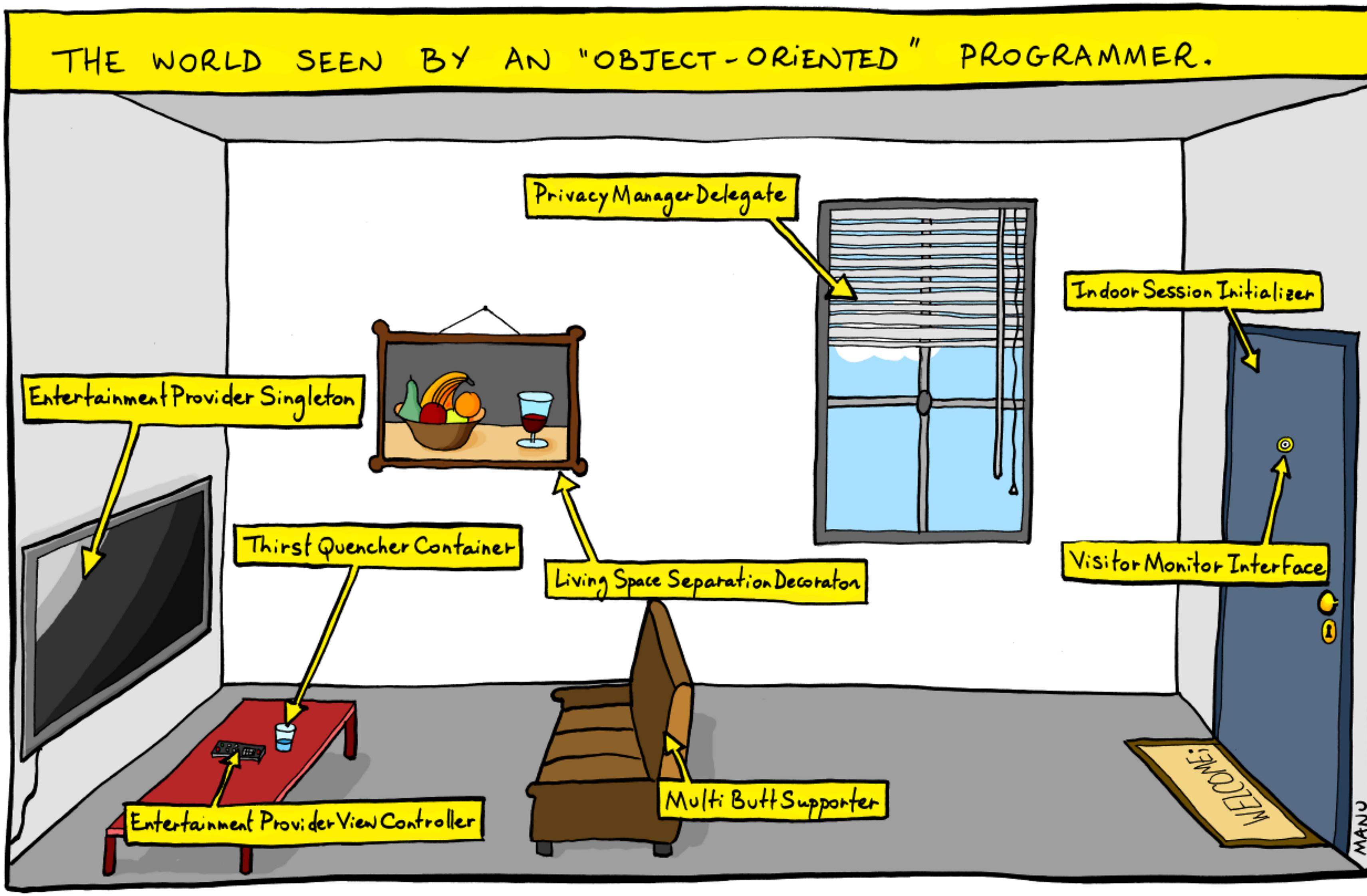
CSCI 104

Rafael Ferreira da Silva

rafsilva@isi.edu

Slides adapted from: Mark Redekopp and David Kempe

USC Viterbi School of Engineering



CLASSES (CONT.)

