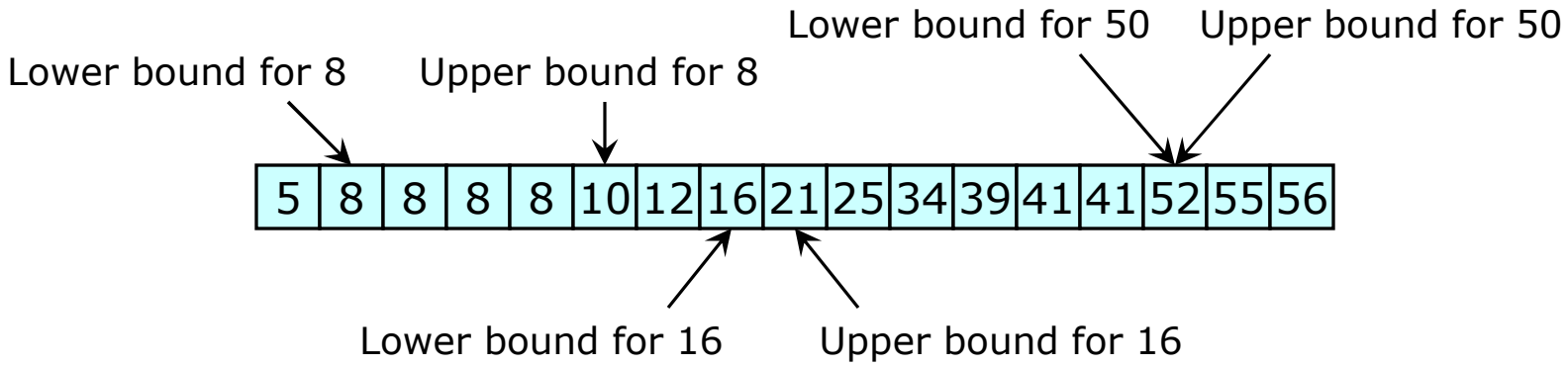
- Improve function **`binSearch`**:
  - Avoid redundant computations in finding size of list, base case.
  - Do not require **`ValueType`** to have a copy ctor, dctor, or **`operator==`**.
    - Use **only** **`operator<`**.
  - Allow the iterators to be forward iterators.
    - Is this really an improvement? Maybe …

Done. See **`binsearch2.cpp`**, *on the web page.*

Search Algorithms
Search in the C++ STL [1/2]

The C++ Standard Template Library includes Binary Search.

- Function **`std::binary_search`** (header **`<algorithm>`**) searches and returns a **`bool`** indicating success/failure.
- The following functions (also **`<algorithm>`**) return iterators to where the value was found, or where it could be inserted.
  - **`std::lower_bound`**
  - **`std::upper_bound`**
  - **`std::equal_range`**



Search Algorithms
Search in the C++ STL [2/2]

These functions (**`std::binary_search`** in particular) are very similar to our version:

- 3 parameters: two iterators specifying a range and a value to search for.
- They are templates, and they work for a wide range of types.
- They require the data to be sorted.
- They are faster on random-access data, but they do not require it.
- They use only **`operator<`** on the value type, and search based on equivalence, not equality.

In addition they have alternate forms that allow the client to specify a comparison other than **`operator<`**.

- This is typical of the STL.
- Why is this a (very) good idea?

Sequential Search is also available in the STL.

- It is called **`std::find`**, also declared in **`<algorithm>`**.
- It identifies the item to be found using equality (**`==`**), not equivalence.

Fibonacci Numbers — Introduction

The **Fibonacci numbers** are 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, …

- To get the next Fibonacci number, add the two before it.

They are defined formally as follows:

- We denote the nth Fibonacci number by F_n (n = 0, 1, 2, …).
- $F_0 = 0$.
- $F_1 = 1$.
- For $n \geq 2$, $F_n = F_{n-2} + F_{n-1}$.

Another recurrence relation

As before, recurrence relations often translate nicely into recursive algorithms.

TO DO

- Write a recursive function to compute a Fibonacci number: given n, compute F_n.

Done. See **`fibo1.cpp`**, *on the web page.*

Recursion vs. Iteration
TO BE CONTINUED …

Recursion vs. Iteration will be continued next time.