CS103 Unit 1e – Algorithms and Runtime; Compilation and Debugging

Mark Redekopp

ALGORITHMS AND RUNTIME

Algorithms

- Algorithms are at the heart of computer systems, both in hardware (HW) and software (SW)
  - They are fundamental to Computer Science and Computer Engineering
- **Informal definition**
  - An algorithm is a **precise** way to accomplish a task or solve a problem
- A more **formal definition**:
  - An **ordered** set of **unambiguous**, **executable** steps that defines a **terminating** process (**see inset**)
- Examples: What is the algorithm for
  - Brushing your teeth?
  - Calculating your GPA?

- **Ordered**: the steps of an algorithm have a particular order, not just any order
- **Unambiguous**: each step is completely clear as to what is to be done
- **Executable**: Each step can actually be performed
- **Terminating Process**: Algorithm will stop, eventually. (Sometimes this requirement is relaxed)

Algorithm Representation

- An algorithm focuses on how to solve a problem regardless of the language or specific implementation.
  - An algorithm is not a program NOR a programming language
- Just as a story may be represented as a book, movie, or spoken by a story-teller, an algorithm may be represented in many ways
  - Flow chart
  - Pseudocode (English-like syntax using primitives that most programming languages would have)
  - A specific program implementation in a given programming language like C++

Pseudocode Primitives

- **Assignment**:

  name ← expression

  – *name* is a descriptive name/variable and *expression* describes the value to be associated with name

- **Select one of two possible choices (conditionals)**:

  if (condition) **then** (activity) **else** (activity)

  if (condition) **then** (activity)

- **Repeated execution of statements (loops):**

  while (condition) **do** (activity)

  repeat (activity) **until** (condition)

  foreach name **in** (set / collection) **do** (activity)

Algorithm Example 1

- List/print all factors of a natural number, **n**
  - How would you check if a number is a factor of **n**?
  - What is the range of possible factors?

```
i ← 1
while(i <= n) do
  if (remainder of n/i is zero) then
      List i as a factor of n
  i ←  i+1
```

- An improvement

```
i ← 1
while(i <= ________ ) do
  if (remainder of n/i is zero) then
      List i and _____ as a factor of n
  i ←  i+1
```

Algorithm Time Complexity

- We often judge algorithms by how long they take to run for a given input size, **n**
- Algorithms often have different run-times based on the input size [e.g. **n** = # of elements in a list to search or sort]
  - Different input patterns can lead to best- and worst-case times
  - Average-case times can be helpful
  - But we usually use worst case times for comparison purposes
- We also want to be able to compare the time an algorithm takes in a way that is independent of the computer running it (i.e. the same algorithm might run a lot faster on a server than your phone due to the hardware)

Calculating Runtime

- Given an input to an algorithm of size **n**, we start by deriving an expression (in terms of n) for the steps of work an algorithm performs, usually for its **WORST CASE** run time
- Just walk the code and count up "steps" of work
  - It is fine if what what we call a "step" of work is a bit imprecise when we apply big-O notation.

```
i ← 1
while( i <= n ) do
  if (n mod i == 0) then
      List i as a factor of n
  i ← i+1
```

Applying Big-O Notation

- From the expression, we find the corresponding big-O expression for that runtime by assuming n is LARGE (approaching infinity) and, thus, we:
  - Only keep the dominant term in the expression
  - Discard constant coefficients. The result is the big-O (worst-case or upper-bound) run-time
- Example:
  - If an algorithm with input size of n runs in **$3n^2 + 10n + 1000$** steps, we say that it runs in **$O(n^2)$** because if n is large **$3n^2$** will dominate the other terms

```
i ← 1                                          1
while( i <= n ) do                             1*n
  if (remainder of n/i is zero) then           2*n
     List i as a factor of n                   1*n
  i ← i+1                                      1*n
```

5n+1

O(5n+1) = O(n)

Big-O notation lets us express the amount of work an algorithm performs (or what we define as a "step" of work) in a way that doesn't require exact precision but helps us compare the efficiency of different algorithms.

Algorithm Example 1

- List/print all factors of a natural number, **n**
  - What is a factor?
  - What is the range of possible factors?

```
i ← 1
while(i <= n) do
  if (remainder of n/i is zero) then
    List i as a factor of n
  i ←  i+1
```

O(n)

- An improvement

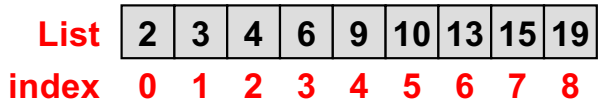
```
i ← 1
while(i <= sqrt(n) ) do
  if (remainder of n/i is zero) then
    List i and n/i as a factor of n
  i ←  i+1
```

O($\sqrt{n}$)

Algorithm Example 2a

- Searching an ordered list (array) for a specific value, k, and return its index or -1 if it is not in the list
- Sequential Search
  - Start at first item, check if it is equal to k, repeat for second, third, fourth item, etc.

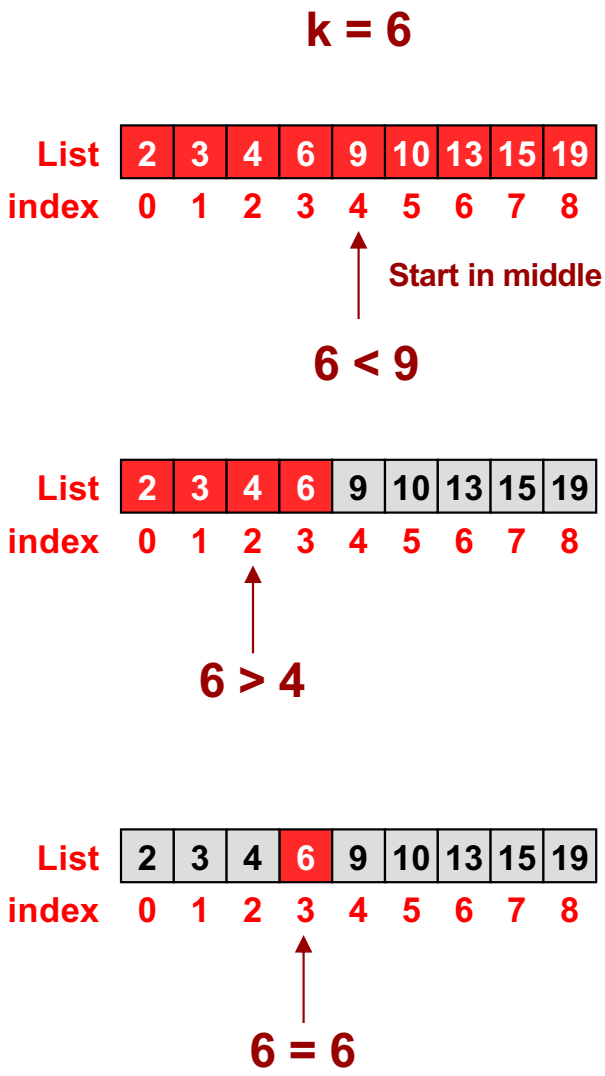
k = 12




```
i ← 0
while ( i < length(List) ) do
  if (List[i] equal to k) then
    return i
  else i ← i+1
return -1
```

