

For Friday

- Finish Weiss, chapter 3. This should be largely review. If you're struggling with the C++ aspects, you may refer to Savitch, chapter 17.
- Homework
 - Weiss, chapter 2, exercises 7, 11(a,c,d), 12(a,c,d).

Algorithm Analysis

- What's the goal?

The Selection Problem

- What is it?

Measuring Program Performance

- Given: 2 programs to accomplish a task
- Both are correct
- Which do we pick?
- Two answers
 - Smallest
 - Fastest

Space Complexity

- Instruction Space
- Data Space
 - Constants and simple variables
 - Structures, arrays, dynamically allocated memory
- Environment Stack Space
 - Particularly relevant to recursive programs

Measuring Space Complexity

- Two parts
 - Fixed portion
 - Dynamic portion that depends on input
 - Dynamically allocated space
 - Recursive depth
- $S(P) = c + S_P(\text{instance characteristics})$

Time Complexity

- Actual time computation is difficult
- Estimates are typically good enough
- Two major estimates:
 - Operation Counts
 - Step Counts

Operation Counts

- Determine one or more operations which we believe are most important (contribute the most) to the time complexity
- We then count how many times the operation(s) occur(s).

Step Counts

- Determine the number of steps in the program in terms of some input characteristic
- Defining a “step”
 - Any computation unit that is **independent** of the input characteristics

Ways to Count Steps

- Can actually add code to a program to count the number of executed steps
 - Create a global variable count
 - Increment count for each step execution
- Step count table
 - List all statements in program
 - For each statement
 - determine how many steps it is worth
 - determine its frequency
 - multiply to produce total steps for the statement
 - Total the steps per statement

Notes about Step Counts

- Need to take into account best, worst, average cases
- Take all operations into account, but inexactly.
- Purposes of complexity analysis are
 - Compare two programs that compute the same function
 - Predict growth in run time as instance characteristics change

Comparing Program Complexities

- Suppose one program has a step count of $100n + 10$ and one has a step count of $3n + 3$. Which is faster?
- Now suppose that one program has a step count of $100n + 10$ and one has a step count of $90n + 9$. Which is faster?
- What if one is $3n^2 + 3$ and the other is $100n + 10$?

Comparing Programs

- If we have two programs with complexities of $c_1n^2 + c_2n$ and c_3n , respectively, we know that:

$c_1n^2 + c_2n > c_3n$ for all values of n
greater than some constant

- Breakeven point

Asymptotic Notation

- Asymptotic notation gives us a way to talk about the bounds on a program's complexity, in order to compare that program with others, without requiring that we deal in exact step counts or operation counts

Big Oh Notation

- $f(n) = O(g(n))$ iff positive constants c and n_0 exist such that $f(n) \leq cg(n)$ for all $n, n \geq n_0$.
- Provides an upper bound for function f .
- Standard functions used for g :

1	constant
$\log n$	logarithmic (base irrelevant)
n	linear
$n \log n$	$n \log n$
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

Finding Big Oh Bounds

- $3n + 2$
- $100n + 10$
- $10n^2 + 4n + 2$
- $6 * 2^n + n^2$
- 9
- 2045

Omega Notation (Ω)

- Lower bound analog of big oh notation.
- Definition:
 $f(n) = \Omega(g(n))$ iff positive constants c and n_0 exist such that $f(n) \geq cg(n)$ for all $n, n \leq n_0$.

Theta Notation (Θ)

- Theta notation can be used when a function can be bounded from above **and** below by the same function.

Little oh notation

- Little oh notation (o) applies when an upper bound is not also a lower bound.

Computing Running Times

- Rule 1: For loops
 - The running time is at most the running time of the statement inside the for loop (including tests) times the number of iterations.
- Rule 2: Nested loops
 - Analyze inside out.

- Rule 3: Consecutive statements
 - Add these (meaning the maximum is the one that counts)
- Rule 4: if/else
 - Running time of an if/else statement is no larger than the time of the test plus the larger of the running times of the bodies of the if and else.
- In general: work inside out.

Abstract Data Type

- A specification of a data type, including the operations of the data type
- Describes data items and operations in an **implementation-independent** way
- Can be implemented as a C++ class
- Could also be implemented as a set of variables and associated functions in C or another language

Linear List ADT

- AbstractDataType *LinearList*{
 instances (or data)
 ordered finite collection of zero or more elements
 operations
 Create()
 Destroy()
 IsEmpty()
 Length()
 Find(index) // returns an element
 Search(key) // returns an element
 DeleteAt(index)
 DeleteValue(key)
 Insert(index,element)
 Output()
}

Using an ADT

- ADTs are not directly usable
- They must be implemented
- Most ADTs can be implemented in more than one way
 - What would be different ways to represent the linear list ADT?

Linked Lists

- What's the basic concept?

What Is a Node?

- Two parts
 - Data
 - Pointer to the next one
- Why might you use a class instead of a struct to represent a node?

Creating Nodes

- We'll always create nodes dynamically.
- Note that we almost never actually work with a node itself; we use pointers to the nodes.
- Important:
 - ALWAYS initialize next to NULL when a node is created unless you are immediately assigning it another meaningful value.

C++ Details

- Deleting nodes . . .

Linked List Variations

- Doubly-linked lists
- Empty head node