Lecture 1

COMP 3760 – Winter 2020

Textbook: 1.1, 1.2, 1.3, 2.1

Instructor

- Tom Magliery
- Office: 5th floor DTC
- Best way to find me:
  - Email: tmagliery@bcit.ca

My history

- Undergrad: Math and Computer Science (Kansas)
- Grad: Computer Science (Illinois)
- Notable work: NCSA Mosaic, HTML, XML, XMetaL Author
- My home page is older than you are

Learning objectives

- Discuss the importance of algorithms in the problem-solving process.
- Choose the appropriate data structure or container for modeling a given problem.
- Describe, implement, and use common data structures and algorithms.
- Design and implement new algorithms using several common techniques, *e.g.* divide and conquer, greedy, dynamic programming, graph techniques, etc.
- Argue the correctness of their algorithms.
- Analyze pseudo-code using the Big-O notation.
- Deduce the complexity of a program by running different experiments.
- Discuss the computational efficiency of the principal algorithms for sorting, searching, and hashing.

Textbook

- Introduction to The Design and Analysis of Algorithms, 3rd Ed.
  - Author: Anany Levitin

- I expect you to read the sections of the textbook that relate to the material we cover in class

Pre-requisites

- Java (COMP 2526)
  - You need to be able to program in Java
  - You can use any IDE you want
- Discrete math (COMP 2121)
- Pseudocode
  - Remember Postel's Law: Be conservative in what you produce and liberal in what you accept.

Grading

Item	**%**	**Notes**
Lab assignments	25	Probably 11 total
Quizzes	20	Held in lab (not next week) Probably 8 total One lowest score dropped
Midterm	25	
Final exam	30	

Labs and quizzes

- Lab attendance is mandatory
- Assignments will relate to recent material
- You may *and should* discuss assignments but you *must* do your own work
  - Duplicate assignments = grade of 0
- Quizzes will happen during lab time
  - Quizzes are a significant portion of your grade
  - They are also great practice and study for the exams

Tips for success

- Practice!
- Keep up with the material
- Interrupt me ANY time in class
  - If something confuses you, don't naturally assume that I know what I'm talking about
  - I make mistkaes*!

* I swear I *literally, actually* made that typo. I did leave it here on purpose, though.

Math Review

(Ungraded) pre-test

- Please install the Kahoot app or go to www.kahoot.it

Review topics

- Logarithms
- Floor and Ceiling
- Counting
    - Permutations
    - Subsets
- Summations

Logarithms

- The main thing to know:
  - $log_b n = e$
  - just means: $b^e = n$

- So these two questions are the same:
  - $\log_2 16 = ?$
  - $16 = 2^?$

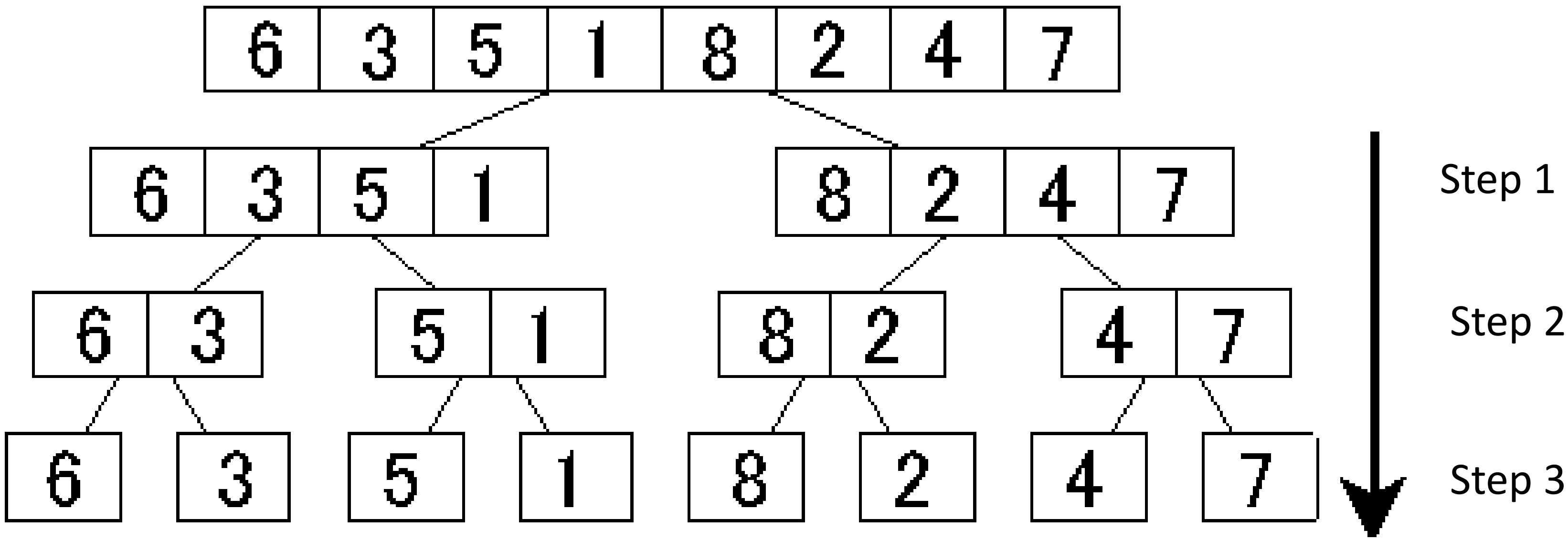
- In words:
  - “What is log base 2 of 16?”
  - “What power of 2 gives 16?”

