PIC 10B
PROFESSOR POLLETT

FALL 2000
SEPTEMBER 29, 2000

Web page for class: http://www.math.ucla.edu/~cpollett
Prof. Pollett's OH: MWF 10-11am

1st Hw is already on the web. Due next Friday (Oct. 6) for lectures 1 and 2.

Plan for today (9/29/00, Friday): Go over syllabus and talk about classes in C++.

Major difference between 10A and 10B
In 10B we will talk about abstract data types which are useful in any language: lists, list processing, various kind of trees, sorting algorithms, hashing. We'll also be concerned with the efficiency of our algorithms. So we'll learn about tools to measure this such as O-notation.

Classes in C++ (Ch. 6.2 in Savitch)
A class is a C++ construct used to group several related variables and functions together (this grouping is called encapsulation).

Ex. ostream, ofstream, istream, ifstream are classes for input and output. An example object of type ostream is cout. cout has some member functions such as cout.setprecision(3);

How do define a class

```cpp
class EmployeeAccount //Name of class: notice that 1st letter of each word is capitalized.
{
    public:
        EmployeeAccount(); //Constructors say how to make this kind of object.
        EmployeeAccount(int amt);
        int getAmt();
    private:
        int amount; //Where data is stored.
};
```

To create an object of type EmployeeAccount in main or some other function we could do:

```cpp
int main()
{
    EmployeeAccount a, b(10); // a is created using 1st constructor: a() is not legal C++. b(10) is created using 2nd constructor.
    // to access info stored in a and b
    cout << a.getAmt() << endl; // prints 0
    cout << b.getAmt() << endl; // prints 10
}
```

Dot operator is used to access member variables or functions.

END OF LECTURE

OCTOBER 2, 2000

Reminder: HW 1 is due Friday.
Last lecture: talked about classes in C++
Today: We will talk about
1) How to define member functions
2) Difference between public and private
3) ADT's

```cpp
class EmployeeAccount
{
    public:
```
EmployeeAcct();
EmployeeAcct(int amt);
int getAmt();
private:
    int amount;
};

To define member functions:
EmployeeAcct::EmployeeAcct(); // This would appear outside ); ending the class.
{
    amount = 0;
} // Two colons are called scope resolution operator.
// which class we are defining member function of

EmployeeAcct::EmployeeAcct(int amt)
{
    amount = amt;
}

int EmployeeAcct::getAmt() // int is the return type
{
    return amount;
}

Now the class has been defined. An example of how to use the class:
int main()
{
    EmployeeAcct a, b(10); // a uses the 1st constructor
    cout << a.getAmt() << endl; // b uses the 2nd constructor
    return 0;
}

on screen: 0
10

Public vs Private
What's the difference?
Public variables/functions can be accessed by any other object or function.
(Ex) In main above we accessed a's getAmt function
In contrast...
Private variables can only be accessed within the scope of that class
(Ex) Above could access amount when defining EmployeeAcct constructors since within the scope of EmployeeAcct class. It would be a
criminal offense to do this in main since not in scope of EmployeeAcct class
Why bother having the distinction?
Allowing us to hide the way we're storing the data internally in case we need to change it. This is called Information Hiding.
More of this in a sec...

First some definitions:
A data type consists of a collection of values together with a set of base operations defined on these values.
(Ex) Data type int has +=, -=, *, /...
A data type is an abstract data type (ADT) if the programmer who uses the type does not have access to the details of how the values are
implemented.
(Ex) For data type int you do not know how code for + is actually written.

Classes allow us to define ADT's: what J.Programmer sees is public part.
Private part is how we actually do the implementation.
(Ex) class Date
    {
        public:
            Date(int day, int month, int yr, int calendarType); // For calendarType 0-Gregorian 1-Julian etc....
            Date(int day, int month, int yr); // calendarType is assumed to be Gregorian.
            int getDay(int calType);
            int getMonth(int calType);
            int getYear(int calType);
            int addDay();
            void output(int calType);
        private:
            // When we write this we have to make decisions about what is stored internally.
            // We assume information stored in Gregorian format.
```cpp
int d;
int m; // Or internally we could have stored just the total number of days since year zero.
int y;
}
```

Public part is called the interface. Private part is called the implementation.
The point is as long as we write our public member functions correctly outsiders never need to know how we stored things.

```cpp
Date::Date( )
{ 
  d=0;
  m=0;
  y=0;
}
```

// could define other member functions.

Above example does not show how to define ==, +, *, etc. for an ADT

suppose we had

```cpp
Date d1, d2(1, 1, 1), d3(1, 1, 2);
```

// then

```cpp
d1 = d2; // statement is legal and it sets date inside d1 to be same as d2
   // Every private member of d1 set to corresponding value in d2.
```

// if we did

```cpp
if (d1 == d2) cout << "hi there" << endl;
```

// statement is not legal at this point since we have not defined == operator

// for Date class

END OF LECTURE

---

OCTOBER 4, 2000

Reminder: HW1 due Friday. Make sure to get file name right.

Last Day - talked about defining member functions for classes. Talked a little about ADT's. Mentioned operator overloading.

Today - mainly talk about operator overloading. At the end of today we will start talking about separate compilation.

Last Day - we talked about class Date. We said we couldn't do

```cpp
Date d1, d2;
if (d1 == d2) cout << "hi\n";
```

yet since == wasn't defined for this class.

To define == for Date ...

```cpp
class Date
{ public:
   friend bool operator ==(const Date& d1, const Date& d2);
   // rest of class
};
```

friend - this keyword means that the function given after it is not a member function of the class, but is allowed access to any private data stored in objects of this class.

```cpp
const Date& d1 // & means call-by-reference i.e., get address at where a Date object is stored
   // means we will not change
```

Now outside of above class definition we'd write ...

```cpp
bool operator ==(const Date& d1, const Date& d2)
{ return (d1.d == d2.d &&
       d1.m == d2.m &&
       d1.y == d2.y);
}
```

allowed access to these private numbers variables since this was a friend function of Date.

```cpp
class Hour //class to store roughly the hour hand from a clock. Stores 0 to 11.
{ public:
   Hour();
};
```
Hour(int h);
int getHour();
friend bool operator ==(const Hour& h1, const Hour& h2);
friend Hour operator +(const Hour& h1, const Hour& h2);
// could do -, /, %, * this way.
friend ostream& operator <<(ostream& out, const Hour& h);
friend istream& operator >> (istream& in, const Hour & h);

private:
    int hour;
// Let's define member functions
Hour::Hour()  
    { hour = 0;  }
Hour::Hour(int h)
    { hour = h%12;  }  // If entered 23 then output 11th hour.
int Hour::getHour()
    { return hour;  }
bool operator ==(const Hour& h1, const Hour& h2)
    { return h1.hour == h2.hour;  }
Hour operator +(const Hour& h1, const Hour& h2)
    { return (h1.hour + h2.hour)%12;  }

// before defining << and >> let me explain a little.

When we have an expression like
    cout << "hi there" << "you";
It's like we'd written (cout << "hi there") << "you";
          prints hi there to the screen and returns the cout object.
Then (cout << "you"); is evaluated.
prints you to screen and outputs cout which since no more << operators does nothing.
This kind of evaluation explains prototype for << object.
ostream& operator <<(ostream& out, Hour& h)

usually address of cout usually cout thing to be printed
ostream& operators <<(ostream& h)
    { out << "Hour:"
      <<h.hour<<endl;
      return out;
    }
istream& operator >> (istream& in, const Hour & h)
    { int hour;
      in >> hour;
      h.hour = hour%12;
      return in;
    }

END OF LECTURE

OCTOBER 6, 2000

Last day - talked about operator overloading
Today - separate compilation of files and dynamic allocation of objects and arrays. If have time we will talk about the this pointer.

Separate Compilation
It is often the case that many people work on the same project. We would like to be able to split large programs into several files so that people working on a project can work independently.
Standard way to do this . .
Usually split a program based on class definitions.

Ex: Consider the Hour class we talked about last time. We'd put the interface in a file Hour.h and the implementation into a file Hour.cpp.
Anyone who wants to use the Hour class tells the compiler by putting \texttt{\#include "Hour.h"} at the start of their file.

### Why do we separate the code for the interface and implementation into two files?

Idea is that the user of class does not need to know the details of how class is implemented. User just needs to know member functions and what they're supposed to do. Also looking through a file with only the interface is easier to do than looking through both interface and implementation.

One problem with this setup.

We'd like Hour.h to be smart enough to "know" if it's already been loaded.

To make Hour.h smart use \	exttt{\#ifndef}

\begin{verbatim}
#define HOUR_H //preprocessor flag
#include "Hour.h" // here says flag has been defined
class Hour
{
    // our code
};
#endif // end \#ifndef HOUR_H
\end{verbatim}

### Dynamically Allocating Arrays and Objects

Why do we want to do this? Sometimes we would like to set size of an array at runtime or create an object at runtime.

To do this for objects.

Ex. Hour *h;  // creates a pointer to an object of type Hour, i.e., h can
              // store the memory address of where something of type
              // Hour lives.
               h = new Hour();  // creates at runtime an object of type Hour using default
                 // constructor

               //To use member functions of this object could do things like:
               cout<<*h->getHour()<<endl; // prints 0 to the screen

               //Same as (*h).getHour();
               //(*h) means object stored at address h.

               //When we're done using this object we free up memory by doing
delete h; // frees memory used by h. If h has a destructor call it.

To do this for arrays . . .

```cpp
int *myarr;
int size;
cin >> size;
myarr = new int[size]; // Creates an array of size many ints, returns its
// location to myarr.
// To set a value of this array
myarr[4] = 6; // Sets 5th element since we start counting at 0.

// this is equivalent to
*(myarr + 4) = 6; // Move over 4 ints from n and store 6.
// To get rid of this array when done . . .
delete[ ] myarr; // Says we're deleting an array
```

Like to start writing class to illustrate what we've talked about.

```cpp
#ifndef ARRAY_H
#define ARRAY_H
// Class for dynamic arrays
class Array
{
  public:
    Array(int s); // Allows us to create an array of size s.
    Array(); // Default array size is 10.
    Array(const Array& arr); // Copies arr into current object. Called
      // a copy constructor
    ~Array(); // Destructor frees up memory used by object
    Array& operator = (const Array& rhs); // Overloads = so that copies dynamically allocated
      // array properly. Notice this is a member function
      // Not a friend function

    // Other methods
  private:
    int *array; // Where the data is stored
    int max; // How big array is.
};
#endif
```

END OF LECTURE AND SET #1 ###################################################################################
OCTOBER 9, 2000

Reminders: HW 2 is on the web. Section 2C now meets Bunche 3150. HW 1 solutions are posted.