Algorithm Example 2b

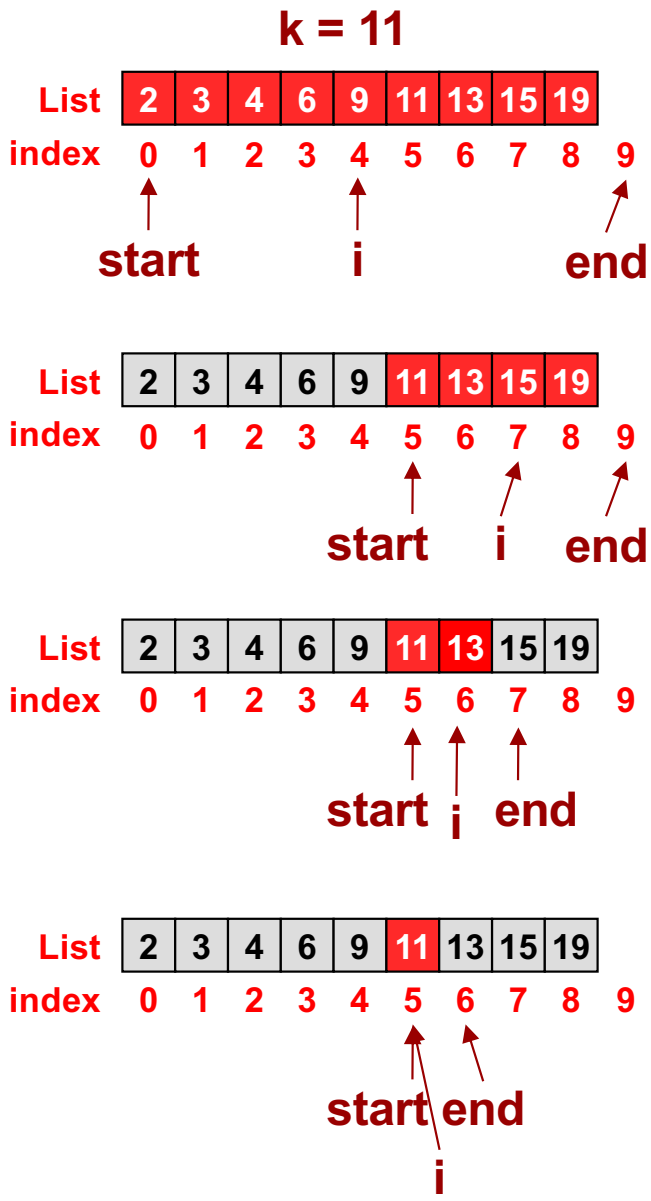
- Sequential search does not take advantage of the ordered nature of the list
  - Would work the same (equally well) on an ordered or unordered list
- Binary Search
  - Take advantage of ordered list by comparing k with middle element and based on the result, rule out all numbers greater or smaller, repeat with middle element of remaining list, etc.



Algorithm Example 2b

- Binary Search
  - Compare k with middle element of list and if not equal, rule out ½ of the list and repeat on the other half
  - Implementation:
    - Define range of searchable elements = [start, end) (i.e. start is inclusive, end is exclusive)

```
start ← 0; end ← length(List);
while (start != end) do
    i ← (start + end) /2;
    if ( k == List[i] ) then return i;
    else if ( k > List[i] ) then
         start ← i+1;
    else
         end ← i;
return -1;
```

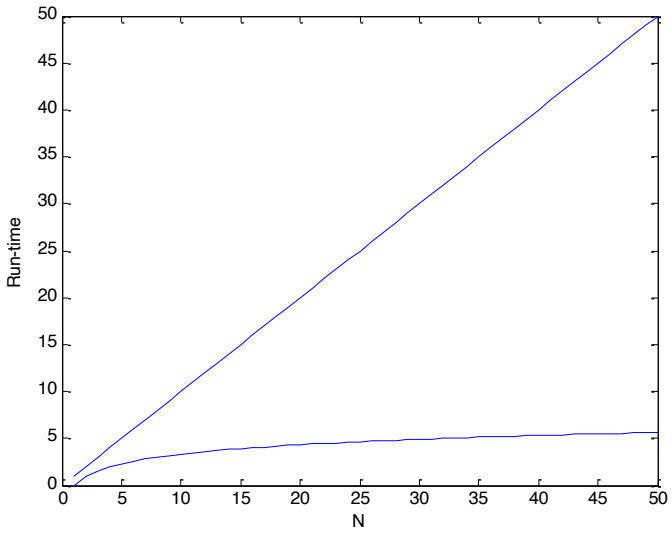


Complexity of Search Algorithms

- Sequential Search: List of length n
  - Worst case: Search through entire list
  - Time complexity = an + k
    - a is some constant for number of operations we perform in the loop as we iterate
    - k is some constant representing startup/finish work (outside the loop)
  - Sequential Search = O(n)
- Binary Search: List of length n
  - Worst case: Continually divide list in two until we reach sublist of size 1
  - Time = $a*\log_2 n + k = O(\log_2 n)$
- As n gets large, binary search is far more efficient than sequential search