Composite Objects

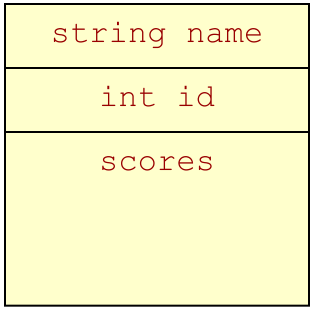
- Fun Fact: Memory for an object comes alive before the code for the constructor starts at the first curly brace '{'

```
#include <string>
#include <vector>
using namespace std;

struct Student
{
  string name;
  int id;
  vector<double> scores;
   // say I want 10 test scores per student

  Student() /* mem allocated here */
  {
    // Can I do this to init. members?
    name("Tommy Trojan");
    id = 12313;
    scores(10);
  }
};

int main()
{
  Student s1;
}
```



Composite Objects

- You cannot call constructors on data members once the constructor has started (i.e. passed the open curly '{')
  - So what can we do??? Use initialization lists!

```
#include <string>
#include <vector>
using namespace std;

struct Student
{
  string name;
  int id;
  vector<double> scores;
   // say I want 10 test scores per student

  Student() /* mem allocated here */
  {
    // Can I do this to init. members?
    name("Tommy Trojan");
    id = 12313;
    scores(10);
  }
};

int main()
{
  Student s1;
}
```

This would be "constructing" name twice. It's too late to do it in the {…}

string name
int id
scores

Constructor Initialization Lists

```
Student::Student()
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

If you write this…

```
Student::Student() :
    name(), id(), scores()
    // calls to default constructors
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

The compiler will still generate this.

- Though you do not see it, realize that the **default constructors** are implicitly called for each data member before entering the {…}
- You can then assign values but this is a **2-step** process

Constructor Initialization Lists

```
Student:: Student() /* mem allocated here */
{
  name("Tommy Trojan");
  id = 12313;
  scores(10);
}
```

You can't call member constructors in the {…}

```
Student::Student() :
    name("Tommy"), id(12313), scores(10)
{   }
```

You would have to call the member constructors in the initialization list context

- Rather than writing many assignment statements we can use a special initialization list technique for C++ constructors
  - Constructor(param_list) : member1(param/val), …, memberN(param/val) { … }
- We are really calling the respective constructors for each data member

Constructor Initialization Lists

```
Student::Student()
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

```
Student::Student() :
    name(), id(), scores()
    // calls to default constructors
{
  name = "Tommy Trojan";
  id = 12313
  scores.resize(10);
}
```

You can still assign data members in the {…}

But any member not in the initialization list will have its default constructor invoked before the {…}

- You can still assign values in the constructor but realize that the **default constructors** will have been called already
- So generally if you know what value you want to assign a data member it's **good practice** to do it in the initialization list

Member Functions

- Have access to all member variables of class
- **Use "const" keyword if it won't change member data**
- Normal member access uses dot (.) operator
- Pointer member access uses arrow (->) operator

```
class Item
{ int val;
 public:
  void foo();
  void bar() const;
};

void Item::foo() // not Foo()
{ val = 5; }

void Item::bar() const
{ }

int main()
{
  Item x;
  x.foo();
  Item *y = &x;
  (*y).bar();
  y->bar();  // equivalent
  return 0;
}
```

Exercises

- cpp/cs104/classes/const_members
- cpp/cs104/classes/const_members2
- cpp/cs104/classes/const_return

C++ Classes: Other Notes

- Classes are generally split across two files
  - ClassName.h – Contains interface description
  - ClassName.cpp – Contains implementation details
- Make sure you remember to prevent multiple inclusion errors with your header file by using #ifndef, #define, and #endif

```
#ifndef CLASSNAME_H
#define CLASSNAME_H
class ClassName { … };


#endif
```

```
#ifndef ITEM_H
#define ITEM_H

class Item
{ int val;
 public:
  void foo();
  void bar() const;
};

#endif
```

item.h

```
#include "item.h"

void Item::foo()
{ val = 5; }

void Item::bar() const
{ }
```

item.cpp

CONDITIONAL COMPILATION

Multiple Inclusion

- Often separate files may #include's of the same header file
- This may cause compiling errors when a duplicate declaration is encountered
  - See example
- Would like a way to include only once and if another attempt to include is encountered, ignore it