Last day-Separate compilation. Started on example of separate compilation that also was going to give an example of the pointer.

```c++
#ifndef ARRAY_H
#define ARRAY_H

class Array {
    // stuff
};
#endif

// stuff
```

Today-We will implement Array class. Talk about = overloading and the this pointer. We will also begin doing algorithm efficiency analysis.

//To implement Array make a tile called Array.cpp
#define include<iostream.h>
#include<stdio.h> //for exit() 
#include<cstdlib.h> //for NULL constant

Array::Array() {
    max=100; //default array size
    array=new int[max];
    if (array==NULL) { //ascertain if memory really got allocated
        cout "Error: not enough memory";
        exit (1);
    }
}

Array::Array(int s) {
    max=s;
    array=new int[max];
    if (array==NULL) { //same code as before
        cout << "Error: not enough memory";
        exit(1);
    }
}

Array::size( )
{ return max; }

Array::Array(const Array& arr) //copy constructor: argument is an array we are going to copy
{ max=arr.size( );
    array=new int[max];
    if (array == NULL) { //same code as before
        for (int i=0; i< max; i++) array[i] = arr.array[i];
    }
}

Array::~Array( ) //destructor is called when we delete an array.
{ delete[ ] array; } 

Array& Array::operator = (const Array& rhs) 
{ //this is a member function not a friend
    if (this == &rhs) "this" pointer is always a pointer to the current object. Left hand size of the operator is what this refers to.
    return *this; } //So in a=b this refers to object a.

// equals has a return value. That way, expressions like a=(b=c) makes sense. Also if (a=b) means set a equal to b then use the value of a in if
// condition.
Now we handle a case where two sides of equality not the same
else
    { delete[] array;
      max = rhs.size();
      array = new int[max];
      if (array == NULL)  {  //same code as before
        for (int i=0; i<max; i++) array[i]=rhs.array[i];
      }
    }
//other member functions code

Then in a new file myFile.cpp we could use above class.
#include <iostream.h>
#include "Array.h"
int main()
{
  Array a(10), b; c(a);
  //uses default constructor // uses copy constructor
c=a;... etc.
  // uses overloaded =. c has same value as a but not same address, thus else clause is evaluated because conditional expression of if clause is
  //false.
}

Can also do other operator overloading with member functions.
ex. class A
    { A& operator +(A& b);   // left hand side is passed using this.
    }

int main()
{
  A c,d;
  cout << c+d;
}
// There is a reason you might not want to do the above for +, but use friend function instead: Often times you would like to say what adding an
//integer constant to class means.
//for example, suppose you have a class RealNumber. Then if a is a real number, a+10 makes sense. But if overload + as a member function
// a+10 will make sense whereas 10+a won't.

END OF LECTURE  *************************************************************************************************************************************

OCTOBER 11, 2000

Made some corrections to HW2 yesterday. Prof. Pollett notified students by email. Send Prof. Pollett on email if you didn't receive one from
him.

Last Day - talked about the this pointer and overloading + operator
Today - mention some things about copy constructors and start talking about now to analyze algorithms.

Copy Constructors

Copy constructors can be explicitly called as in...
Array a, c(a); ← //copy constructor explicitly called for object c.
On the other hand sometimes copy constructors called automatically.
Ex. Void MyFunction(Array b) ← //call by value
    {  //some code
      int main()
      {  Array a(10)
          MyFunction(a);
          return 0;
      }
      //when b is created the copy constructor is used to copy a's value into b.
      //If you don't have a copy constructor then b is given the same reference as a. This can cause problems as the destructor b called when
      //MyFunction. So also a's will be called as same.
      //So important to have a copy constructor!
      Here's another situation where this comes up.
Array Function2( )
{
    Array b(10);
    return b;  //just before b gets deleted (since it's a local variable)
}
    //copy constructor called and the object created is what comes back from return.

Analyzing algorithms
ex. consider a situation similar to HW2. We want to have a dynamically sized array and have an operation + which allows us to add one integer to what's stored in the array.

ex. initial array
    index = 0  // last place where we've stored stuff
    +5
    index = 1

//if we want to now add 6 to this array our ADT will handle this by dynamically creating a new larger array by copying the contents of our old //array into new array, adding 6 to the new array and deleting old array.
//How big should the new array be?
//Let's say we just increased its size by 1, then

    index = 2  // we've copied 1 element to make this array from old array.
// if now we add 3 we would get

    index = 3  // we've copied 2 elements in going to this array from previous array.
//generalizing, if we add n elements then how many copies do we make totally?

# of copies = 1+2+3+...+(n-2)+(n-1) = (n-1)n/2 = n²
//roughly, n²ish many copies.
//Since # of copies determines the speed of the algorithm, we ask if we can do better?
//Answer: yes, we can do better if when we resize array we create an array twice as large as the old array.

ex. 1  //we copied 1 element in going to this array from old array.
    ↓ +5
    1 5
//if now we add 6

    index=3  //now only resize if index=size of array

    index=4  //no copies were made in this step.
//If we add in 2

    index=5
//so if we add n elements how many copies will we make?
//first how many times did we increase array size?
//Biggest the array gets is at most twice the number of elements inserted
//Array size is a power of 2. So number of times we double is the least x such that 2^x > n
//so 2^x=n if x=log₂n. So 2^x is greater than n where x=[log₂n]
//so the number of copies we do with this scheme is bounded by

1+2+2²+...+2^[log₂n]-1
//=(1+2+2²+...+2^[log₂n]-1)(2-1)
//=(2+2²+...+2^[log₂n]-1+2^[log₂n]+(-1-2-2²-...-2^[log₂n]-1)
//=2^[log₂n]-1 = n
//roughly n which is smaller than n²ish (the speed of the first algorithm)
//so second algorithm is better
//second algorithm is used in vector class in STL (standard library)
OCTOBER 13, 2000

Added a bonus problem to HW2. Bonuses are worth 1 point each and are added on top of your curved final grade.

Last day - We analyzed some algorithms for dynamically sized arrays.
Today - We will talk more about analyzing algorithms and we will talk about growth rates of various computer science functions.

How to figure how good is an algorithm.
One way...

Empirical analysis
Figure out algorithm behavior by running a program. Implementing it on various kinds of inputs (like software testing).

Three types of test data
1. Actual data - data similar to what will occur for your applications of the algorithm.
2. Random data - randomly generating inputs to your algorithm.
3. Perverse or adversarial data - data designed to make algorithm as inefficient as possible.
Problem with empirical analysis is that it can be very machine and operating system dependent.

Another approach is...

Comparative analysis
Compare different algorithms for same problem based on the number of times certain they use certain base operations. Ex. copying one element from an old array to a new one. As with Empirical Analysis, we can compare algorithms on worse-case or adversarial inputs or on average-case inputs. So to do this kind of comparative analysis, we need to understand how to compare growth rates of various functions of input.

Typical growth rates one sees in Computer Science
1 - cost of one base operation (constant time).

log N - (log in computer science is always base 2 since we're dealing with binary numbers). log N is roughly the length of N written in binary.
log 2 = 1, log 1 = 0, log 5 > 2 (since log 4 = 2)
Ex. Search for a number less than N using only questions of form, "Is number bigger than x?"
Number of steps to find number x log N.

N - linear growth
Ex. Scanning a file of length N takes N reads

N log N - it had to do binary search as in log N example N times.

N^2 - two nested for loops of same size

N^3 - three nested for loops of same size.

2^n - look at all possible strings of length N and do something with them.

Drawing a picture of how these functions look like:

Fractious don't exist in computer science world.

Ex. \([x]\) - round x down
[2.5] = 2

Ex. \([x]\) - round x up
[2.5] = 3

Ex. \([log_{10}(x)]\) = \([x]\) (length of x)

Estimating function growth rates with using integrals.

The Harmonic Series \(H_n = 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n\)

Value of \(H_n\) is the sum of the areas in the boxes.
So lower band is given by \( \int_{1}^{N} \frac{1}{x} \, dx = \log N \)

In fact, \( H_n = \ln N + \gamma + \frac{1}{12N} \) Where \( \gamma \) is Euler's gamma constant approximately 0.5772.

Ex. 2: Consider function \( N! = N(N-1) \ldots 1 \)

To get a lower bound note:

\[
\ln N! = \sum_{i=1}^{N} \ln i \geq \int_{1}^{N} \ln x \, dx = N \ln N - N
\]

Actually, Stirling's Formula says \( \ln N! = N \ln N - N + \ln \sqrt{2\pi n} \)

so \( N! \approx e^{\ln N} \sqrt{2\pi n} \)

**Big O Notation**

In computer science we usually only care about the asymptotic behavior of your function.

If runtime is \( 7N+3 \), the \( +3 \) doesn't really matter compared to \( 7N \) on large inputs.

Mainly it's the fastest growth rate term which most influences runtime of algorithm.

Big O notations give us a way to measure just this largest term.

**Definition:** \( O(f(N)) \) is the class of functions \( g \) such that there exists a constant \( K \) such that \( g(N) \leq K f(N) \) for all \( N \geq N_0 \) for \( N_0 \) fixed.

Ex. 1: \( 7 \in O(1) \)

Proof: Choose \( N_0 = 1, k = 7 \), then \( N \geq 1 \) \( 7 \leq 7.1 \)

Ex. 2: \( 7N + 3 \in O(N) \)

Proof: Choose \( N_0 = 1, k = 10 \), then \( 7N + 3 \leq 10N \) \( N \geq 1 \)

**END OF LECTURE AND SET #2**

---

5
OCTOBER 16, 2000

Remember HW2 is due Wed. Prof. Pollett will try to get students a practice midterm by Friday.

Last day-talked about math stuff
Today-will talk about big O-notation and recurrence relations.