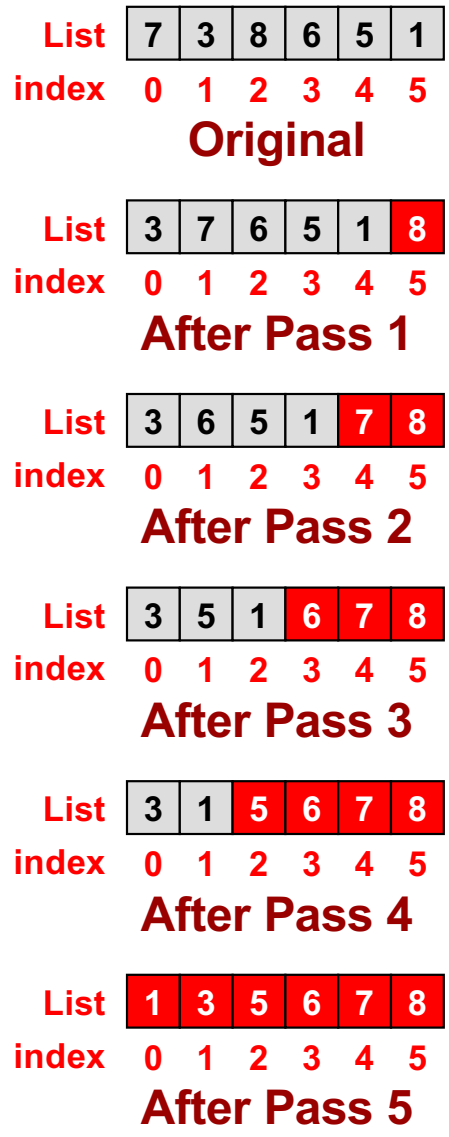
Dividing by 2 for k-times yields: $n / 2^k = 1$

$k = \log_2 n$



Sorting

- If we have an unordered list, sequential search becomes our only choice
- If we will perform a lot of searches, it may be beneficial to sort the list, then use binary search
- Many sorting algorithms of differing complexity (i.e. faster or slower)
- Bubble Sort (simple though not terribly efficient)
  - On each pass through thru the list, pick up the maximum element and place it at the end of the list. Then repeat using a list of size n-1 (i.e. w/o the newly placed maximum value)

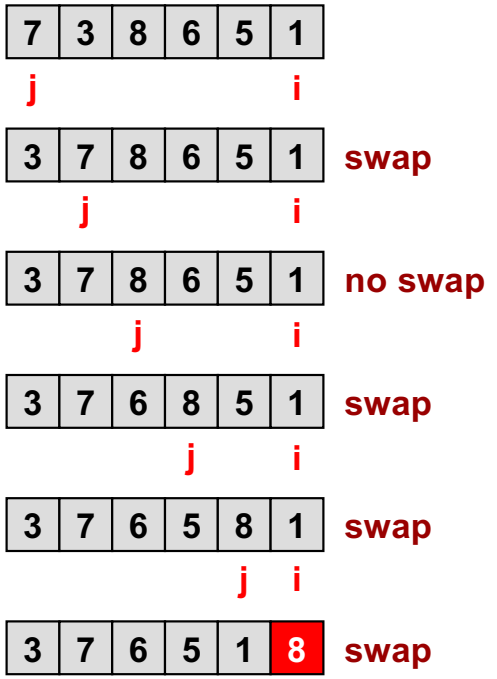


Bubble Sort Algorithm

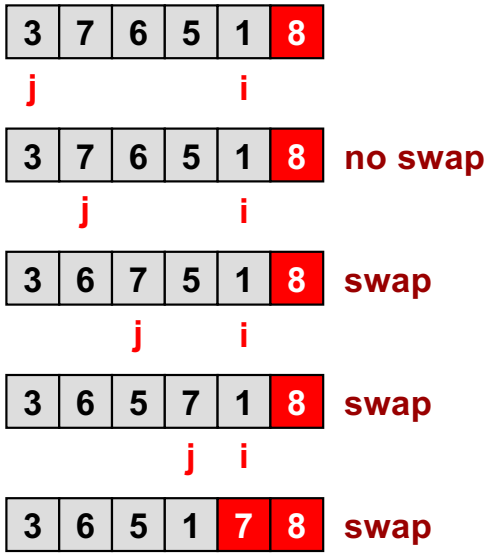
```
void bsort(int dat[], int len)
{
  int i ;
  for(i=len-1; i > 0; i--){
     for(j=0; j < i; j++){
        if(dat[j] > dat[j+1]) {
           swap(dat[j], dat[j+1])
  }  }  }
}
```