When we’ll see logarithms

- The most common time:
  - Algorithms that divide a problem in half* at each iteration
  - How many steps does it take to get down to one?

- * or some other number of equal pieces

Example



$$\log_2 8 = 3$$

Floor and ceiling

- If *x* is not a whole number, these are useful:

  $\lceil x \rceil$ = The closest whole number *above* x
  (the ***ceiling*** of x)

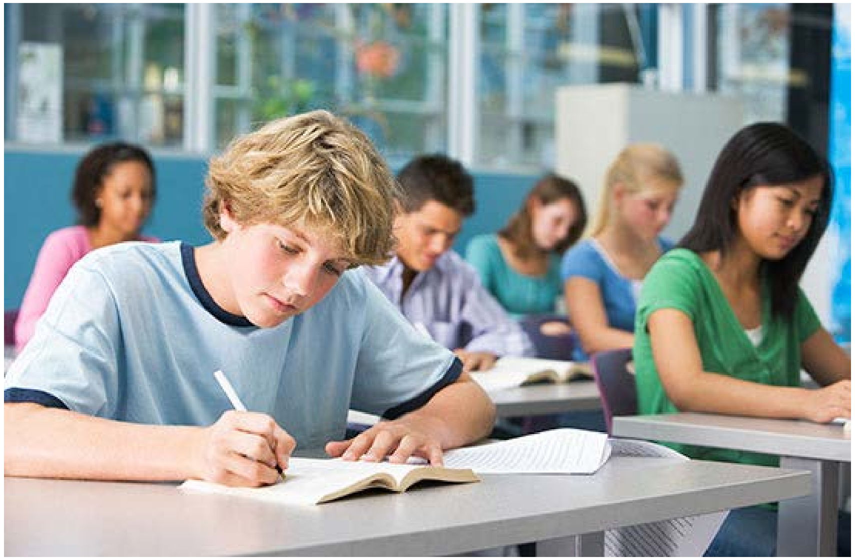
  $\lfloor x \rfloor$ = The closest whole number *below* x
  (the ***floor*** of x)

So: $\lceil \log 38 \rceil = 6$

$\lfloor \log 38 \rfloor = 5$

Counting

- Sometimes, we need to count things
- Example



In how many different arrangements could students sit on the chairs in a class?

Permutations

- A permutation is an arrangement in which order matters. ABC differs from BCA
- There are only two ways to arrange *2* items: AB, BA
- How many permutations are there on a collection of *3* items, A, B, C?
- ABC, ACB, BAC, BCA, CAB, CBA
- What if you have *n* items?

Counting trick

- A trick for many counting problems is:
  - Divide the problem into a series of independent choices
  - Count the options for each choice
  - Multiply those numbers together

Permutations

- A permutation is like placing n items A_1, …, A_n into a row of buckets:

- At each bucket, the choice of what goes in is independent
- n options for 1st bucket, n-1 options for 2nd, etc.

- Multiply together:

$$n * (n-1) * \ldots * 1 = n! \text{ permutations}$$

Subsets

- Given a set of 3 items {a, b, c}, how many different subsets can we make?