Big O-notation
Definition: A function \( g(N) \) is in the class \( O(f(N)) \), i.e., \( g(N) \in O(f(N)) \), if there exists \( c_0, N_0 \) such that \( g(N) < c_0 f(N) \) for all \( N > N_0 \)

The point of big O-notation is it tries to capture the runtime of algorithms in a machine independent way. If we have an algorithm that performs \( N^2 + N \) operations, the \( N^2 \) part of the algorithm will be most important to improve the speed of.

ex. show \( N \in O(N^2) \)
Let's take \( c_0=1 \) and take \( N_0=3 \)
Claim for all \( N > N_0 \), \( N < c_0 (N^2 - 7) = N^2 - 7 \)
Proof (by induction):
Base case: \( N=4 \) then \( 4(16 - 7) = 9 \) is true
Inductive step: Assume \( N < N^2 - 7 \) to prove that \( N+1 < (N+1)^2 - 7 \)
\[ (N+1)^2 - 7 = N^2 + 2N + 1 - 7 \]
\[ > (N^2 - 7) + 1 \]
\[ > N+1 \] by the inductive hypothesis
Thus \( N < N^2 - 7 \) for all \( N > 3 \)

General facts about O-notation
1. \( O(1) \subseteq O(N) \)
2. \( O(N^k) \subseteq O(N^{k'}) \) if \( k' \geq k \)
3. \( O(f) \circ O(g) = O(f \circ g) \)
4. \( c \circ O(f) = O(f) \) where \( c \) is a constant
5. \( O(f) \circ O(g) = O(f \circ g) \)
6. \( O(f) \circ O(f) = 2O(f) \)

Ex. show
\[ x + x^2 + x^3 \in O(x^3) \]
\[ x + x^2 + x^3 \in O(x) + O(x^2) + O(x^3) \] by 2
\[ O(x) + O(x^2) + O(x^3) = O(x^3) + O(x^2) + O(x^3) \] by 2
\[ O(x^3) + O(x^2) + O(x^3) = 3O(x^3) \] by 6
\[ 3O(x^3) = O(x^3) \] by 4
Thus \( x + x^2 + x^3 \in O(x^3) \)

Ex. \( N^2 \not\in O(N) \)
Arguing by contradiction assume \( N^2 \in O(N) \). Then there exists \( C_0, N_0 \) such that \( N^2 \leq C_0 N \) for all \( N > N_0 \). Take \( N = \max(C_0, N_0) + 1 \). Then \( N^2 = (\max(C_0, N_0) + 1)^2 > C_0 \max(C_0, N_0) + 1 = C_0 N \) but that contradicts our assumption that \( N^2 \leq C_0 N \). Thus if we assume \( N^2 \in O(N) \) that leads to a contradiction. Hence \( N^2 \not\in O(N) \).

Other useful facts:
1. \( O(\log N) + O(N) \subseteq O(N) \)
( \( E \subset F \) if \( E \subset F \) is a proper subset of \( F \) i.e. \( E \subset F \), \( E \neq F \) )
2. \( O(n^k) \subseteq O(m^k) \) for \( k, m \) fixed

Ex. \( N^3 \in O(2^n) \) is true by 2 which says \( O(n^k) \subseteq O(m^k) \) for \( k=3, m=2 \).

How we can manipulate \( O \) in an expression (we can pretend it is like a constant).

Ex. show \( \frac{N}{N+O(1)} = 1+O\left(\frac{1}{N}\right) \)

\( N \)
\( N+O(1) \)
is a class of functions, \( 1+O\left(\frac{1}{N}\right) \) is a class of functions.

To show \( \frac{N}{N+O(1)} = 1+O\left(\frac{1}{N}\right) \) we have to show (a) and (b) are true

(a) \( \frac{N}{N+O(1)} \geq 1+O\left(\frac{1}{N}\right) \)
(b) \( \frac{N}{N+O(1)} \leq 1+O\left(\frac{1}{N}\right) \)

(a) is easy to see

\( \frac{N}{N+O(1)} = 1+O\left(\frac{1}{N}\right) \)

Claim: \( \frac{1}{O(N)} = O\left(\frac{1}{N}\right) \)

END OF LECTURE

OCTOBER 18, 2000

HW2 due today. HW3 on web. HW1 has been returned; 1 week period starting from 10/18/00 for regrades.

Recurrence Relations

We can often bound the number of operations for an algorithm on inputs of size \( N \) as a function of number of operations required on smaller instances.

Ex. Number of copies made for \( n \) inserts into a dynamic array.

If resized array one bigger than old array . . .

\( C_N = C_{N-1} + N \) recurrence relation

Recall \( C_N = N + N - 1 + C_{N-2} \)

\( N + N - 1 + N - 2 + \ldots + 3 + 2 + 1 = \frac{N(N+1)}{2} \)

\( = \frac{1}{2} N^2 + \frac{1}{2} N \)

\( = O(N^2) \)

Ex. If resized the array by doubling . . .

\( C_{2N} = C_{N} + N \) with \( C_1 = 1 \), for \( N \) a power of 2.

So for \( C_m \) between \( N \) and \( 2N \) \( C_m = C_{2N} \)

So \( C_m \leq C_{2N} + m/2 \)

\( C_m \leq C_{2N} + m/2 + m/4 \)

\( \leq C_{2N} + m/2 + m/4 + m/8 \)

To solve for \( C_m \) in closed form . . .

Let \( m = 2^k \), then

\( C_m = C_{2^k} \leq C_{2^{k+1}} = C_{2^{k+1}} \leq C_{2^{k+2}} \)

\( \leq C_{2^{k+3}} + 2 + \ldots + 1 = 2^k - 1 \)

\( = 0(2^k) \)

Another very common recurrence relation is the Fibonacci sequence defined as

\( F_0 = 0 \)

\( F_1 = 1 \)

\( F_n = F_{n-1} + F_{n-2} \quad n \geq 2 \)
To solve for $F_n$ in closed form, let $\alpha = \sum_{n=0}^{\infty} F_n x^n$. Then notice if

$$\alpha - \alpha x - \alpha x^2 = (F_0 + F_1 x + F_2 x^2 + \ldots) - (F_1 x + F_2 x^2 + \ldots) - (F_2 x^2 + \ldots)$$

$$= F_0 + (F_1 - F_0) x + 0 x^2 + 0 x^3 + \ldots$$

$$= x$$

So $\alpha(1 - x - x^2) = x \iff \alpha = \frac{x}{1 - x - x^2}$

Express $\frac{x}{1 - x - x^2}$ as a Taylor expansion and match $F_n$ with its coefficient you will get $F_n = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left( \frac{1 - \sqrt{5}}{2} \right)^n$.

Solving for the Fibonacci sequence in closed form will not be on the test. We’ll only consider linear recurrences in this class.

**Ex:** $C_n = 2C_{n-1} + N \iff$ only depends on one previous instance

This recurrence might come up with an algorithm like merge sort

Let $N = 2^n$

$$C_n = 2C_{n-1} + 2^n$$

$$= 2(2C_{n-2} + 2^{n-1}) + 2^n$$

$$= 4C_{n-2} + 2^n + 2^n$$

$$= 2^n + 2^n + \ldots + 2^n$$

n times

$$= n2^n$$

So $O((\log N)^2)$ since $n2^n = (\log N)^2$ because $N = 2^n$

**Ex:** $C_n = 2C_{n-1} + 1$

Let $N = 2^n$

$$C_n = 2C_{n-1} + 1$$

$$= C_{n-1} + 1 + 1$$

$$= 1 + 1 + 1 \ldots + 1$$

n times

$$= n$$

So $O(n) = O(\log N)$ because $N = 2^n$.

Let’s start analyzing Algorithms

**Ex:** Sequential Search

Input: $a[1] \ldots a[r]$, $1 < r$

Problem: Find $i$ such that $a[i] == value$. If it doesn’t exist return -1.

int search (int a[], int value, int i, int r)
{
    for (int i = 1; i < r; i++)
        if (value == a[i]) return i;
    return -1;
}

This algorithm’s runtime will vary depending on the array.

```
1 9 2
```

If search for 1 it takes only 1 check.

**Theorem:** Let $N$ be the size of array. If all locations equally likely, then sequential search examines on average $\frac{N + 1}{2}$ numbers on successful searches and $N$ numbers on unsuccessful searches.

**Proof:** Probability of looking at $i$ numbers is $\frac{1}{N}$ but we have to do $i$ steps in this case. Average number of steps to look is

$$\frac{1}{N} \left( \sum_{i=1}^{N} i \right) = \frac{1}{N} \left( \frac{N(N + 1)}{2} \right) = \frac{N + 1}{2} = O(N)$$

**END OF LECTURE**  

-------------------------------

**OCTOBER 20, 2000**

HW3 is up. Practice midterm is up. HW2 solutions available later today.
Last day - Talked about recurrence relations. Analyzed average runtime for linear search.
Today - Analyze another algorithm argc, argv.
Then start talking about the list ADT.

More of analyzing algorithms

Consider the problem that we would like to make an array a such that a[i] = 1 if i is a prime and 0 otherwise.
(A prime is an integer, whole only divisors are 1 and itself)
We will use so called sieve of Eratosthenes.
We'll produce an array for numbers less then...

```cpp
#include<iostream.h>
static const int N=1000 // N is how big a is.
int main()
{
    int i, a[N];
a[0] = 0;
a[1] = 0;
    for (int i=2; i<N; i++)
        a[i]=1; //assume all numbers <N are prime to start
    for (int i=2; i<N; i++)
    {
        if (a[i] == 1)
        {
            for(int j=i; j*i<N; j++)
                a[j*i]=0;
            //if a[i] is still a prime, set all of its multiples to be not
            // prime
        }
    }
    for (i=2; i<=N; i++)
    {
        if(a[i] == 1) cout <<i << "\n; //print out our primes
    }
    //end main
pass 1 2 3 ...
a[0] 0 0 0 ...
a[1] 0 0 0 ...
a[2] 1 1 1 ...
a[3] 1 1 1 ...
a[4] 1 0 0 ...
a[5] 1 1 1 ...
a[6] 1 0 0 ...
a[7] 1 1 1 ...
a[8] 1 0 0 ...
a[9] 1 1 0 ...
```

What's the runtime of this algorithm as a function of N?
1st line takes constant amount of time. O(1)Time
1st for loop we do = N assignments
   each assignment takes some constant amount of time NO(1) = O(N) time
2nd loop done N times and it's nested for loop does
   first N/2 assignments then N/3 assignments then N/S assignments...
So 2nd for loop takes N/2+N/3+N/5+N/7+1/11+...

\[
\sum_{p \text{ prime}}^{N}\frac{1}{p} \leq \sum_{i=1}^{N} \sum_{p \text{ prime}}^{\frac{N}{i}} \frac{1}{p} \leq \sum_{i=1}^{\log N} \sum_{p \text{ prime}}^{\frac{N}{i}} \frac{1}{p} \leq \sum_{i=1}^{\log N} \frac{\log N}{i} \leq \log N \log \log N
\]
\[ \int_1^{N+1} di = (N+1)\ln(N+1) \]

Final for loop prints to screen \(N\) times: \(\mathcal{O}(N) \in \mathcal{O}(N\log N)\)

\(\mathcal{O}(k) + \mathcal{O}(N) + \mathcal{O}(N\ln N) + \mathcal{O}(N) = \mathcal{O}(N\ln N)\)

argc, argv
Say we want to run our program from command line and we want to pass a value to our program that sizes an array. i.e., if under DOS or Unix

\[ \text{> myprog 1000 //supposed to tell your program that array size should be 1000.} \]

//myprog
int main(int argc, char* argv[])
{
    int i, N = atoi(argv[1]);  //atoi converts ascii string to integer
    int* a = new int[N];      //size array according to \(N\).
    myprog 1000 hi
    // argc = 3
    // argv[0] = "myprog"
    // argv[1] = "1000"
    // argv[2] = "hi"
}

Linked list:
Definition: A linked list is a set of items where each item is a part of a node that contains a link to a node.