$O(n^2)$

Pass 1

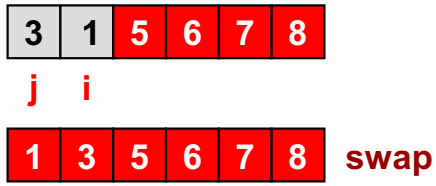


Pass 2



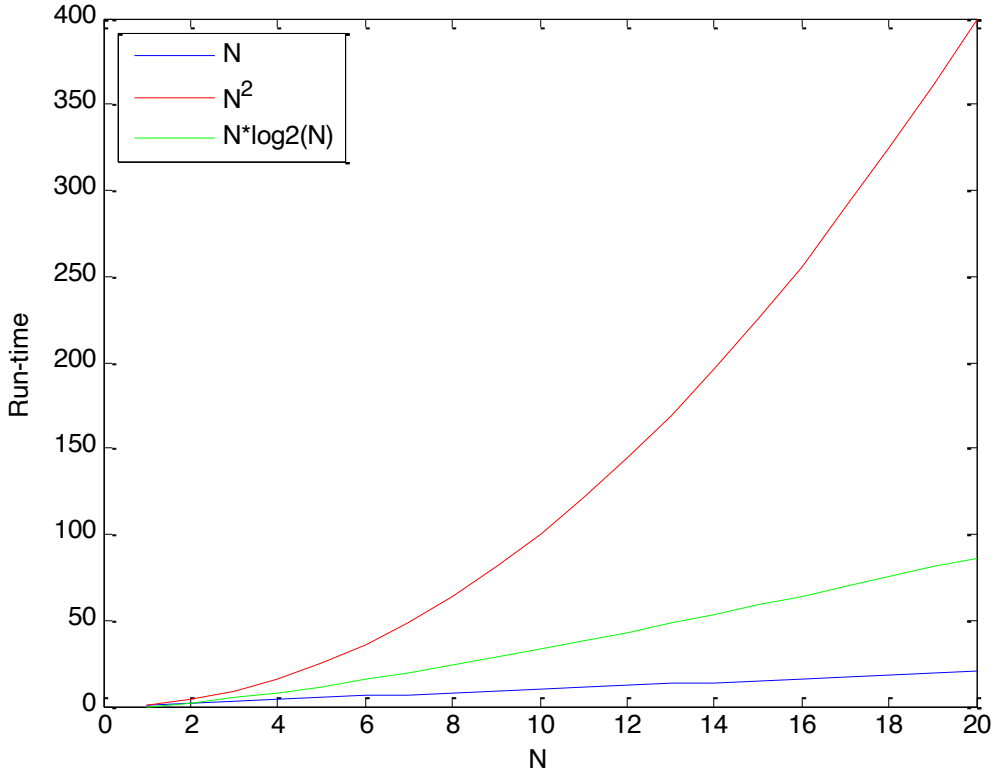
...

Pass n-2



Complexity of Sort Algorithms

- Bubble Sort
  - 2 Nested Loops
  - Execute outer loop n-1 times
  - For each outer loop iteration, inner loop runs i times.
  - Time complexity is proportional to:
    n-1 + n-2 + n-3 + … + 1 =
    $(n^2 + n)/2 = O(n^2)$
- Other sort algorithms can run in $O(n*\log_2 n)$



Importance of Time Complexity

- It makes the difference between efficient, possible, and impossible
- Many important problems currently can only be solved with exponential run-time algorithms (e.g. $O(2^n)$ time) [No known polynomial-time algorithm exists]
- Usually algorithms are only practical if they run in polynomial time

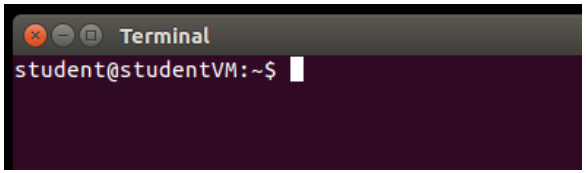
N	O(1)	$O(\log_2 n)$	O(n)	$O(n*\log_2 n)$	$O(n^2)$	$O(2^n)$
2	1	1	2	2	4	4
20	1	4.3	20	86.4	400	1,048,576
200	1	7.6	200	1,528.8	40,000	1.60694E+60
2000	1	11.0	2000	21,931.6	4,000,000	#NUM!

COMPILATION

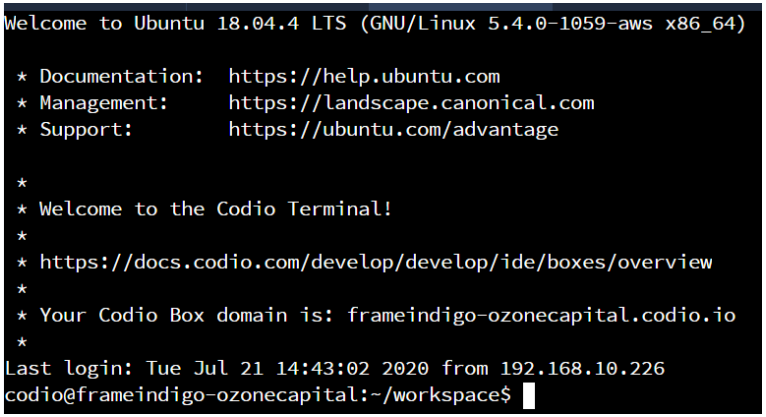
Using the Command Line

- While GUIs are nice, we often have more control when using the command line interface (i.e. the **terminal**)
- Linux (the OS used by Codio and in CS 103, 104, etc.) has a rich set of command line utilities
  - Mac & Windows do too, though Windows uses different names for the utilities
- By typing commands, we can
  - Navigate the file system (like you would with Explorer or Finder)
  - Start programs (vs. double-clicking an icon),
  - Copy, move, delete, rename files and folders
- Documentation often uses the symbols: `$` or `>` as a placeholder for the command prompt
  - Don't type it
  - If you see: `$ ./test`, you should just type `./test`

Terminal Icon



Linux Terminal View



Codio Terminal View

Compilers

- Several free and commercial compilers are available
  - g++
  - clang++
  - XCode
  - MS Visual Studio
- Several have "integrated" editors, debuggers and other tools and thus are called IDE's (Integrated Development Environments)

Compilation & Execution Process

```
#include <iostream>
using namespace std;
int main()
{
  int x = 42;
  cout << "Hello" << endl;
  cout << "x=" << x;
  return 0;
}
```

C++ file(s) (test.cpp)

C++ Standard & Other Libraries

g++

Compiler

```
1110 0010 0101 1001
0110 1011 0000 1100
0100 1101 0111 1111
1010 1100 0010 1011
0001 0110 0011 1000
```

Executable Binary Image ("test")

Load & Execute

Compiler options/flags:
- -g = Enable Debugging
- -Wall =Show all warnings
- -o <name> = Specify Output executable program name

Options (aka Flags) | Input file(s) (source) | Output executable name

```
$ g++ -g -Wall test.cpp -o test
```

Command prompt (don't type)

```
$ g++ -g -Wall test.cpp -o test
$ ./test
```

1. Edit & write code
2. Compile & fix compiler errors
3. Load & run the executable program

Compiling with g++

- Most basic usage
  - g++ *cpp_filenames*
  - Creates an executable using default name `a.out`
- Common Options
  - `-o` => Specifies output executable name (other than default a.out)
  - `-g` => Include info needed by debuggers like gdb, kdbg, etc.
  - `-Wall` => show all warnings
- Most common usage form:
  - **`$ g++ -g -Wall test.cpp -o test`**
- Run the program by preceding the executable name with `./` *(specifies local directory)*
  - Use whatever executable name you gave to g++ via the `-o` option

```
$ g++ test.cpp
# implicitly makes output
#  program name:  a.out
$ ./a.out
```

```
$ g++ -g -Wall test.cpp -o test
$ ./test
```