- Subsets are:

  {a, b, c},

  {a, b}, {b, c}, {a, c},

  {a}, {b}, {c},

  {}

Subsets

- Suppose you have n items: $A_1, \ldots, A_n$

- To construct a subset you have n items to consider:

  A_1 .... A_n

- Each item will be (independently) IN or OUT of a given subset:

  i/o .... i/o

- Multiply together:

  $2 * 2 * \ldots * 2$ (n times) $= 2^n$ subsets

Summations

- We use compact notation for summations

End by putting this value for i.

Start by putting this value for i.

$$\sum_{i=1}^{n} f(i)$$

f(i) can be any function… Think of it as a formula that has an i in it.

Put all values in between … and add them up.

- So this is really just a shorthand for:

$$f(1) + f(2) + f(3) + \ \dots + f(n)$$

Example

- Evaluate this expression:

$$\sum_{i=1}^{4}(2+i^2)$$

- Start with i=1, end with i=4…

$$(2+1^2)+(2+2^2)+(2+3^2)+(2+4^2)$$

- Now you just have numbers …

$$= 3 + 6 + 11 + 18$$

$$= 38 \; .$$

Sum of a constant

$$\sum_{i=1}^{n} C$$

- What it means:

$$\underbrace{C + C + \ldots + C}_{\text{(n times)}}$$

- So:

$$\sum_{i=1}^{n} C = nC$$

Another one

$$\sum_{i=1}^{n} n$$

- In this case *n* is also a constant!
- This means:

$$\underbrace{n + n + \ldots + n}_{\text{(n times)}}$$

  - So: $$\sum_{i=1}^{n} n = n * n = n^2$$

Changing the start and end

- We don't always go from 1 to n
- What is this sum?

$$\sum_{i=m}^{n} c = \underbrace{c + c + \cdots + c}_{(n-m+1)\ \text{times}}$$

$$\sum_{i=m}^{n} c = (n - m + 1) * c$$

Question

- What is this sum?

$$\sum_{i=0}^{n} 1$$

- Be careful … before we had i=1

$$\sum_{i=0}^{n} 1 = \underbrace{1 + 1 + \cdots + 1}_{(n-0+1)\text{ times}} = (n+1) * 1 = n + 1$$

Sums of sums

- Sometimes you have a sum with two parts added together:

$$\sum_{n=s}^{t} [f(n) + g(n)]$$

- You can just break it into two sums:

$$\sum_{n=s}^{t} f(n) + \sum_{n=s}^{t} g(n)$$

Constant rule

- You can move the constant in front for any sum

$$\sum_{n=s}^{t} C \cdot f(n) = C \cdot \sum_{n=s}^{t} f(n), \text{ where } C \text{ is a constant}$$

More summation rules

- There are many more summation rules in the appendix of your text.
- A few handy ones:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + ... + n = \frac{n(n+1)}{2} \ .$$

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + ... + n^2 = \frac{n(n+1)(2n+1)}{6} \ .$$

$$\sum_{i=1}^{n} i^3 = 1^3 + 2^3 + 3^3 + ... + n^3 = \frac{n^2(n+1)^2}{4} \ .$$

Practice problems

- Try to evaluate these:

$$\sum_{i=0}^{3}(5+\sqrt{4^i})$$

$$\sum_{i=1}^{100}(4+3i)$$

Solution 1

$$\sum_{i=0}^{3}(5+\sqrt{4^i}) = (5+\sqrt{4^0}) + (5+\sqrt{4^1}) + (5+\sqrt{4^2}) + (5+\sqrt{4^3})$$

$$= (5+\sqrt{1}) + (5+\sqrt{4}) + (5+\sqrt{16}) + (5+\sqrt{64})$$

$$= (5+1) + (5+2) + (5+4) + (5+8)$$

$$= 6 + 7 + 9 + 13$$

$$= 35\ .$$

Solution 2

$$\sum_{i=1}^{100}(4+3i)=\sum_{i=1}^{100}4+\sum_{i=1}^{100}3i$$

$$=\sum_{i=1}^{100}4+3\Big(\sum_{i=1}^{100}i\Big)$$

$$=4(100)+3\Big\{\frac{100(100+1)}{2}\Big\}$$

$$=400+15{,}150$$

$$=15{,}550\ .$$

Sum of summations

- We will often see things like this:

$$\sum_{j=1}^{i}\sum_{k=j}^{n} 1$$

- What does this mean?
  - It means you have a sum of sums
    - (NOT two sums multiplied)
  - To simplify it … you work from the inside out.