```
class string{
... };
```

string.h

```
#include "string.h"
class Widget{
 public:
  string s;
};
```

widget.h

```
#include "string.h"
#include "widget.h"
int main()
{ }
```

main.cpp

```
class string { // inc. from string.h
};

class string{ // inc. from widget.h
};
class Widget{
... }
int main()
{ }
```

main.cpp after preprocessing

Conditional Compiler Directives

- Compiler directives start with '#'
  - #define XXX
    - Sets a flag named XXX in the compiler
  - #ifdef, #ifndef XXX … #endif
    - Continue compiling code below until #endif, if XXX is (is not) defined
- Encapsulate header declarations inside a
  - #ifndef XX
    #define XX
    …
    #endif

```
#ifndef STRING_H
#define STRING_H
class string{

... };
#endif
```

String.h

```
#include "string.h"
class Widget{
 public:
  string s;
};
```

Character.h

```
#include "string.h"
#include "string.h"
```

main.cpp

```
class string{ // inc. from string.h
};

class Widget{ // inc. from widget.h

...
```

main.cpp after preprocessing

Conditional Compilation

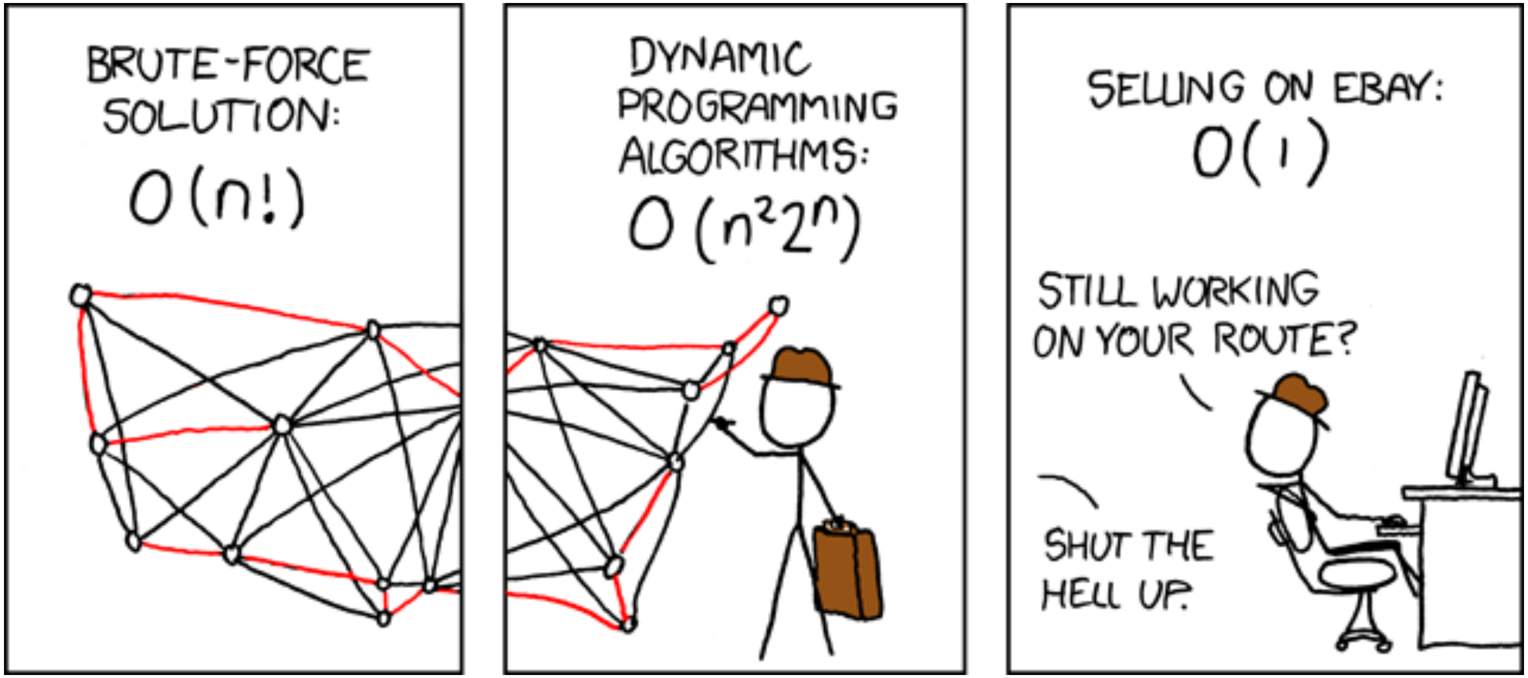
- Often used to compile additional DEBUG code
  - Place code that is only needed for debugging and that you would not want to execute in a release version
- Place code in an #ifdef XX...#endif bracket
- Compiler will only compile if a #define XX is found
- Can specify #define in:
  - source code
  - At compiler command line with (-Dxx) flag
    - g++ -o stuff –DDEGUG stuff.cpp