Dealing with Compile Errors

- **Compiler** errors are a **good kind of error** (or at least better than **runtime** errors)
  - The compiler GIVES you a **line number** and **error message** to help you
  - Runtime errors usually just produce the wrong output or crash, leaving no clues as to why

Tips for solving compiler errors

- Find the **line number**, look at your code on that line, and read the error message carefully
- **Do an Internet search of key words from the error message** (leave out any specific names in the error message) to see if there is an explanation online
- Deal with the **FIRST** error by scrolling to the top of the error messages
- If there is too much output, use ***I/O redirection*** (more to come)
  (e.g. `$ g++ -g prog.cpp -o ./prog >& errors.txt` and then view `errors.txt`)

An Example

```
1   #include <iostream>
2
3   int main(){
4       int a, b, c;
5       void* ptr;
6       cout << bc;
7       *ptr = a;
8       return "0";
9   }
```

```
redekopp@bytes:~/cs103/lecture/03-compile$ g++ -Wall errors.cpp
errors.cpp: In function ‘int main()’:
errors.cpp:6:5: error: ‘cout’ was not declared in this scope; did you mean ‘std::cout’?
    6 |     cout << bc;
      |     ^~~~
      |     std::cout
In file included from errors.cpp:1:
/usr/include/c++/9/iostream:61:18: note: ‘std::cout’ declared here
   61 |   extern ostream cout;  /// Linked to standard output
      |                  ^~~~
errors.cpp:6:13: error: ‘bc’ was not declared in this scope; did you mean ‘c’?
    6 |     cout << bc;
      |             ^~
      |             c
errors.cpp:7:6: error: ‘void*’ is not a pointer-to-object type
    7 |     *ptr = a;
      |      ^~~
errors.cpp:8:12: error: invalid conversion from ‘const char*’ to ‘int’ [-fpermissive]
    8 |     return "0";
      |            ^~~
      |            |
      |            const char*
errors.cpp:4:12: warning: unused variable ‘b’ [-Wunused-variable]
    4 |     int a, b, c;
      |            ^
errors.cpp:4:15: warning: unused variable ‘c’ [-Wunused-variable]
    4 |     int a, b, c;
      |               ^
```

MULTIFILE COMPILATION

Splitting over Multiple .cpp Files

- Most real-world software applications have their source code **split** over multiple files
- The code to the right is split over two files
  - main() in one file which declares an array
  - A function, sum(), in another file which sums the array
- We need both files to create a full program.

```
#include <iostream>
using namespace std;
  // no prototype for sum()
bool done = false;
int main()
{
  int array[3] = {4,5,6};
  // how does compiler know if
  // this is a valid func. call
  int val = sum(array, 3);
  cout << val << endl;
  return 0;
}
```

split-main.cpp

```
extern bool done;
int sum(int a[], int n)
{
  int s = 0;
  for(int i = 0; i < n; i++) {
    s += a[i];
  }
  done = true;
  return s;
}
```

split-sum.cpp

Splitting over Multiple .cpp Files

- **Fact 1:** The compiler only compiles **one** file at a time.
- We want functions defined in one file to be able to be called in another
  - How does the compiler know if the function exists?
  - It doesn't… it only checks when the linker runs (last step in compilation)
  - When translating a single file, it uses/requires a function prototype to verify & know the types of the argument(s) and return value
- Without a **prototype**, a compile-error is generated

```
$ g++ -g -Wall split-main.cpp
split-main.cpp: In function ‘int main()’:
split-main.cpp:8:13: error: ‘sum’ was not declared
in this scope
    8 |   int val = sum(array, 3);
      |             ^~~
```

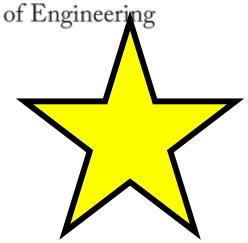
```
#include <iostream>
using namespace std;
  // no prototype for sum()
bool done = false;
int main()
{
  int array[3] = {4,5,6};
  // how does compiler know if
  // this is a valid func. call
  int val = sum(array, 3);
  cout << val << endl;
  return 0;
}
```

split-main.cpp

```
extern bool done;
int sum(int a[], int n)
{
  int s = 0;
  for(int i = 0; i < n; i++) {
    s += a[i];
  }
  done = true;
  return s;
}
```

split-sum.cpp

Compiling Multiple .cpp Files



- The compiler uses the prototype to check if the right number and type of arguments are being passed and what the return type is.
  - Compiler "trusts" that if there is a matching prototype, then somewhere and sometime later it will find the definition of that function in some other file (if not in this one)
- **Fact 2:** ALL source code files must be supplied in the compiler command for it to link and create an executable
  - If your source code is broken into 100 files, you need to compile all 100 together

```
$ g++ -g split-main.cpp split-sum.cpp -o split
$ ./split
15
```

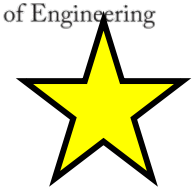
```
#include <iostream>
using namespace std;
int sum(int a[], int n);
bool done = false;
int main()
{
  int array[3] = {4,5,6};
  // compiler checks arg. types
  // and usage of return value
  // against the prototype
  int val = sum(array, 3);
  cout << val << endl;
  return 0;
}
```

split-main.cpp

```
extern bool done;
int sum(int a[], int n) {
  int s = 0;
  for(int i = 0; i < n; i++) {
    s += a[i];
  }
  return s;
}
```

split-sum.cpp

Undefined References



- Forgetting to list a source code file on the command line results in an "**undefined reference**" error!
  - These are quite common in CS 103 and CS 104 ***so please*** know their cause and what to look for to fix them
- We must provide ALL `.cpp` (or later, `.o`) files that have relevant code for our application.