END OF LECTURE AND SET #3
midterm on Friday in class. Remember to bring photo id.
Last day-started talking about linked lists. Talked about argc and argv
Today-Talk about linked lists

Typo correction
argc
↑
array of command line strings

number of command line strings

If command line was
myprog hi there
//argc = 3
//argv[0]=myprog
//argv[1]=hi
//argv[2]=there

Linked Lists
definition: a linked list is a set of items where each item is a part of node that contains a link to a node

(Recursive Data Structure)

Some types of lists
1) standard singly-linked list

2) circularly linked list

3) doubly linked lists

head has no previous node
each node has a pointer to a next and previous node

Advantage: can back up in list
Disadvantage: more memory to implement

Implementing Lists
In class we will follow book and implement using structures. For HW use classes.
Code for a Node

typedef int Item;  //now item type can be used (will mean same as int)
struct node { Item item, node *next};

    //why recursive as refers to self. Just an address so we don't need to know what a node is to define a node
typedef node *link;

ex. creating a list
link head = new node(0, NULL);
link x;

x = head;  //now x points to same node as head
x->item = 4;  //same as (*x) item = 4 which sets item in x to be 4.
   //would also change head's item
x->item = new node(3, NULL);

//can also do
x->next->item = 5;  //would change picture to...


List operations

suppose we want to go through a list performing some operation on each item in list. Could do this with the following code...
for(link t=x; t != NULL; t=t->next)
     visit (t->item);  //visit is code we wish to perform on each item.

How to reverse the order of a list
i.e. change
x-> 1 2 3

  to
x-> 3 2 1

Link reverse (link x)
{
    link t, y=x, r=NULL;
    while (y != NULL)
    {
        t = y->next;
        y->next = r;
        r = y;
        y = t;
    }
    return r;
}
Midterm is on Friday. Closed books, closed notes. You must have photo I.D. Prof. Pollett will take 2 points off midterm test score if you begin test early or end test late. No baseball caps during testing.

Practice Midterm

(1) Define the following
(a) Copy constructor - is a member function of a class that takes as an argument an object, say \( O \), of that class and produces a new object of this class, say \( O' \), such that \( O' \) has all its member data is the same as \( O \). Copy constructor can explicitly invoked by the programmer or is invoked when returning objects of this class by value or when instantiating call-by-value object of a class.

```cpp
MyClass fun() {
    MyClass a;
    return a; // copy constructor evoked
}

void fun2(MyClass 6)
```
(b) friend function - a nonmember function of class which has access to a class' private data. Not called by runtime system.
(c) destructor - a member function of an object used to free up what memory it uses when we get rid of that object.
- MyClass() is name of destructor for MyClass.
Can be called by runtime system. Ex: if object is a local variable and it goes out of scope of enclosing block.
- Ex.
  { MyClass a;
  }
  // a's destructor would be called.
(d) private variable - member variable of a class only-accessible by friend functions of class or by member of class.
(e) public variable - member variable of a class accessible from any function.

(2) Prove that N log N = O(N^2)
Let C_0 = 1, N_0 = 0. Need to show N log N ≤ C_0 N^2 for all N > N_0
Well N log N ≤ N^2 ⇔ log N ≤ N
⇔ 2^log N ≤ 2^N
⇔ N ≤ 2^N
We prove N ≤ 2^N by induction.
Basis step: N = 1 1 ≤ 2^1 is true
Inductive step: Assume N ≤ 2^N is true to prove (N+1) ≤ 2^(N+1)
2^(N+1) = 2 * 2^N = 2^N + 2^N ≥ N + 2^N by the inductive hypothesis.
≥ N + 1 since 2^N ≤ 1 ∀ N ≥ 0
Therefore since N ≤ 2^N for all N > 0, that implies N log N ≤ N. Thus if C_0 = 1, N_0 = 0
N log N ≤ C_0 N^2 for all N > N_0. Then N log N = O(N^2) by definition.

(3) Show N^2 ∈ O(N log N^2)
Proof (by contradiction): Assume N^2 ∈ O(N log N^2), then there exist N_0, C_0 such that N^2 ≤ C_0 N log N^2 for all N > N_0. N^2 ≤ C_0 N log N^2 ⇔ N log N ≤ C_0. Take N = max(N_0, (C_0 + 1)) to get a contradiction.

(4) Say how to use #ifdef, #endif in header files.
    #ifndef My_h // prevents the problem that header could be included
    #define My_h // twice in your implementation or user files.
    // your interface
    #endif

(5) Explain what the this pointer is and give and example of use.
The this pointer is a pointer to the current object of which we're running the member function of.
Ex.
int main()
{    MyClass a;
    a.fun();
}
void MyClass::fun()
{    this->myfun();
} // In this example "this" would be the memory address of a.
// this pointer is useful for overloading = .

(6) Consider
C_0 = 0
C_n = C_{n-1} + log N
Show C_n ∈ O(N log N)
Well C_n = (c_{n-1} + log(N-1)) + log N
    = ((C_{n-2} + log(N-2)) + log(N-1) + log N
Expand N times to get C_n completely expanded as a sum of N things.
Each of which is less than log N.
so get C_n ≤ log N + ... + log N
N times
≤ N log N for all N > 0
So C_n ∈ O(N log N)

END OF LECTURE AND SET #4
Today: Example of using linked-lists. Automatic type casting in C++. Templates

Application of Linked List

Sorting (Insertion Sort)

1st pass

\[3 \rightarrow 1 \rightarrow 7 \rightarrow 4\]

2nd pass

where we're at

\[3 \rightarrow 1 \rightarrow 7 \rightarrow 4\]

3rd pass

\[3 \rightarrow 1 \rightarrow 7 \rightarrow 4\]

4th pass

\[3 \rightarrow 1 \rightarrow 7 \rightarrow 4\]

//Keep inserting item under consideration
//where it fits in sorted list.

For n node list we do insertion operation O(n) times. Each insertion takes O(n) time. So this is an O(n^2) algorithm.

Code fragment for insertion sort

node heada(0, NULL);
link a=&heada, t=a;
for (int i=0; i<N; i++)
    t=(t->next=new node(rand%1000, NULL));
node headb(0, NULL);
link u, x;
link b=&headb;
for(t=a->next; t!=NULL; t=x)
    { u=t->next;
        for (x=b; x->next!=NULL; x=x->next)
Automatic Type Casting
Consider float a=0.2;
    a=a+1;       //This is an int the way computer handles this is 1st typecasts
    //to a float then adds
This is different from
float operator + (float a, int b)
Can do this style also in C++ to allow type casting from say an int to your Class A just add a constructor which just takes an int.
class A
{
    public:
        A(int i);       //used to convert from int to class A.
}

So if we wanted to we could just define operator + for objects of type A

Templates
Often the case that we would like to write code that works on a variety of kinds of inputs.
Consider
void swap (int& var1, int& var2)
{
    int temp;
    temp=var1;
    var1=var2;
    var2=temp;
}

If we changed int& var1 and int& var2 to type char& the same code would swap the two char's. Rather than have to re-enter same code for whatever type we want better to use templates
template<class TYPE>  //could be any C++ name
void swap (TYPE& var1, TYPE& var2)
{
    TYPE temp;
    temp=var1;
    var1=var2;
    var2=temp;
}

int main ( )
{
    char a= 'A', b='B';
    swap (a, b);       //TYPE in this case is clear
cout<<a<<b<<endl;    //BA
    int c=1, d=2;
    swap (c, d);
    cout<<c<<d<<endl;    //21
    return 0;
}

END OF LECTURE

NOVEMBER 1, 2000

HW3 due Friday. Midterm average = 15
Last day - started talking about templates.
Today - more templates
Selection Sort with templates

Set up: have an array to be sorted. Have an index to keep track of what's already been sorted. We pick smallest item yet to be sorted and swap it with element in index and advance index.

index (everything before index is sorted)

\[
\begin{array}{cccccc}
5 & 4 & 6 & 2 & 9 \\
\end{array}
\]

2nd pass

\[
\begin{array}{cccccc}
2 & 4 & 6 & 5 & 9 \\
\end{array}
\]

3rd pass

\[
\begin{array}{cccccc}
2 & 4 & 6 & 5 & 9 \\
\end{array}
\]

4th pass

\[
\begin{array}{cccccc}
2 & 4 & 5 & 6 & 9 \\
\end{array}
\]

done

We do \(O(N)\) swaps. Searching for smallest for a given swap look at \(O(N)\) elements. Sp total number of elements examined to perform all swaps \(O(N^2)\).

Above algorithm sorts "in-place" i.e. we do not need to create a temporary array to do sort.

Now let's write a generic sorting algorithm that works for any object for which \(<\) is defined.

Aside:

Definition: A generic function is one that can work on many types because defined using templates.

Will use swap values function from last lecture.

template <class T>
void sort(T a[], int size) // "size" is the size of array a.
{
    int indSmall;
    for (int i = 0; i < size-1; i++) // by time we get to size-1 array already sorted.
    {
        indSmall = IndexSmallest(a, i, size);
        swap_value(a[i], a[indSmall]);
    }
}

template <class T>
int IndexSmallest (const T a[], int start, int size)
{
    T min = a[start]; //smallest so far
    int indMin = start; //where smallest lives
    for (int i=start+1; i<size; i++)
    {
        if (a[i] < min)
        {
            min = a[i];
            indMin = i;
        }
    }
    return indMin;
} //end of IndexSmallest

int main()
{
    char c[5] = {'h', 'o', 'w', 'd', 'y'};
    sort(c, s);
    cout<<c[4]<<endl;
    return 0;
}

could also use sort for int's.