Sum of summations

- In this example:

$$\sum_{j=1}^{i}\sum_{k=j}^{n} 1 = \sum_{j=1}^{i}(n - j + 1)$$

- Now you can divide into three sums and solve:

$$\sum_{j=1}^{i} n - \sum_{j=1}^{i} j + \sum_{j=1}^{i} 1 = \mathrm{n} * \mathrm{i} - \frac{\mathrm{i} * (\mathrm{i} + 1)}{2} + \mathrm{i}$$

We will solve this kind of sum often (for a while) … so make sure you understand how to do it.

Algorithm efficiency

Section objectives

- State a definition of the term "algorithm"
- Explain the difference between "time efficiency" and "space efficiency"
- Determine the "basic operation" for a given algorithm represented in pseudocode
- Determine a formula for the number of times that any step in an algorithm will be performed, as a function of N (the size of the input to the algorithm)

Why do we care about algorithms?

Some reasons we care

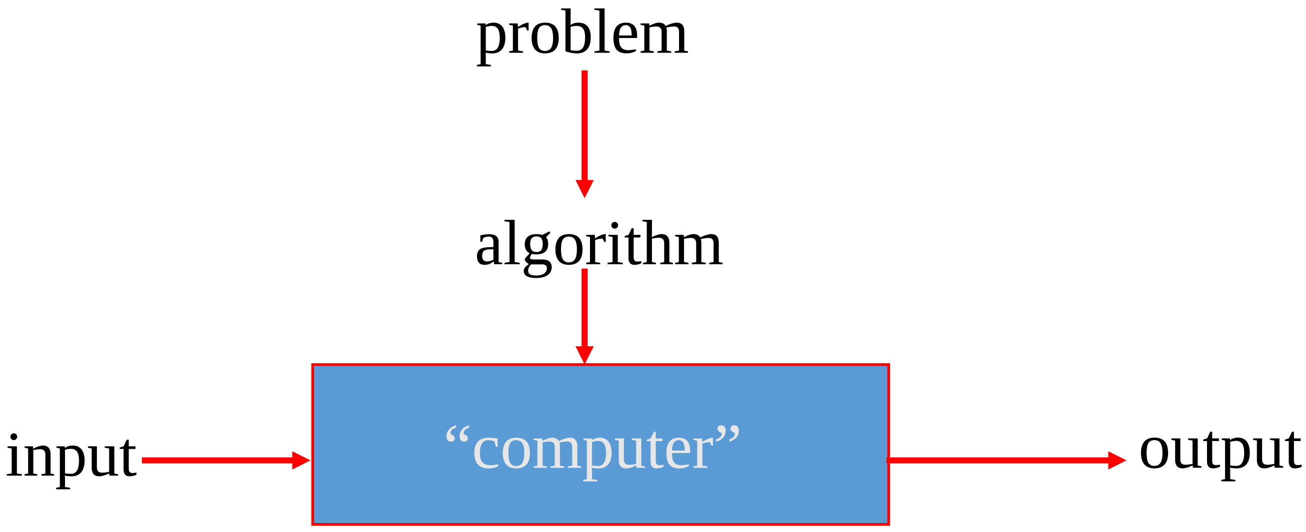
- Algorithms are at the core of computer programming
- There are many important, standard algorithms
- We want to design new algorithms and analyze their efficiency

What is an Algorithm?

- One definition:

  An algorithm is a sequence of ***unambiguous instructions*** *for solving a problem.*

  i.e: for obtaining a required output for any ***legitimate input*** *in a* ***finite amount of time***



Key points

- Each step is precise

- There can be more than one algorithm for the same problem

Example

- Here is a pseudocode algorithm:

```
Algo: find( A[0…n-1] )
    m ← A[0]
    for i ← 1 to n-1 do
        if A[i] > m
            m ← A[i]
    return m
```

- What does it do?

It finds the largest element of an array

Time Efficiency

- Is *find* a time-efficient algorithm?