Whenever you see an "undefined reference", you've either:

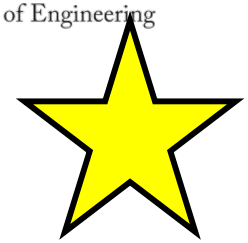
a) **(80% of the time) Forgot to list a source file on the g++ command line**
- **Verify all files are listed**

a) **(20% of the time) Have a typo in the prototype or function definition**
- **Prototyped** `int sum(char[], int)`
- **But defined** `int sum(int[], int)`

```
$ g++ -g split-main.cpp -o split
/tmp/ccDyvjR3.o: In function `main':
split-main.cpp:(.text+0x39): undefined
reference to `sum(int*, int)'
collect2: error: ld returned 1 exit status
```

```
$ g++ -g split-sum.cpp -o split
/usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/Scrt1.o: In
function `_start':
(.text+0x20): undefined reference to `main'
collect2: error: ld returned 1 exit status
```

Header Files



- Suppose `split-sum.cpp` defined multiple functions (i.e. more than just `sum()`)
- Further suppose other applications would like to use those functions.
- Rather than each application retyping the prototypes, place them in a header (`.h`) file and `#include` it into each source (`.cpp`) file that uses it
- Header files will also be used to define classes (objects) in the future…stay tuned!

```
// prototype only
int sum(int a[], int n);
```
split-sum.h

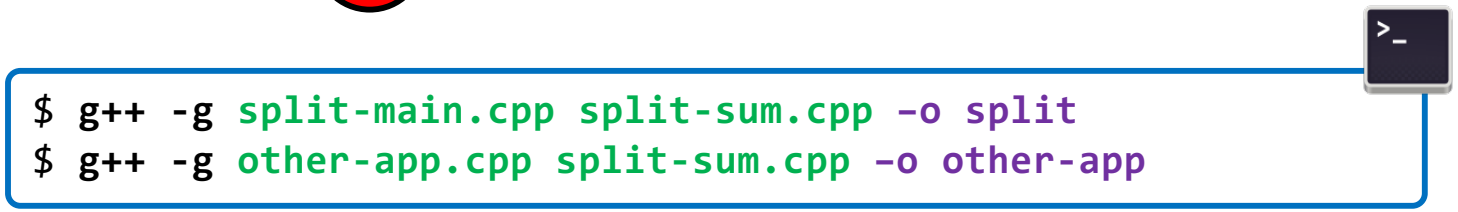
```
extern bool done;
int sum(int a[], int n) {
  // implementation
}
```
split-sum.cpp

```
#include <iostream>
#include "split-sum.h"
using namespace std;
bool done = false;
int main()
{
  int array[3] = {4,5,6};
  int val = sum(array, 3);
  cout << val << endl;
  return 0;
}
```
split-main.cpp

```
#include <iostream>
#include "split-sum.h"
using namespace std;
int main() {
  int vals[5];
  // ...
  cout << sum(vals, 3) << endl;
```
other-app.cpp

Header File Dos and Don'ts

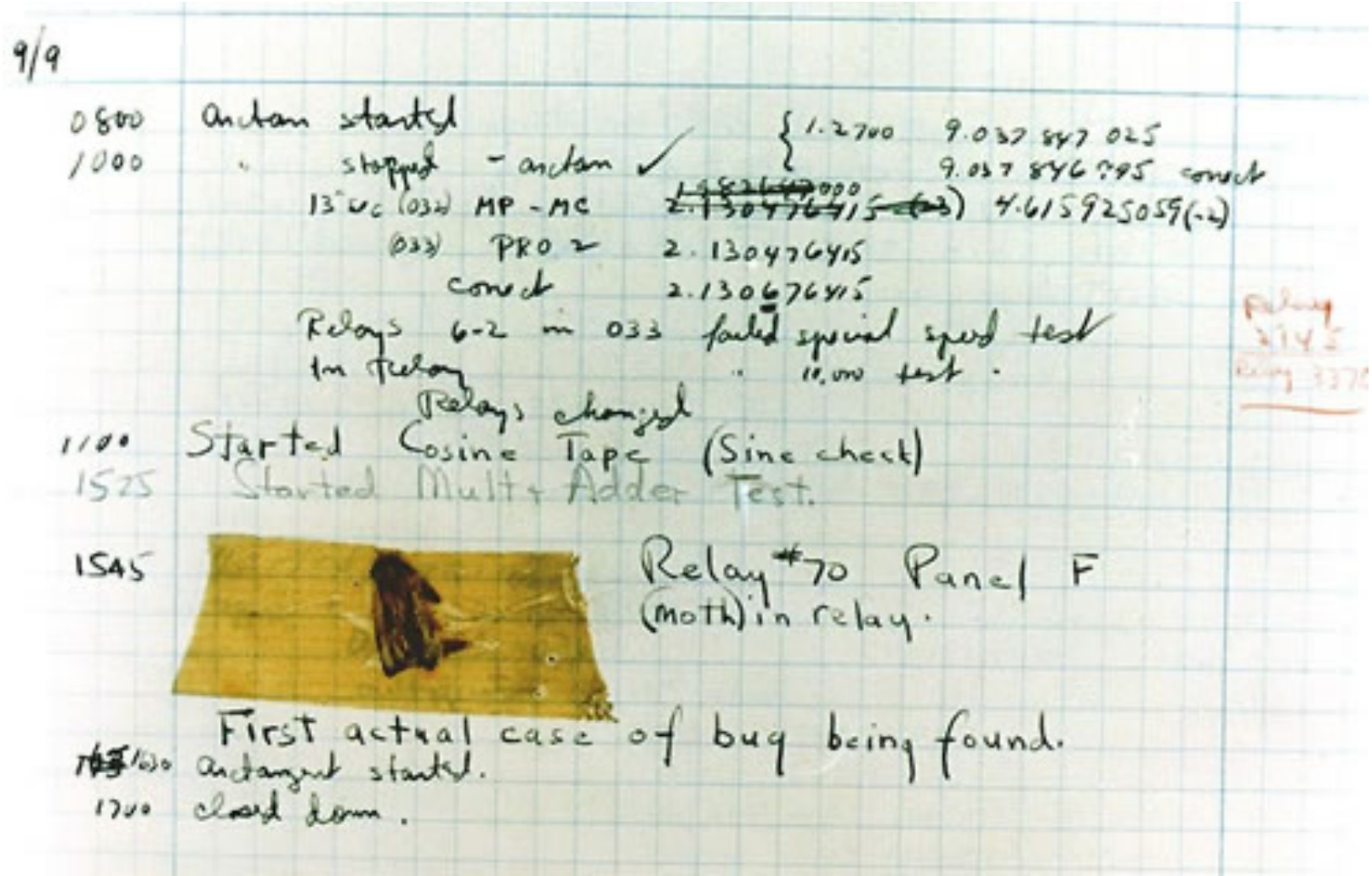
- NEVER compile .h files in a g++ command
  - Simply compile the .cpp files and the #include'd header files will be compiled as part of those
- DO #include header files in each source (.cpp) file that uses those functions (or, later, C++ classes)
- DO recompile any .cpp files that #include the header file WHEN the header file changes.