Templates with classes

Ex template <class T> //can have more than one generic type if so
/ separate by comma

class StoreObj //stores one object of whatever type we want
Stor€Obj CI obl);
T getObj ( );
private:
    T object;
};

//To write member functions
template <class T>
Stor€Obj::Stor€Obj (T obl)
{ object=obl; }

template <class T>
T Stor€Obj::getobj ( )
{ return object; }

//To use this class
Ex 1 Storeobj<int> StoreInt(3); //<int> says T is int
cout<<StoreInt.getobj ( )<<endl; //prints 3 to screen
Ex 2 Storeobj<char> storeChar('a');
cout<<storeChar.getobj ( )<<endl; //prints an a to screen

END OF LECTURE

NOVEMBER 3, 2000

HW4 will be up later today

Last day - templates with functions and with classes
Today - Stacks (will use templates)

Stacks (since it's short, here's the interface...)

template <class Item>
class Stack
{ public:
    Stack(int s); //initializes the stack array to be size s.
    bool empty() const; //const means we won't modify private data.
    void push(Item item);
    Item pop ( );

    private:
    Item *array;
    int maxSize, curSize;
};

//before look at code to implement a stack let's see how it's used/what's
//it good for.
Stack <int> s(10); //creates a stack of into of size 10.
push(5);
push(4);
push(3);

cout << pop ( );
cout << pop ( );
cout << pop ( ) << endl;

2nd pop return 4

3rd pop returns and print

4 returned and printed
Where are stacks used?
Any language after Algol uses runtime stack for function calls.

```c
int main() {
    int a = 4, b = 5, c = 6
    myfun(); \rightarrow we push
}
void myfun() {
    int e = 6, t = 10;
    myfun2(); \rightarrow after this call
}
```

So after myfun2 ends pop e, f off stack to get their values and stack looks like

- e would be set to 6 again, f would be set to 7.

One advantage is storing local variables in the stack during function calls is that allows functions to call themselves

```c
int factorial(int n) {
    if (n == 0) return 1;
    return n*factorial(n-1);
}
```

(2) Used by compilers in syntax checking
Consider a language with the following pairs of tag `for` `endfor`
The following code would be legal

```c
for
    for
        for
            for
                for
                    //but this would be illegal
    for
        for
    endfor
endfor
endfor
```

We could keep track of whether things nested correctly by whenever we see a for we push a symbol on the stack. And whenever we see an endfor insist we can pop an item from stack. If can't pop then give an error.

(3) HW3
To read number push item onto stack
if had 1 2 3 would push 3 on stack
if second number was 456 on another stack would have 6
5
4

Now to odd, pop digit from top of each stack add them and any carry push result onto a third stack.
Now to point out sum keep popping till third stack empty each time printing a digit.

(4) reverse polish expression evaluation
5 4 + is same as 5+4
reverse polish notation
3 5 4 + - is 3-(5+y)
3 5 - 4+ is (3-5)+y
To evaluate such an expression with stacks
(1) Whenever see number push it on the stack
(2) Whenever see an n-ary operator pop n items from stack and evaluate accord to that operator push result onto stack.
After finished expression result is top of stack.
Ex. 3 5 - 4 +

\[
\begin{array}{c}
3 \\
5 \\
3
\end{array}
\]

after

\[
\begin{array}{c}
2
\end{array}
\]

now push y

\[
\begin{array}{c}
4 \\
2
\end{array}
\]

Then see + \[
\begin{array}{c}
2
\end{array}
\] ← answer

END OF LECTURE AND SET #5

**********
HW 4 is up. Get input now returns as int.
HW3 solutions up, module commenting.

Last Day - was talking about stacks
Today - implementation of stacks. Make some remarks. Then talk about Queues and recursion.

Code for stacks

```cpp
template <class Item>
Stack<Item>::Stack(int s)
{
    CurSize = 0;
    maxSize = s;
    array = new Item[s];
    if(array == NULL)
    {
        cout << "Out of memory";
        exit(1);
    }
}
```

```cpp
Random Remarks:
(1) Code for implementation in book is in class definition
template<class Item>
class Stack
{
    //stuff
    Item pop() { return array[--CurSize];}
    // stuff
}
```

This is legal and sometimes useful if the implementation is really short. But if takes more than a couple lines to implement do as in class.

(2) Rather than implement using arrays could have implemented Stack using list. For example, to push an Item on stack create a new Node with that Item and insert at head of list.

(3) In above, pop, push, empty all take O(1) time to run.

(4) Sometimes hear term FILO used for stacks (First in Last out)

(5) Related to (2) we can in fact implement lists as 2-D arrays.
Queues - Like a stack except FIFO (first in first out)
Interface for Queues

class Queues
{
private:
  // implementation Specific
public:
  Queues (int s);
  ~Queues ( );
  bool empty ( ); // is queue empty?
  void put (Item);
  // put i at end of Queue.
  Item get ( );
  // return first item in Queue.
};

Ex. Printer Queues.
Suppose you want to print a bunch of files. In Unix, type
lpr a.ps <ret>
lpr b.ps <ret>
lpr c.ps <ret>
Can type this faster than it takes to print out a.ps. Also there may have been people before you printing stuff. What happens is your documents are placed on a print queue.

<table>
<thead>
<tr>
<th>earlier stuff</th>
<th>← head of queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.ps</td>
<td>→ after earlier stuff prints</td>
</tr>
<tr>
<td>b.ps</td>
<td></td>
</tr>
<tr>
<td>c.ps</td>
<td>→ tail of queue</td>
</tr>
</tbody>
</table>

After a.ps goes to printer

<table>
<thead>
<tr>
<th>b.ps</th>
<th>← head</th>
</tr>
</thead>
<tbody>
<tr>
<td>c.ps</td>
<td>← tail</td>
</tr>
</tbody>
</table>

Then b.ps goes to print.

| c.ps | ← headTail |

Then Queue is empty. Basically printer call remove function of Queue.

Ex. 2: Operating systems sometimes use multilevel priority queues to schedule tasks. Each task has a priority. There is queue for each priority level giving which task to execute next.

<table>
<thead>
<tr>
<th>Level</th>
<th>Job 1</th>
<th>→</th>
<th>Job 2</th>
<th>→</th>
<th>Job 3</th>
<th>← queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>level 10</td>
<td>Job 20</td>
<td>→</td>
<td>Job 21</td>
<td>→</td>
<td>Job 22</td>
<td>← tail</td>
</tr>
</tbody>
</table>

Jobs of level 0 execute first, then level 1, etc.
Among jobs of a given level acts like a queue, i.e., First in First out.
NOVEMBER 8, 2000

Couple of typos from last day. When we have templates in driver file need to include .cpp file of implementation. So should probably put the .cpp file between #ifndef/#endif. When we implement member functions we should do template <class Item>
Stack<Item>::Stack (int s)
{
  //code
}

Today we will talk about recursion.
Definition: A recursive algorithm is an algorithm which calls itself.

Ex int factorial (int N)
{
  if (N==0) return 1; ← Base cases or stopping case
  return N*factorial(N-1); //**
}
Ex factorial (3)
returns 3*factorial (2)
returns 2*factorial (1)
returns 1*factorial (0)
returns 1

So get 3*2*1*1 = 6 returned.

Saw before that function calls are implemented with stacks. So functions that call themselves okay. What is the advantage of having recursive functions?
1. Very useful in coming up with correctness proofs of your algorithms.
2. Generally, easier to get a recurrence relation that bounds runtime of your algorithm.
3. Much easier to write compilers using recursive descent methods than iterative ones.

What are the disadvantages?
Since use runtime stack takes more space to run a recursive program.
Also calls to runtime stack generally slower than incrementing a counter in an iterative program.

Ex proof of correctness of factorial
Claim for all n factorial (N) returns N!
Proof by induction
Base Case: When N==0 factorial (N) returns 1 and 0! = 1, so works.
Inductive Step: Assume factorial (N) = N! to prove factorial (N+1)=(N+1)!
Consider factorial (N+1), line ** will be executed.
So (N+1)*factorial(N) returned. Factorial(N) = N! By the inductive hypothesis.
So factorial (N+1)=(N+1)*factorial(N) = (N+1)*N! = (N+1)!

Number of sub calls to factorial on factorial(N), Subcalls(N) = 1+Subcalls(N-1) Subcalls(0)=0. Subcalls(N) is a recurrence relation and Subcalls(N)=N.

Ex Let's try to write a recursive function which computes x^n for n≥0
double pow(double x, int n)
{
  if (n==0) return 1;
  return x*pow(x, n-1);
}
As in factorial case pow(x, n) makes O(n) subcalls to compute n.

\[ x^n = \begin{cases} 
(x^{n/2})^2 & \text{if } n \text{ is even} \\
(x^{n/2})^{n}x & \text{if } n \text{ is odd} 
\end{cases} \]

Using this can get an algorithm which takes O(log N) many subcalls.
Double pow2(double x, int n)
{
  double temp;
  if (n==0) return 1;
}
temp = pow2(x, n/2); // n/2 rounds down
if(x%2==0) return temp*temp;
return temp*temp*x;
}

Why did I use temp rather than pow(x, n/2)*pow(x, n/2)*x?
To avoid recomputing pow2 twice which would produce no savings over previous algorithm.

Some remarks on recursion:
If don't have a good base case can get infinite loops.
void bob()
{
    bob(); // will go till runtime stack overflows.
}
NOVEMBER 13, 2000

Hw3 will be returned by 3. Hw 5 up tomorrow.

Last day-recursive functions
Today-more recursive functions, recursive member functions, function pointers, memoization.

Recursive member functions (a member function which calls itself)
ex. class MyNum
    
    public:
        //some methods
        double pow(int n);
    
    private:
        double x;
    
    double MyNum::pow(int n)
    
    { int temp;
        if (n==0) return 1;  //example of recursion in a member
        temp=pow(n/2);  //function
        if (n%2==0) return temp*temp;
        else return temp*temp*x;
    }

Some recursive list operation
int length(link x) ← computes length of list
int length(link x)
    
    { if (x==NULL) return 0;
        return 1+length(x->next);
    }

void traverse(link h, void visit(link))   hey, we're passing a function
//this function goes through each node of the list and does a visit operation on each node.
void traverse(link h, void visit)
    
    { if (h==NULL) return;
        visit(h);  //do whatever visit does on h
        traverse(h->next, visit);
    }

since recursive call last statement in block of code, this function is called tail recursive. A tail recursive function can easily be replaced by a
for loop and some compilers will do this automatically.