- Seems good
  - To find the largest, you need to check each array element exactly once

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

Space Efficiency

- Is *find* a space-efficient algorithm? (amount of memory)

- Again… it seems reasonable
  - One temp variable introduced

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

Example

- What if you knew that the array A were already sorted?
- Is *find* still efficient?
- Could you think of a better algorithm?

```
Algo: find( A[0…n-1] )
   m ← A[0]
   for i ← 1 to n-1 do
       if A[i] > m
          m ← A[i]
   return m
```

Why do we care?

- Think about computing the n^{th} Fibonacci number:
  - 0, 1, 1, 2, 3, 5, 8, 13, …

First algorithm

```
Algo: fib( n )
   if n ≤ 1
      return n
   else
      return fib( n-1 ) + fib( n-2 )
```

Java implementation

```
public static int fib(int n)  {
   if (n<=1)
      return n;
   else
      return ( fib(n-1) + fib(n-2) );
}
```

Why do we care, Part 2

- Now look at a different algorithm

Second algorithm

```
Algo: fib2( n )
   F[0] ← 0; F[1] ← 1;
   for i ← 2 to n do
      F[i] ← F[i-1] + F[i-2]
   return F[n]
```

```
public static int fib2(int n) {

   int[] f = new int[n+1];

   f[0] = 0;
   f[1] = 1;
   for (int i=2; i<=n; i++)
       f[i] = f[i-1] + f[i-2];
   return f[n];
}
```

Difference

- First approach
  - Recursively calls the Fib function over and over again
- Second approach
  - Stores successive results so we don't have to re-compute them
- Before long the second approach is much, much faster

N	Fib1 (ms)	Fib2 (ms)
30	9	0
31	11	0
32	22	0
33	83	0
34	90	0
35	148	0
36	237	0
37	429	0
38	722	0
39	1105	0
40	1627	0

So?

- Fib is a basic example of why we care about algorithm efficiency

- A well thought out algorithm can run much faster

- There can be big variation in efficiency

How to determine efficiency

- Could do it experimentally
  - i.e. Write a bunch of implementations, see which one is fastest
- Problem?
  - Time consuming and expensive
  - It is not accurate

‣ Want to estimate efficiency before writing code

How to determine efficiency

- What we know:

1. Running time (efficiency) of an algorithm depends on the input size

2. The total execution time for any algorithm depends FIRST on the number of instructions executed
   - Execution time of specific instructions is secondary

Example

- Remember this algorithm:

```
1. Algo: find( A[0…n-1] )
2.    m ← A[0]
3.    for i ← 1 to n-1 do
4.       if A[i] > m
5.          m ← A[i]
6.    return m
```

for n=3

stmt	#times
1	0? 1?
2	1
3	2
4	2
5	2
6	1

- How many instructions are executed if n=3?

$$f(3) = 1 + 3*(3-1) + 1$$

Example

- What about n=8?

```
1. Algo: find( A[0…n-1] )
2.     m ← A[0]
3.     for i ← 1 to n-1 do
4.         if A[i] > m
5.             m ← A[i]
6.     return m
```

for n=8

stmt	#times
1	0
2	1
3	7
4	7
5	7
6	1

- $f(8) = 1 + 3*(8-1) + 1$
- For input of size n, the running time is

$$f(n) = 1 + 3*(n-1) + 1$$
$$= 3n - 1$$

Basic operations

- Which instruction in *find* gets executed the most?

```
1. Algo: find( A[0…n-1] )
2.      m ← A[0]
3.      for i ← 1 to n-1 do
4.          if A[i] > m
5.              m ← A[i]
6.      return m
```

stmt	(n=3) #times	(n=10) #times	(n=100) #times
1	0	0	0
2	1	1	1
3	2	9	99
4	2	9	99
5	2	9	99
6	1	1	1

- We define the basic operation of an algorithm as the statement that gets executed most frequently
  - Tiebreakers: deepest inside the loop; which one is more "expensive"; or maybe sometimes we don't care

Basic operations

This is the fundamental concept we use to analyze algorithmic efficiency:

count the number of basic operations
executed for an input of size n

- Using this idea, we would say the efficiency of *find* is:

$$f(n) = n-1$$

- We don't count instructions that are not basic operations

Example 1

- Consider this algorithm:

```
1. Mystery1(n)  // n > 0
2.   S ← 0
3.   for i ← 1 to n do
4.       S ← S + i * i
5.   return S
```

1. What does this algorithm do? Calculates: $1^2 + 2^2 + 3^2 + \ldots + n^2$
2. What is the basic operation? It's line 4
3. How many times is the basic operation executed for input size n?