```
int main()
{
  int x, sum=0, data[10];
  ...
  for(int i=0; i < 10; i++){
    sum += data[i];
#ifdef DEBUG
    cout << "Current sum is ";
    cout << sum << endl;
#endif
  }

  cout << "Total sum is ";
  cout << sum << endl;
```

stuff.cpp

```
$ g++ -o stuff -DDEBUG stuff.cpp
```



XKCD #399

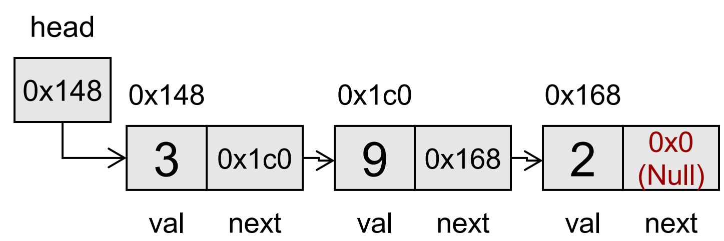
RUNTIME ANALYSIS

Runtime

- It is hard to compare the run time of an algorithm on actual hardware
  - Time may vary based on speed of the HW, etc.
    - The same program may take 1 sec. on your laptop but 0.5 second on a high performance server
- If we want to compare 2 algorithms that perform the same task we could try to count operations (regardless of how fast the operation can execute on given hardware)…
  - But what is an operation?
  - How many operations is: `i++` ?
  - `i++` actually requires grabbing the value of `i` from memory and bringing it to the processor, then adding 1, then putting it back in memory. Should that be 3 operations or 1?
  - Its painful to count 'exact' numbers of operations
- Big-O, Big-Ω, and Θ notation allows us to be more general (or "sloppy" as you may prefer)

Complexity Analysis

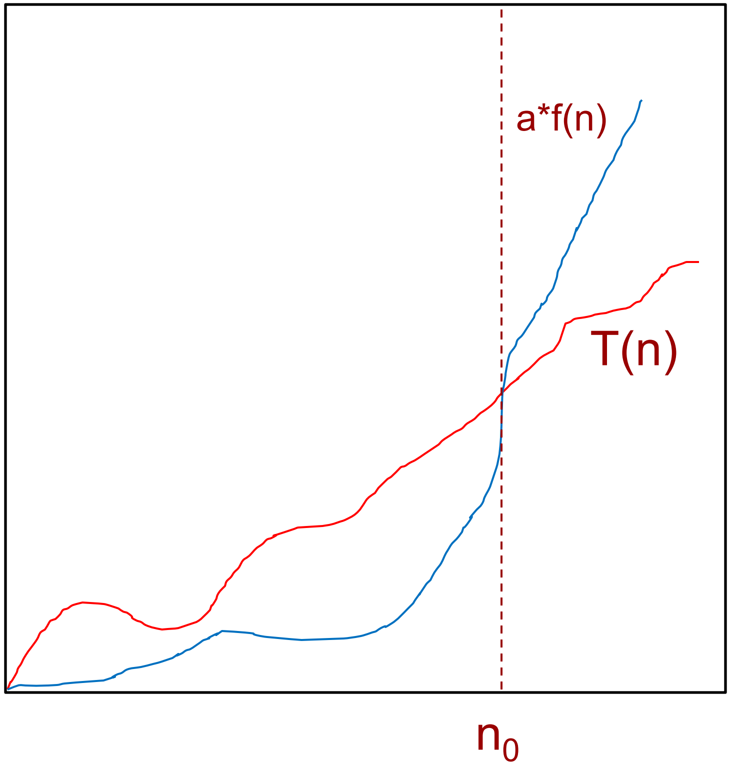
- To find upper or lower bounds on the complexity, we must consider the set of all possible inputs, I, of size, n
- Derive an expression, T(n), in terms of the input size, n, for the number of operations/steps that are required to solve the problem of a given input, i
  - Some algorithms depend on i and n
    - Find(3) in the list shown vs. Find(2)
  - Others just depend on n
    - Push_back / Append
- Which inputs though?
  - Best, worst, or "typical/average" case?
- We will always apply it to the "worst case"
  - That's usually what people care about



Note: Running time is not just based on an algorithm, BUT **algorithm + input data**

Big-O, Big-Ω

- T(n) is said to be O(f(n)) if…
  - T(n) < a*f(n) for n > n_0 (where a and n_0 are constants)
  - Essentially an upper-bound
  - We'll focus on big-O for the **worst case**
- T(n) is said to be Ω(f(n)) if…
  - T(n) > a*f(n) for n > n_0 (where a and n_0 are constants)
  - Essentially a lower-bound
- T(n) is said to be Θ(f(n)) if…
  - T(n) is both O(f(n)) AND Ω(f(n))



Worst Case and Big-Ω

- What's the lower bound on List::find(val)
  - Is it Ω(1) since we might find the given value on the first element?
  - Well it could be if we are finding a lower bound on the 'best case'
- Big-Ω does ***NOT*** have to be ***synonymous*** with 'best case'
  - Though many times it mistakenly is
- You can have:
  - Big-O for the best, average, worst cases
  - Big-Ω for the best, average, worst cases
  - Big-Θ for the best, average, worst cases

Worst Case and Big-Ω

- The key idea is an algorithm may perform differently for different input cases
  - Imagine an algorithm that processes an array of size n but depends on what data is in the array
- Big-O for the ***worst-case*** says ***ALL*** possible inputs are bound by O(f(n))
  - Every possible combination of data is at MOST bound by O(f(n))
- Big-Ω for the ***worst-case*** is attempting to establish a lower bound (at-least) for the worst case (the worst case is just one of the possible input scenarios)
  - If we look at the first data combination in the array and it takes n steps then we can say the algorithm is Ω(n).
  - Now we look at the next data combination in the array and the algorithm takes $n^{1.5}$. We can now say worst case is $\Omega(n^{1.5})$.
- To arrive at Ω(f(n)) for the ***worst-case*** requires you simply to find ***AN*** input case (i.e. the worst case) that requires ***at least*** f(n) steps
- Cost analogy…

Deriving T(n)

- Derive an expression, T(n), in terms of the input size for the number of operations/steps that are required to solve a problem
- If is true => 4
- Else if is true => 5
- Worst case => T(n) = 5