Back to fact we passed a function as a argument...
Just like any variable each function you write has an address in memory so you can have a pointer to this address and use this pointer to run the
code at that address.

ex. void (*myfnptr)(link h);
*myfnptr is a pointer to function which takes a link as its argument and returns a void.
void myvisit(link h)
    
    { cout << h->item << endl; }

myfnptr=&myvisit;  //in main or some function we would write this
//now myfnptr points to where compiled my visit code is.
(*myfnptr)(mylist);  //where mylist is a link calls the function myvisit on mylist
To call *traverse* could do:

```c
traverse(mylist, _myvisit); //can't remember may need &
```

Can "fake" classes in C using functions pointers as member variable of structures.

Let me now go back to writing one more recursive version of a list operation

```c
//traverse a list from the end to the start
void traverse R(link h, void visit (link))
{
    if (h == NULL) return;
    traverse R(h->next, visit);
    visit(h);
}
```

//this code uses the runtime stack to store list in reverse order first then visits each node in this reversed list.

```
1 2 3 4

        //deus next is NULL stop

----fu- ii[l- i.{tl swa.L.* 6 ^lqLL s+ae

```

Divide and conquer

Idea: recursively split problem into two subproblems of roughly same size, do each subproblem, then combine the result.

Divide and conquer to find max item in an array.

```c
template <class Item>
Item max(Item a[], int l, int r)

//precondition: < defined for Items
{
    if (l == r) return a[l];
    int m = (l + r) / 2; //midpoint of area in which searching for a maximum
    Item u = max(a, l, m);
    Item v = max(a, m + 1, r);
    if (u > v) return u;
    else return v;
}
```

END OF LECTURE

November 15, 2000

HW5 is up. Will have HW4 solutions up later today.

Last day - talked about Divide and conquer algorithms

Today - memoization, graphs/trees

Dynamic Programming/Memoization = approach to solving recursive problems where we store subproblems have already taken care of.

Consider the following very inefficient algorithm to compute Fibonacci numbers:

```
F(0) = 0
F(1) = 1
F(n) = F(n-1) + F(n-2)
```

```c
int F(int i)
{
    if (i < 1) return 0;
    if (i = = 1) return 1;
    return F(i-1) + F(i-2);
}
```

This algorithm will take exponential amount of time to compile F(i).

To see this consider computation of F(10):

```
F(9) + F(8)
F(8) + F(7) + F(7) + F(6)
```

Notice we compute F(8) twice
We recompute \( F(1), F(0), \ldots, F(10) \) many times since the Fibonacci sequence is exponential in \( i \). We have to do exponentially many recomputations of leaves. However, not too hard to compute Fibonacci sequence in linear time iteratively.

```c
int maxN = 1000, F[maxN];
F[0] = 0; F[1];
for (i = 2; i < maxN; i++)
  F[i] = F[i-1] + F[i-2];
//starts at F[0], F[1] and computes up to F[maxN-1].
//This is called bottom-up approach. Recursive approach work, from
//F(i) to F(1), F(0). This is called top-down approach.
Can we make the top-down approach as efficient as bottom-up approach? Yes.
Solution is to memoize i.e., store subcomputations.
int F(int i)
{
  static int knownF[maxN]; //will store values we've already computed.
  //Static local variables return their values
  //between function calls.
  if(knownF[i] ! = 0) return knownF[i];
  int t;
  if(i < ) return 0;
  if(i = = 1) return knownF[i] = 1;
  if(i > 1) t = F[i-1] + F[i-2];
  return knownF[i] = t; //store t then returns its value
}
```

Consider \( F(5) \)

```
\[ F(5) = 5, \quad \text{so this algorithm is linear in } n. \]
```

Graphs/Trees
Trees - useful in describing dynamic properties of algorithms. For instance, the Fibonacci program above. Also useful as data structures to organize information.

Ex. (1) File directories: can think of desktop as root of tree.
    Folders are internal nodes. Files are leaves.
    
    mycomputer    Desktop    windows
    \[\text{File 1} \quad \text{File N}\]

Ex. (2) Search trees: a way to organize data say in a dictionary, database, for quick retrieval.
Ex. (3) Sorting algorithms

Uses of graphs
Useful to represent any kind of network (internet, neural net, cellular automata). In graphics all objects are approximated with polygons which are graphs.

Formal Definitions
Definition: A (directed) graph is an ordered pair \( G = <V,E> \)
\( V \) a set of vertices
\( E \subseteq V \times V \) - a set of edges

\[ G = <\{1,2,3,4,5\}, \{(1,2),(1,3),(3,2),(4,5)\}> \]

\[ G \text{ represent picture.} \]

1 2 3 4 5
\[ \text{G represent picture.} \]
If we had written edge as (3,2) rather than (3,2) and similar for each edges, we'd get undirected graph.

END OF LECTURE

NOVEMBER 17, 2000

HW5 due Wednesday not Monday after Thanksgiving
Last day - we defined graph
Today - talk about trees. Draw some pictures, define things.
Recall a graph was a pair \( G = (V,E) \)

Need a couple definitions to talk about trees

Definition: a sequence of vertices connected by edges leading from one vertex to another vertex with no vertex appearing twice is called a simple path. A simple path with same start and end vertices is called a cycle.

Ex.

\[ (1, 2, 3, 1) \text{ is a cycle.} \]

\[ (1, 2, 3, 4) \text{ is a simple path.} \]

\[ (1, 4, 3) \text{ not a simple path since no edge between 1 and 4} \]

Definition: A graph is connected if any two vertices are connected by a simple path.

Ex. To avoid having to make doodles
to say a vertex is connected to itself we consider singleton sequences as legitimate simple paths

Ex.

\[ 1 \rightarrow (1) \text{ is connected} \]

\[ 1 \rightarrow 2 \rightarrow 3 \text{ is not connected} \]

\[ 1 \rightarrow 2 \text{ is connected} \]

Ex.

\[ 1 \rightarrow 2 \text{ can't go 2 to 1, not connected} \]

Ex.

\[ 1 \rightarrow 2 \text{ no can't go 2 to 1} \]

Definition: A rooted tree is a graph with a vertex called the root such that there is exactly one simple path from the root to any vertex and any edge the graph belongs to one such path.

\[ \text{not a tree since there are 2 paths to 3.} \]

If take a rooted tree and "forget" directions then you have a tree (sometimes called a free tree)

A tree is always connected

Ex.

not a tree but this graph contains two trees.
so a graph made up of trees called forest
Let's talk about implementing trees

```cpp
template <class T>
struct node { Item i; node *l, *m, *r; }
```

// these addresses
// would be stored in consecutive locations in memory
l
m
r

(Aside: A subgraph of graph \( G = <V, E> \) is \( G' = <V', E'> \) such that \( V' \subseteq V \), \( E' \subseteq E \). A subtree is a subgraph of a tree which is a tree).

An ordered tree is a rooted tree in which children of every vertex are given in some order. (So corresponds to how we implement trees with structures.)

Also how we draw trees

- left \( \rightarrow \) right

So can order node by how for right they are.

Can tweak above structure if want two children or more than 3.

```cpp
template <class Item>
struct node { Item i; node *l, *r; }
```

If we wanted more than three

```cpp
template <class Item>
struct node { Item i; node *children[ ]; } however many we want
```

**Definition:** M-ary tree is an ordered tree such that every vertex has 0 or M many children. In case M = 2, called binary tree. Now 0 children case is implemented by a pointer to a NULL node. These are called external nodes. Nodes which aren't external are internal. At leaf for an ordered tree is an internal node whose children are all external (A leaf in a rooted tree is a node without children)

**Ex.**

```
    |4|
  /  |  |
 Null|2| Null
        \
 external node
 Null           Null
```

This is a binary tree

**Definition:** The level of a node in a rooted tree is the number of edges in a simple path from root to node.

**Ex.** This tree has height 2.
NOVEMBER 20, 2000

Last day: was talking about trees
Today: more trees, tree traversals.

Given a rooted tree

```
1
/   \
2     3
/ \
1   5
```

the level of a node is the length of simple path from the root to the node.

level of node 3 is 1
root is level 0
level of node 5 is 2

Height of tree is the maximum level in the tree. For above tree is 2.
Height useful to bound search times for items in trees.

Definition: the Path Length of a tree is the sum of the levels of all the tree's nodes.

ex. for above tree 0+1+1+2+2+2+2=10

Internal path length—sum levels of internal nodes
External path length—sum levels of external nodes

```
    1
   /   \
  2     3
 / \
1   5
```

IPL=0
EPL=1+1=2

**Theorem** A binary tree tree with N internal nodes has N+1 external nodes.

Proof (by induction)
If tree is just a root which is a null pointer. Have 0 internal nodes; 1 external node so works.
Assume true for binary trees with <N nodes. Consider a tree which N internal nodes. Left subtree has N-K-1 internal nodes. Right subtree has K<N internal nodes.

```
L
/ \
/    \
/      \
N-K-1
```

By the induction hypothesis the left subtree has N-K-1+1=N-K external nodes right subtree has K+1 external nodes. Adding we get N-K+K+1=N+1. QED.

**Theorem** The height of a binary tree with N internal nodes is at least \( \log N \) and at most N-1.

Intuition
Proof (by induction) for the tree with 1 internal node is true
Height is 0 = \log 1 = 1 - 1

Suppose true of trees with <N nodes. Take any tree with N internal nodes. It must have at least one leaf (prove this by induction). Replace this leaf with an external node gives a tree with N-1 internal nodes.

By induction hypothesis, this latter tree has height <N-2. So former is at most one taller so has height <N-1.

For log N case, consider root of tree.

So can apply induction hypothesis to two subtrees.
Height of the whole tree is max { \log K + \log (N-K-1) } + 1

If try to make this as small as possible K = N/2. So height is at least \log \left( \frac{N}{2} \right) + 1 = \log N + \frac{1}{2} + 1 = \log N.

Tree traversals
ex. 2+(3*5)
could be useful to go through the nodes of tree in some order to calculate the value of this expression.