How many times?

```
Mystery(n)  // n > 0
  S ← 0
  for i ← 1 to n do
      S ← S + i * i
  return S
```

- Basic operation is executed once each time through the loop
  - 1^{st} time: 1
  - 2^{nd} time: 1
  - …
  - n^{th} time: 1
- So you have a sum: $\sum_{i=1}^{n} 1$
- What does this equal?

1+ 1 + 1 … + 1 (n times)

= n

Example 2

- Consider this algorithm:

```
1. Mystery2(A[0..n-1][0..n-1])  // n > 0
2.  S ← 0
3.  for i ← 0 to n-1 do
4.     for j← 0 to n-1 do
5.          S ← S + A[i][j];
6.  return S
```

1. What does this algorithm do? Calculates sum of the elements in array A
2. What is the basic operation? Addition on line 5
3. How many times is the basic operation executed for input size n?

Example 2

```
Mystery2(A[0..n-1][0..n-1])  // n > 0
 S ← 0
 for i ← 0 to n-1 do
    for j← 0 to n-1 do
         S ← S + A[i][j];
 return S
```

- The outer loop
  - i goes from 0 to n-1
  - So we have:

$$\sum_{i=0}^{n-1} (\textit{whatever the inner loop is})$$

```
Mystery2(A[0..n-1][0..n-1])  // n > 0
 S ← 0
 for i ← 0 to n-1 do
    for j← 0 to n-1 do
         S ← S + A[i][j];
 return S
```

Example 2

- The inner loop:
  - j goes from 0 to n-1
  - At each iteration, we do one basic operation
  - So for the inner loop we have

$$\sum_{j=0}^{n-1} 1$$

  - We do this for each iteration of the outer loop

$$\sum_{i=0}^{n-1} \sum_{j=0}^{n-1} 1$$

Simplifying the sum

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1} 1$$

- The inner summation is:

$$\sum_{j=0}^{n-1} 1 = 1 + 1 + \ \dots + 1 = n$$

- So the outer summation is:

$$\sum_{i=0}^{n-1} n = n + n + \ \dots + n = n^2$$

Example 3

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```

- What does this algorithm do?
- What is the basic operation?
- How many times is the operation executed for input size n?

What does it do?

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```

5	**2**	4	6	1	3
2	5	**4**	6	1	3
2	4	5	**6**	1	3
2	4	5	6	**1**	3
1	2	4	5	6	**3**
1	2	3	4	5	6

Basic operation

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```

- Two options:
  - There are variable assignments and comparisons
  - Most people would say the basic operation is the key comparison *A[j]>v*
  - Why?
    - It is really the key thing being checked in each loop

Example 3 analysis

```
Loops(A[0..n-1])
 for i ← 1 to n-1 do
   v ← A[i]
   j ← i-1
   while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
   A[j+1] ← v
```

- Look at outer loop first

- There is a variable *i* getting incremented from *1* up to *n-1*

- So we have: $\sum_{i=1}^{n-1} (something)$

Example 3 analysis

```
Loops(A[0..n-1])
  for i ← 1 to n-1 do
    v ← A[i]
    j ← i-1
    while j≥0 and A[j]>v do
      A[j+1] ← A[j]
      j ← j-1
    A[j+1] ← v
```

- The inner loop:
  - j goes from i-1 down to 0
  - At each iteration, we do one basic operation
  - Mathematically, the number of steps is:

$$\sum_{j=0}^{i-1} 1$$

- We do this for each iteration of the outer loop
- So the total number of basic operations is:

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1$$

Simplifying the sum

- We know: $\sum_{j=0}^{i-1} 1 = i$

- So: $\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i$

- Which equals: $\frac{(n-1)n}{2}$

(we showed this earlier… and it is in appendix A)

Two main areas of interest in this course

- How to design algorithms

- How to analyze algorithm efficiency
  - Time/space efficiency

Algorithm design techniques

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Important problem types

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Numerical problems

Practice problems

- Chapter 1.1 page 8, question 5
- Chapter 1.2 page 18, question 9
- Chapter 1.3 page 23, question 1