BASIC DEBUGGING

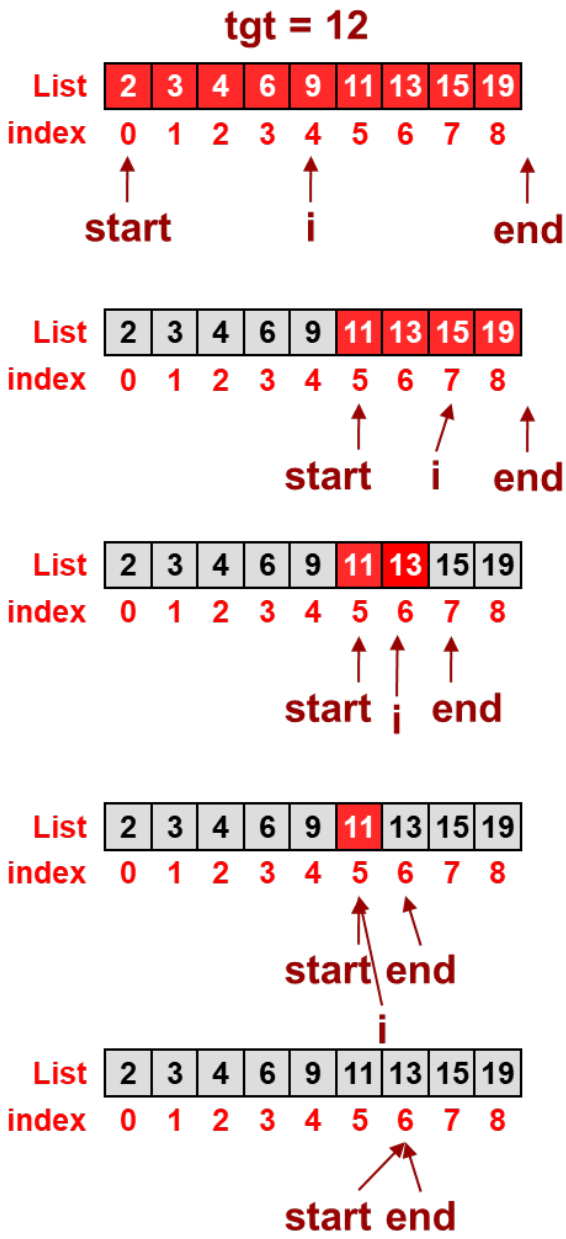
Bugs

- The original "bug"



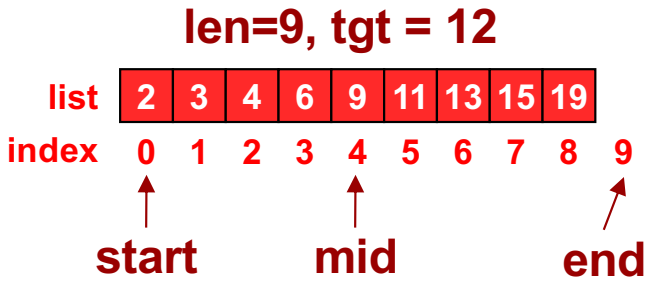
Step 1: Test Cases & Expected Outputs

- Do a few examples on paper and work out what the **correct (expected)** outputs should be (both intermediate results and final results)
  - You cannot effectively debug without an expectation of the ***right*** output so you know when the program is working
  - Example: For binary search, take the input array and target value and show how **start** and **end** will update on each iteration



Step 2: Hand-Tracing

- Use one of your input scenarios that is not working and trace the execution of your code by hand
  - Make a table of variables and walk the code line by line
  - Compare to the expected values from Step 1




```
#include <iostream>
using namespace std;

int bsearch(int list[], int len, int tgt)
{
   int start = 0, end = len;
   while(start != end) {
      int mid = (start + end) / 2;
      if(tgt == list[mid]) {
         return mid;
      }
      else if(tgt > list[mid]) {
         start = mid;
      }
      else {
         end = mid;
      }
   }
   return -1;
}
```

start	end	mid

Step 3: Print Statements / Narration

- Let the computer "trace" for you by using print statements
- Now that you know what to expect, the most common and easy way is to find the error is to add print statements that will "narrate" where you are and what the variable values are
- Be a **detective** by narrowing down where the error is
  - Put a print statement in each **'for', 'while', 'if'** or **'else'** block…this will make sure you are getting to the expected areas of your code
  - Then print variable values so you can see what data your program is producing