```
#include <iostream>

using namespace std;


int main()
{

  int i = 0;              1

  x = 5;                  1

  if(i < x){              1
     x--;                 1
  }
  else if(i > x){         1
     x += 2;              1
  }
  return 0;
}
```

Deriving T(n)

- Since loops repeat you have to take the sum of the steps that get executed over all iterations
- $T(n) =$
- $= \sum_{i=0}^{n-1} 5 = 5 * n$
- Or you can setup a relationship like:
- $T(n) = T(n-1) + 5$
- $= T(n-2) + 5 + 5$
- $= \sum_{i=0}^{n-1} 5 = 5 * n$
- $= \sum_{i=0}^{n-1} O(1) = O(n)$

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

int main()
{

  for(int i=0; i < N; i++){
    x = 5;
    if(i < x){
       x--;
    }
    else if(i > x){
       x += 2;
    }
  }
  return 0;
}
```

Common Summations

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \theta(n^2)$
  - This is called the arithmetic series
- $\sum_{i=1}^{n} \theta(i^p) = \theta(n^{p+1})$
  - This is a general form of the arithmetic series
- $\sum_{i=1}^{n} c^i = \frac{c^{n+1}-1}{c-1} = \theta(c^n)$
  - This is called the geometric series
- $\sum_{i=1}^{n} \frac{1}{i} = \theta(\log n)$
  - This is called the harmonic series

Skills You Should Gain

- To solve these running time problems try to break the problem into 2 parts:
- FIRST, setup the expression (or recurrence relationship) for the number of operations
- SECOND, solve
  - Unwind the recurrence relationship
  - Develop a series summation
  - Solve the series summation