Preorder traversal visit current node then visit left then visit right ABCDE
Inorder visit left, visit root of current tree, visit right subtree CBDAE
Postorder traversal visit left, visit right, visit root. CDBEA

END OF LECTURE
NOVEMBER 22, 2000

Last Day - we proved some facts about trees we talked about preorder, inorder, postorder tree traversals.
Today - look at code for above and other algorithms about trees.

Code for preorder traversal (using recursion) //visit Root - Left - Right
t void preorder ( link h, void visit(link))
{ if(h == NULL) return;
    visit(h);
    preorder(h->l, visit);
    preorder(h->r, visit);
}

How to "hardwire" a tree to test out this function.
Let's say want to create the tree,

```
3
/  \
1   2
/  \
4   5
```

Using our struct node for binary trees where link is a typedef for node*

```c
node root, left, right, lleft;
link h = &root;
root.item = 3;
root.l = &left;
root.r = &right
left.item = 2;
left.l = &lleft;
left.r = NULL;
right.item = 4;
right.l = right.r = NULL;
lleft.item = 1;
lleft.r = lleft.l = NULL;
preorder(h, vis);
// where vis is
void vis(link h)
    {cout << h->item << endl;}
```

Output
3
2
1
4

Level-order traversal  //useful for breath first search

```
1 2 3 4 5
```

Unlike other three traversals which were coded using recursion (and hence implicitly used stack) we will code this iteratively using a queue.
Let's assume we've already coded a template class for Queue.
To do level order traversal...

```c
void levelorder(link h, void visit(link))
{ Queue<link> q(max); //max is some static constant
    //queue that stores links
    q.put(h);
    while(!q.empty( ))
    { visit(h = q.get( ));
        // get head of queue. Set h equal to this visit h.
        if (h->l != NULL) q.put(h->l);
        if (h->r != NULL) q.put(h->r);
    }
}
```

Suppose h was
Suppose visit is vis given above
First put 3 on queue
Second remove 3 print it
    put 2 4 on queue
Third remove 3 print it
    put 1 on queue
Fourth remove 4 print it
    nothing to put on queue
Fifth remove 1 print it
    done
Print
    3
    2
    4
    1

Simple algorithm to print out a tree.

void printNode(item x, int h)
{ for(int i=0; i < h; i++)
    cout << " ";
    cout << x << endl;
}

void show(link t, int h)
{ if (t == NULL)
    { printNode (*, h);
        return;
    }
    show(t->r, h+1);
    printNode (t->item, h);
    show (t->l, h+1)
}

The function show prints out tree t h spaces over

*  
  4 *
  3 *
  2 *
  1 *

END OF LECTURE AND SET #8


NOVEMBER 27, 2000

HW4 returned by tomorrow. This Friday practice final on the web.

Last Day - talked about tree traversals and algorithms based on trees.
Today - sorting algorithms.

Sorting
Have a collection of objects for which operator < defined (also have operator =, == defined). Want to put objects in order.

For large data sets best algorithms are O(NlogN).
However, the algorithms we consider today will be easy to write and for small data sets work well.

Some considerations about sorting algorithms.

Internal vs. External Sorting
In internal sorting data to be sorted fits entirely in memory.
External sorting involves leaving some of data on disk or tape during sorting process.

Disk access will be slow part of your algorithm

On a list of N items we advance i, O(N) times.
On each advance of i need to do an average O(N) comparisons /swaps.

So algorithm is O(N^2)

Code for Insertion sort

On a list of N items we advance i, O(N) times.

On each advance of i need to do an average O(N) comparisons /swaps.

So algorithm is O(N^2)

Code for Insertion sort

template <class Item>
void exch (Item &A, Item &B)
{ A = B; B = A; } //swaps A and B. Note =
//overloaded.

template <class Item>
void complextch (Item &A, Item &B);
{ if (B < A) exch(A, B); } //note need < defined.

template <class Item>
void sort (Item A[ ], int l, int r) //sort between l and r.
{ for (int i = l + 1; i <= r; i++)
    for (int j = i; j > l; j--)
        complextch (A[j-1], A[j];
}

Above is a so-called nonadaptive sorting algorithm i.e., operations we did, did not depend on how the original array was ordered.
(If array was already ordered would do same comparisons.
Swapping of course depends on array).

Adaptive - comparisons made during sorting depend on how the data set ordered.

Some more issues to consider when sorting . . .

Consider a list of people:

```c
struct person {char fnamed[20], lnamed[20];};
```

Bob Parker & Regis Philben

Then both

1. Bob Parker 2. Bob Parker
   Regis Philben &  Regis Philben
   Regis Philben  Tom Philben
   should be considered sorted.

1. Preserves the fact that Tom Philben occurred before Regis Philben in original list.
2. Does not preserve this property.

Sorting algorithm which preserves this property is called stable.

Insertion sort is stable.

Remember selection sort.

END OF LECTURE ********************
Hw6 up. Bonus up by later today. Practice final up Friday. For honors section we will meet on Monday at 2 and discuss bonus.

Last day—talked about insertion sort
Today—making insertion sort faster and talk about bubble sort.

Remember selection sort for final.

Ex: Selection Sort

```
5 3 9 7
```

Find minimum in unsorted part
i.e. 3

```
3 5 9 7
```

swaps minimum value with left most to be sorted value

```
3 5 9 7 [ ]
```

find min
swap with 5
sorted

```
3 5 9 7
```

find min
7 swap with 9
sorted

```
3 5 7 9
```

all sorted

```
3 5 7 9
```

done

Let's go back to insertion sort and try to make it faster. Suppose already sorted to here in array

Consider

```
2 3 5 4 6 4
```

i.e.

```
2 3 5 4 6
```

compares does nothing

```
2 3 4 5 6
```

compares does nothing

1) Only need to keep comparing values and swapping till find a value less than 4. Actually, we don't need to keep doing swaps. We store 4 then move everything after the 3 right by one.

```
2 3 5 6 4
```

store 4

```
2 3 5 6 6
```

move 6 right by 1 since greater than 4

```
2 3 5 5 6
```

move 5 right by one

4>3, so then place 4 in this opened slot

```
2 3 4 5 6 6
```

this saves us the time needed in swap to store things in a temporary variable.

For this to work need to know that there was some value left of 4 initially that is less than 4. Otherwise, could end up going left of array.

To handle this we first find the least item in the array and put it in the zeroth position. This smallest value is sometimes called a sentinel.

i.e. it's a value that guards so we don't leave our data structure. In this case, the array.

Recalmed code for insertion sort

```
template<class Item>
void insertion(Item a[], int l, int r)
{
    int i;
    for (i=r; i>l; i--)
        complexch(a[i-1], a[i]); //set up sentinel
        //smallest value in a[1]
        //store a[i] till we find where to
        //insert it
        while (v>a[i-1])
        { a[i]=a[i-1];
          j--;
        }
        a[i]=v;
    }
}
```

Save ourselves a couple of assignments each loop over old program since don't use temporary variable. Turns out this program is roughly twice as fast. Unlike our previous version of insertion sort this version is adaptive, i.e., comparisons made does depend on original order of array.

Bubble sort

Start with

```
6 4 9 5
```

compare current value at index with value at index-1. swap if smaller. Keep cycling through array till get to left hand side.

```
6 4 5 9
```

no change

```
6 4 5 9
```

swap

```
4 6 5 9
```

4 is sorted

```
4 6 5 9
```

don't do anything

```
4 6 5 9
```

swap

```
4 5 6 9
```

done
do nothing

\[
\begin{array}{cccc}
4 & 5 & 6 & 9 \\
\end{array}
\]

sorted

\[
\begin{array}{cccc}
4 & 5 & 6 & 9 \\
\end{array}
\]

END OF LECTURE

DECEMBER 1, 2000

Practice Final is up. Final is in Moore 100 on December 10 (Sunday), 3-6

Last Day – Talked about speeding up insertion sort, bubble sort

Today – bubble sort, merge sort

Bubble Sort

Idea: go in passes, right to left “bubbling” smallest unsorted value to the left.

1st pass

Nothing is sorted

\[
\begin{array}{cccc}
1 & 6 & 4 & 5 \ 5 \\
\end{array}
\]

5 smaller so swap

\[
\begin{array}{cccc}
1 & 6 & 4 & 5 \ 9 \\
\end{array}
\]

4 smaller so don’t swap

\[
\begin{array}{cccc}
1 & 6 & 4 & 5 \\
\end{array}
\]

4 smaller so swap

\[
\begin{array}{cccc}
1 & 6 & 5 & 9 \\
\end{array}
\]

Now 4 in correct position

2nd pass

sorted

\[
\begin{array}{cccc}
4 & 5 & 6 & 9 \\
\end{array}
\]

5 smaller so don’t swap

\[
\begin{array}{cccc}
4 & 6 & 5 & 9 \\
\end{array}
\]

5 smaller so swap

\[
\begin{array}{cccc}
4 & 6 & 5 & 9 \\
\end{array}
\]

Now 5 in correct position

3rd pass

sorted

\[
\begin{array}{cccc}
4 & 5 & 6 & 9 \\
\end{array}
\]

don’t swap since 6<9

\[
\begin{array}{cccc}
4 & 5 & 6 & 9 \\
\end{array}
\]

4th pass pass done

How many comparisons made by Bubble sort to sort N items?

We make N passes through array

1st pass make N-1 comparisons

2nd pass make N-2 comparisons

3rd pass make N-3 comparisons

... last pass 0 comparisons

Then total number of comparisons \( \sum = 0 + 1 + \ldots + N-1 = N(N-2)/2 \)

So we make \( O(N^2) \) comparisons

So Bubble sort is \( O(N^2) \) algorithm

Code for Bubble Sort

```cpp
template <class Item>
void bubble (Item a[], int l, int r)
{
    for(int i=l; i<=r; i++) // how many passes we've made i.e., where
        // the sorted marker is.
        for (int j=r; j>l; ;--j) // does bubbling to the left for a given
            // pass
            compexch (a[j-1]; a[j]);
}
```

Let’s look at a faster sort algorithm

Merge Sort

A divide and conquer algorithm

So if want to sort \([5, 2, 9, 6, 7, 3]\)

split into two halves

\([5, 2, 9]\)

\([6, 7, 3]\)

sort these two halves and then “merge” the results.