Example of Print Statements

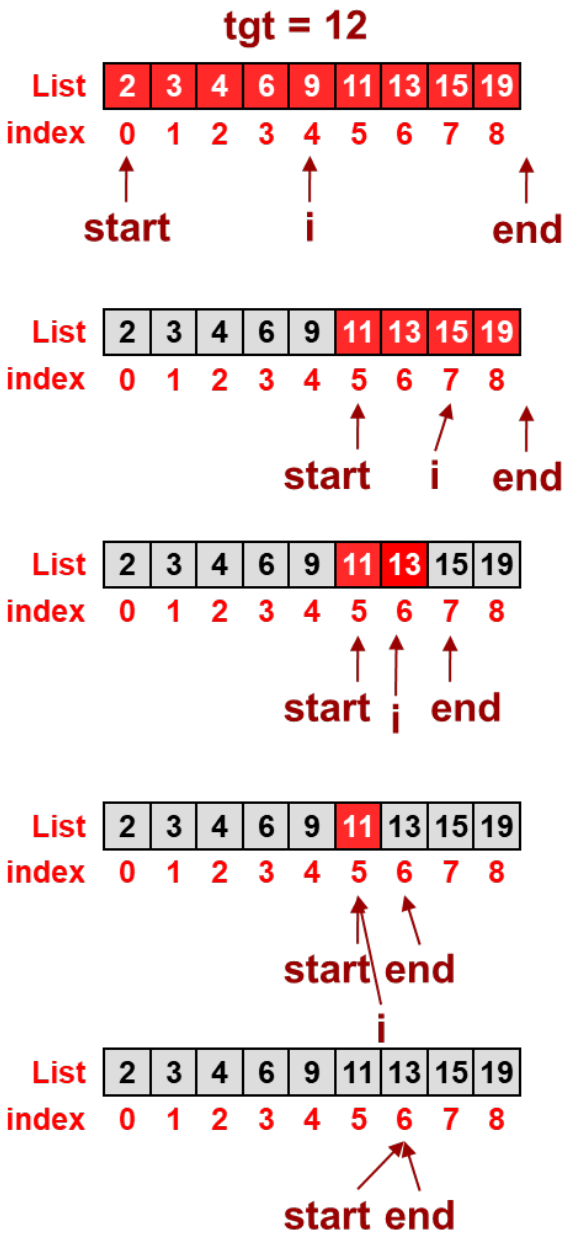
```
int bsearch(int list[], int len, int tgt)
{
   int start = 0, end = len;
   while(start != end) {
      int mid = (start + end) / 2;
      cout << "mid " << mid << endl;
      if(tgt == list[mid]) {
         return mid;
      }
      else if(tgt > list[mid]) {
         start = mid;
      }
      else {
         end = mid;
      }
   }
   return -1;
}
```

With novice print statements

```
int bsearch(int list[], int len, int tgt)
{
   cout << "Starting bsearch: len=" << len
        << " and target=" << tgt << endl;
   int start = 0, end = len;
   while(start != end) {
      cout << "New iter: start=" << start
           << " and end=" << end << endl;
      int mid = (start + end) / 2;
      cout << "\tChecking mid=" << mid << endl;
      cout << "\tdata=" << list[mid] << endl;
      if(tgt == list[mid])     {
         cout << "Found!" << endl;
         return mid;
      }
      else if(tgt > list[mid]) {
         cout << "\tLarger half" << endl;
         start = mid;
      }
      else {
         cout << "\tSmaller half" << endl;
         end = mid;
      }
   }
   return -1;
}
```

With (quality) print statements!

Which Debug Output is Most Helpful?




```
$ ./bsearch
mid 4
mid 6
mid 5
mid 5
mid 5
mid 5
mid 5
mid 5
...
```

With novice print statements

```
$ ./bsearch
Starting bsearch: len=9 and target=12
New iter: start=0 and end=9
   Checking mid=4
   data=9
   Larger half
New iter: start=4 and end=9
   Checking mid=6
   data=13
   Smaller half
New iter: start=4 and end=6
   Checking mid=5
   data=11
   Larger half
New iter: start=5 and end=6
   Checking mid=5
   data=11
   Larger half
New iter: start=5 and end=6
   Checking mid=5
   data=11
   Larger half
...
```

With (quality) print statements!

Fixed Code

```
int bsearch(int list[], int len, int tgt)
{
   cout << "Starting bsearch: len=" << len
        << " and target=" << tgt << endl;
   int start = 0, end = len;
   while(start != end) {
      cout << "New iter: start=" << start
           << " and end=" << end << endl;
      int mid = (start + end) / 2;
      cout << "Checking mid=" << mid << endl;
      cout << "   data=" << list[mid] << endl;
      if(tgt == list[mid])     {
         cout << "Found!" << endl;
         return mid;
      }
      else if(tgt > list[mid]) {
         cout << "Larger half" << endl;
         start = mid+1;
      }
      else {
         cout << "Smaller half" << endl;
         end = mid;
      }
   }
   return -1;
}
```

Fixed Code

Meta-Idea: Binary Search for Debugging

```
int main()
{
   ---
   ---
   ---
   ---
}
```

? Error! ?

```
$ g++ -g prog.cpp -o prog
$ ./prog
<Segmentation fault>
```

```
int main()
{
   ---          OK
   ---
   cout << "L50 x:" << x << endl;
   ---
   ---          Error!
}
```

```
$ g++ -g prog.cpp -o prog
$ ./prog
L50 x:42
<Segmentation fault>
```

```
int main()
{
   ---          OK
   ---
   cout << "L50 x:" << x << endl;
   ---    Error!
   cout << "L60 x:" << x << endl;
   ---
}
```

```
$ g++ -g prog.cpp -o prog
$ ./prog
L50 x:42
<Segmentation fault>
```

Think of debugging as performing a "binary search" for the bug/error. If the program crashes and you aren't sure where, or a variable has an unexpected value, add print statements at the mid-point and see if things are still good there. If not, the error is in the first half of the code. Otherwise, the error is in the second half of the code. Repeat the process on that half.

More Tips

1. Don't write the entire program all at once
2. Write a small portion, compile and test it
   - Write the code to get the input values, add some couts to print out what you got from the user, and make sure it is what you expect
   - Write a single loop and test it before doing nested loops
3. Once one part works, add another part and test it
4. Comment out later portions of the code and verify earlier parts work and then add your later code back in little-by-little to find where it stops/starts working

Alternative to Step 3

- Use a debugger tool
  - A program that allows you to see inside and slow down your program so you can understand what it is doing (vs. what you expect it to do).
  - More in lab!