Loops

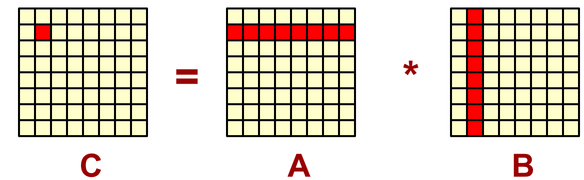
- Derive an expression, T(n), in terms of the input size for the number of operations/steps that are required to solve a problem
- $T(n) =$
- $= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \theta(1) = \sum_{i=0}^{n-1} \theta(n) = \Theta(n^2)$

```
#include <iostream>

using namespace std;
const int n = 256;
unsigned char image[n][n]
int main()
{
  for(int i=0; i < n; i++){
    for(int j=0; j < n; j++){
       image[i][j] = 0;
    }
  }
  return 0;
}
```

Matrix Multiply

- Derive an expression, T(n), in terms of the input size for the number of operations/steps that are required to solve a problem
- $T(n) =$
- $= \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} \theta(1) = \theta(n^3)$



Traditional Multiply

```
#include <iostream>
using namespace std;
const int n = 256;
int a[n][n], b[n][n], c[n][n];
int main()
{
  for(int i=0; i < n; i++){
    for(int j=0; j < n; j++){
      c[i][j] = 0;
      for(int k=0; k < n; k++){
        c[i][j] += a[i][k]*b[k][j];
      }
    }
  }
  return 0;
}
```

Sequential Loops

- Is this also n^3?
- No!
  - 3 for loops, but not nested
  - $O(n) + O(n) + O(n) = 3*O(n) = O(n)$

```
#include <iostream>

using namespace std;
const int n = 256;
unsigned char image[n][n]
int main()
{
  for(int i=0; i < n; i++){
    image[0][i] = 5;
  }
  for(int j=0; j < n; j++){
    image[1][j] = 5;
  }
  for(int k=0; k < n; k++){
    image[2][k] = 5;
  }
 return 0;
}
```

Counting Steps

- It may seem like you can just look for nested loops and then raise n to that power
  - 2 nested for loops => $O(n^2)$
- But be careful!!
- You have to count steps
  - Look at the update statement
  - Outer loop increments by 1 each time so it will iterate N times
  - Inner loop updates by dividing x in half each iteration?
  - After 1st iteration => x=n/2
  - After 2nd iteration => x=n/4
  - After 3rd iteration => x=n/8
  - Say kth iteration is last => $x = n/2^k = 1$
  - Solve for k
  - $k = \log_2(n)$ iterations
  - O(n*log(n))

```
#include <iostream>
using namespace std;
const int n = 256;

int main()
{
  for(int i=0; i < n; i++){
    int y=0;
    for(int x=n; x != 1; x=x/2){
       y++;
    }
    cout << y << endl;
  }
  return 0;
}
```

Analyze This

- Count the steps of this example?

```
for (int i = 0; i <= log2(n); i ++)
   for (int j=0; j < (int) pow(2,i); j++)
      cout << j;
```

- $\sum_{i=0}^{\lg(n)} \sum_{j=0}^{2^i} 1$
- $=\sum_{i=0}^{\lg(n)} 2^i$
- Use the geometric sum eqn.
- $=\sum_{i=0}^{n-1} a^i = \frac{1-a^n}{1-a}$
- So our answer is…
- $\frac{1-2^{\lg(n)+1}}{1-2} = \frac{1-2*n}{-1} = O(n)$

Another Example

- Count steps here…
  - Think about how many times if statement will evaluate true

```
for (int i = 0; i < n; i++)
{
   cout << "i: ";
   int m = sqrt(n);
   if( i % m == 0){
     for (int j=0; j < n; j++)
       cout << j << " ";
   }
   cout << endl;
}
```

- $T(n) = \sum_{i=0}^{n-1}(\theta(1) + O(n))$
- $T(n) =$

Another Example

- Count steps here…
  - Think about how many times if statement will evaluate true

```
for (int i = 0; i < n; i++)
{
   cout << "i: ";
   int m = sqrt(n);
   if( i % m == 0){
     for (int j=0; j < n; j++)
       cout << j << " ";
   }
   cout << endl;
}
```

- $T(n) = \sum_{i=0}^{n-1}(\theta(1) + O(n))$
- $T(n) = \sum_{i=0}^{n-1}\theta(1) + \sum_{k=1}^{\sqrt{n}}\sum_{j=1}^{n}\theta(1)$
- $T(n) = \theta(n) + \sum_{k=1}^{\sqrt{n}}\theta(n)$
- $T(n) = \theta(n) + \theta(n \cdot \sqrt{n})$
- $T(n) = \theta\left(n^{3/2}\right)$