So would first sort left half

\([5, 2]\)

split into halves

\([5]\)

\([2]\)

\([5]\) ← sorted

sort right half of \([5, 2]\)

\([2]\) ← sorted

then merge the results

\([3, 2]\)

\([5, 2]\)

outside of region so copies → \([2, 5]\)

Now sort right hand of \([5, 2, 9]\)

\([9]\) ← sorted

Now merge

\([2, 5]\)

\([9]\)

\([2, 5, 9]\)

\([2, 1, 3]\)

off the array \([2, 1, 3]\)

so copy what’s left of other

Now done sorting left half of \([5, 2, 1, 6, 7, 3]\)

so sort right half

\([6, 7, 3]\)

split in two then sort left and right

\([6, 7]\)

split again

\([6, 7]\)

\([6, 7]\)

so sort right half of \([6, 7]\)

\([7]\) ← sorted

\([6, 7]\)

\([6, 7]\)

\([6, 7]\)

\([6, 7]\)

\([6, 7]\)

\([6, 7]\)

Now sort right half of \([6, 7, 3]\)

\([3]\) ← sorted

Merge results
Now Merge

\[
\begin{array}{c}
2.519 \quad 3.617 \rightarrow 2.517 \\
2.519 \quad 3.617 \rightarrow 2.518 \\
2.519 \quad 3.617 \rightarrow 2.315 \\
2.519 \quad 3.617 \rightarrow 2.316 \\
2.519 \quad 3.617 \rightarrow 3.516 \\
2.519 \quad 3.617 \rightarrow 3.514 \\
2.519 \quad 3.617 \rightarrow 3.516 \\
2.519 \quad 3.617 \rightarrow 3.517 \\
2.519 \quad 3.617 \rightarrow 3.517 \\
2.519 \quad 3.617 \rightarrow 3.516 \\
2.519 \quad 3.617 \rightarrow 3.516 \\
2.519 \quad 3.617 \rightarrow 3.516 \\
\end{array}
\]

Notice when merge two halves which have combined a total of N elements, it is an O(N) operation.

So now let's look at recursion tree for sorting N items

\[
\begin{array}{c}
\text{merge N/2 items} \\
\Rightarrow \text{merge N item} \\
\Rightarrow \text{merge N/2 items} \\
\Rightarrow \text{log N many levels of merges on each level do} \\
0(N) \text{ many operations.} \\
\Rightarrow \text{So total runtime} \\
O(N) \cdot \log N = O(N \log N)
\end{array}
\]

END OF LECTURE AND SET #9 ***************
Last Friday - talked about MERGESORT
Today - will give code of mergesort start talking about Hash tables

Remember how merge Sort worked:
To sort

\[ \begin{array}{cccc} 9 & 3 & 8 & 2 \end{array} \]

Split into two halves

\[ \begin{array}{cccc} 3 & 8 & 9 & \end{array}, \begin{array}{c} 1 & 2 & 7 \end{array} \]

sort both halves
Merge two halves to get sorted array

\[ \begin{array}{cccc} 1 & 2 & 3 & 7 & 8 & 9 \end{array} \]

Let's write code for MergeSort
First we'll write code for doing the merging

```cpp
template <class Item>
void merge (Item a[ ]; int l, int m, int r)
// merges a[l] ...a[m] with a[m+1]...a[r]
{ int i, j; //used for counters
  static Item aux[maxN]; //temporary array used for merge. Should be static since we don't
  //want to reallocate this memory every time merge called.

  // Now we'll first copy original arrays into aux array and copy merged results into
  // original array
  for (i = m+1; i > l; i-) //copies right half in reverse order.
    //Now we merge results
    for (int k = l; k < r; k++)
      //used to figure out what to put in a[k]
      if (aux[j] < aux[i] )
        //i starts at l by the way we set up for loop above
        //Also j = r to start
        a[k] = aux[i--];
      else a[k] = aux[i++]
  }
}
```

If j array finished first need to have check to make sure didn't go out of array.

\[ \begin{array}{cccc} i & j \end{array} \]

Idea of copying 2nd half on in reverse order called \textit{bitonic} merging

Sorting algorithm

template <class Item>
void mergesort (Item a[ ], int l, int r)
//sort a[l]...a[r]
{
  if (r <= l) return;
  int m = (r+l)/2; //calculate midpoint

  mergesort(a, l, m); //sort left half
  mergesort(a, m+1, r); //sort right half
  merge(a[l],m,r); //merge results
}

Hash tables
Often want to store data as key and associated information. This is called a dictionary. In "Real-life" dictionary, the key is a word and the definition is the associated date.

Other example,
Key = Employee ID
Associated date = work history

How to store things in a dictionary
(1) Use a flat file (unordered file)

<table>
<thead>
<tr>
<th>K1</th>
<th>→</th>
<th>K2</th>
<th>→</th>
<th>K3</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td></td>
<td>data</td>
<td></td>
<td>data</td>
</tr>
<tr>
<td>pointer</td>
<td></td>
<td>pointer</td>
<td></td>
<td>pointer</td>
</tr>
</tbody>
</table>

To find an item have to search on average through half of list file.
To insert can just add to head to file so can be done in O(1) time.

END OF LECTURE

DECEMBER 6, 2000

Will go over Practice Final on Friday. Final Moore 100 Sunday 3-6

Last Day - finished up mergesort, talked about dictionaries/tables
Today - talk about hash tables

Dictionaries/Table

<table>
<thead>
<tr>
<th>Key (employee ID)</th>
<th>Data (records)</th>
</tr>
</thead>
<tbody>
<tr>
<td>999 25 9999</td>
<td>Ted Smith etc.</td>
</tr>
<tr>
<td>677 26 7777</td>
<td>Barb Walters etc.</td>
</tr>
</tbody>
</table>

Considered ways to store Tables

1. Flat file
   Just a file/or linked list when data is written out in an unordered fashion. To find data given key expected search time O(N). To add new entry can be done in O(1) time since can just keep as part of our structure a pointer to the last element in file or last element in list. To insert new item add after last item then update this pointer.

Can we implement tables so look up and insert both take O(1) time?
Yes.

1st way: Open Addressing
Let N=total number of different possible keys. Make an array data[] of size N whose elements hold associated info.
Ex: if k = 997532100 can look at data[997532100] to find info.
Insert can be just using assignment.
data[k] = info;
This is very wasteful. If want to store only 50 employees but use SSN as key need an array of size 10^9.
We'd like to achieve same result without wasting so much space.
Suppose we have N items to store. We'd like to use an array of size M not too much bigger than N and somehow map the keys into the set of size M.
This mapping is called a hash function.

h: K \rightarrow [M]

\begin{align*}
  &c & p \\
  &y & a \\
  &c & e \\
  & & 0, \ldots, M
\end{align*}
To store data we’ll use an array called hash table. To insert we’ll usually do: table[h(key)]=info. To look up data for key K we’ll usually look at the entry table[h(K)]. Since Key’s usually of some fixed length not depending on our table size, we’ll choose h(K) so that it takes constant time with respect to table size.

Ex Suppose want to store so employee using SSN as key
Could take hash function h(K) = sum of digits in SSN%10,003 then could use this to store employees in an array of size 10,003.
If store 1000 or 2000 employees amount of time to compute h(K) for a given key won’t change.
Problem with above scheme: What if h(K) = h(K') but K≠K’?
This is called a collision.
One way to resolve this problem is to use linear probing.

Ex Suppose h(x)=x%7

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
</table>
| 1 | 1 | → data for 1
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

If try to store 8 get a collision

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Store 8 and its data in 1st unoccupied space afterward.
To look up an entry with key K. Compute h(K) look if table [h(K)] has same key. If not step forward through table till find K.

Problem: Consider same h but table

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| 6 | 13 | → data

Want to store 6, data 6 so h(13)=h(6)
so have a collision if step forward we go off end of array. Instead of stepping forward we step forward %7.
So would store 6 in location 0.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

→ data 6

Provided not "too many" collisions hash tables give O(1) search/insert times.

Ex Suppose we insert 3147 into a hash table using x%4 and linear probing to resolve collisions. Draw resulting hash table.

Insert 3

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Insert 1

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Insert 4

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Insert 7
gives collision

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

END OF LECTURE

Final has 7 problems. Each 5 pts. = 35 pts. total. No caps/cellphones close book/closed notes. Bring Photo ID

Practice Final

1) Define the following term

a) FILO - First In Last Out - Acronym used for describing inserting and removing objects from the stack.

b) tail recursion - a recursive function where there is one recursive call and it is the last statement in the function.

Ex int fact (int N)

```cpp
    if (N == 0) return 1;
    return N * factorial (N-1);
```

Point is that can replace this kind of recursion is can replace it easily without for loop.

c) memoization - also dynamic programming. The process of storing the results of subproblems of recursive pass as we do recursion so we don’t have to recompute them.

Ex Fibonacci program did in class.

d) external node - in an M-ary ordered tree is a node without children (implementation with a NULL).

e) adaptive sorting - a sorting algorithm whose operations (the comparisons it makes) depends on the way the original unsorted data was sorted.

Ex 2nd insertion sort we did.

2) Write a program to reverse a list whose Node's in list given as on Practice Final.

```cpp
template <class Item>

Node<Item> * reverse (Node<Item> list) { // pointer to reversed list // list to be reversed
    Node<Item> *t, *y = list, *r=NULL;
    1st // will store what's left to be reversed // reversed list so far
    y -> 4 -> 2 -> 3
    r -> NULL

    2nd
    y -> 2 -> 3
    r -> 1
```
done

while (y != NULL)
   { t = y -> getNext();
     y = setNext(t);
     r = y;
     y = t;
   }
return r;
}
// end function

3) Briefly explain why the first version of insertion sort we described in class was nonadaptive and why it runs in time O(N^2) to sort N elements.

1a Version: template <class Item>
   void sort(Item a[], int e, int r)
   { for (int i = l + 1; i <= r; i++)
      for (int j = i; j > 1; j--)
         if (a[j] < a[j-1])
            exch (a[j-1], a[j]);
   }

It's nonadaptive since it will do same comparisons regardless of input array.
It's O(N^2) since outer for loop runs N-1 times on array of size N.
Inner for loop does on average N/2 comparison exchanges. So run time

O((N-1)N/2) = O(N^2)

4) a
   /\  p1
  /\  p2
     \  y

Write out nodes in

a) preorder
   ahp1p2y
b) inorder
   hap2p1y
c) postorder
   hpa2yp1a
d) levelorder
   ahp1p2y

5) Write a program to compute # of external nodes in a tree
int exNodeCount (link h)
{ if (h == NULL) return 1;
   return exNodeCount (h->l) + exNodeCount(h->r);
}

END OF LECTURE AND SET # 10

5