What about Recursion

- Assume N items in the linked list
- T(n) = 1 + T(n-1)
- = 1 + 1 + T(n-2)
- = 1 + 1 + 1 + T(n-3)
- = n = O(n)

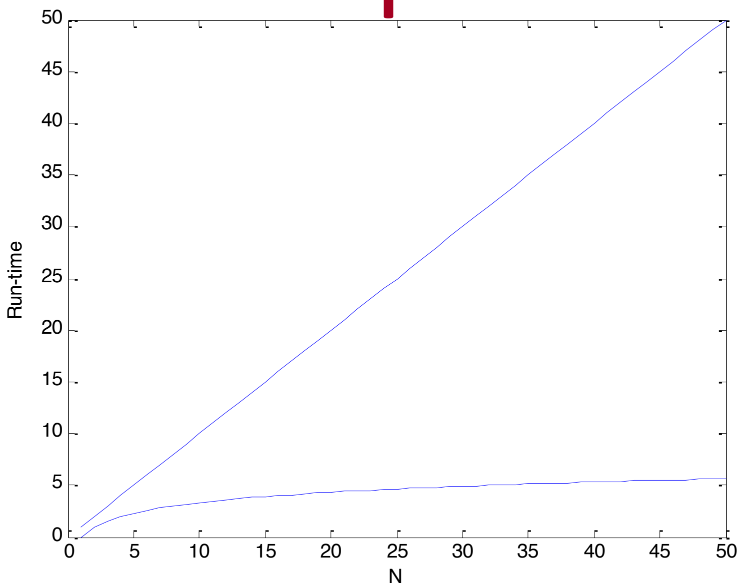
```
void print(Item* head)
{
   if(head==NULL) return;
   else {
     cout << head->val << endl;
     print(head->next);
   }
}
```

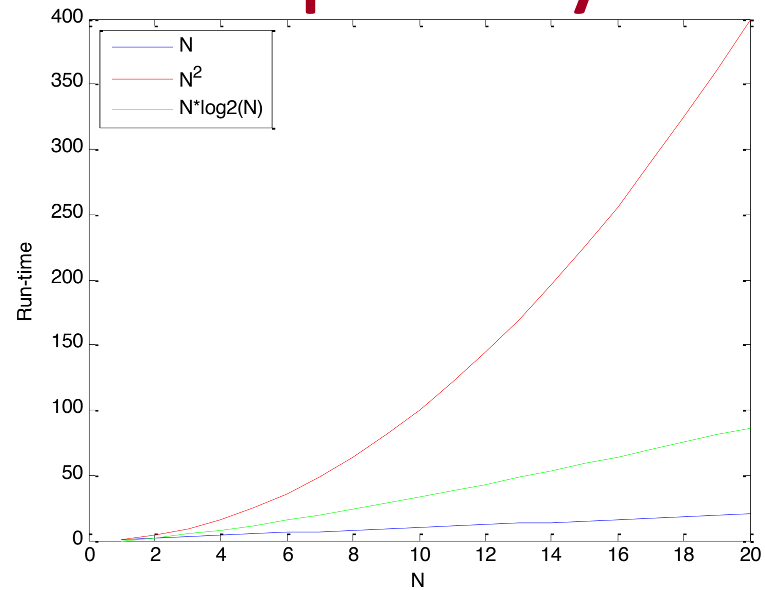
Binary Search

- Assume N items in the data array
- T(n) =
  - O(1) if base case
  - O(1) + T(n/2)
- $= 1 + T(n/2)$
- $= 1 + 1 + T(n/4)$
- $= k + T(n/2^k)$
- Stop when $2^k = n$
  - Implies $\log_2(n)$ recursions
- $O(\log_2(n))$

```
int bsearch(int data[],
            int start, int end,
            int target)
{
  if(end >= start)
    return -1;
  int mid = (start+end)/2;
  if(target == data[mid])
    return mid;
  else if(target < data[mid])
    return bsearch(data, start, mid,
                   target);
  else
    return bsearch(data, mid, end,
                   target);
}
```

Importance of Complexity





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!