CSC270
Algorithms and Data Structures,
(day section)

Danny Heap
heap@cs.utoronto.ca
978-5899
SF4306
and François Pitt

December 16, 2002

You may prefer the HTML version of this document, at
http://www.cs.utoronto.ca/heap/270F02/Notes/Notes.html

Contact information

I can be reached at heap@cs.utoronto.ca, 978-5899, or in SF4306A. My office hours are 4–5 pm, Tuesdays and Thursdays.

Time	Monday	Tuesday	Wednesday	Thursday	Friday
9:00					
10:00					
11:00		ECE242 GB120	CSC270 SF1101		CSC270 SF1101
12:00					
13:00					
14:00	ECE242 MC252			ECE242 RS211	
15:00					
16:00	ECE242 office hour	CSC270 office hour	CSC270 problems	CSC270 office hour	ECE242 office hour

Announcements (reverse chronological order)

- Tuesday, December 17 office hour: starts 35 minutes late (4:35), since I must invigilate another exam from 2–4:30.
- Exam Summary

- Assignment 4 is now available.
- Sample (PostScript), Sample (PDF) midterm solution (except Q3).
- Assignment 3 is now posted: Assignment 3 (PostScript), and Assignment 3 (PDF). There are also Assignment 3 FAQ
- Midterm summary (PostScript), and Midterm summary (PDF) now available.
- **Aid sheet:** Yes — it must be no larger than 8.5"x11", handwritten on (up to) 2 sides.
- Where do you write your midterm? If the **third** letter of your userid is in [a-l], then you write in CG 150. Otherwise, you write in CG 250.
- Previous midterms include François Pitt's Test 1: PostScript, Test 1: PDF, Test 2: PostScript, Test 2: PDF. The usual cautions apply: we aren't covering the material in the same order as François did, we have one test instead of two, I'm not François....
- **Midterm, November 1, 11:10–12:** This will be held in the Canadiana Gallery, rooms 150 and 250. Here's the information on the Canadiana Gallery:

  ```
  Canadiana Gallery (CG)
  14 Queen's Park Crescent West
  ```

  ... and you can look for building (CG) on the U of T map (behind Gerstein Library):
- **September 22:** Here's Assignment 1
- **September 20:** Assignment 1 will be posted this weekend.

September lecture summary

September 11

This course has two components, and each component has either two or four parts.

The first component is a collection of topics that stimulate computer programs:

- numerical methods
- graph theory
- simulation
- dynamic programming

The second component is an extremely condensed introduction to the C and C++ programming languages. You'll use one or the other of these two languages throughout the course, and later in your career.

Introduction to UNIX

Your cdf account is uses a unix (or linux) OS. You'll need to be familiar with the filesystem and a few commands, and these are (mostly) available in "A Student's Guide to CDF." We'll make photocopies available.

Introduction to C

We use the Gnu compiler `gcc`. Here's an example, assuming that somebody created a C source file called `program.c` in the current directory:

```
gcc -Wall -ansi -pedantic -o program program.c
```

- `-Wall` prints "all" (not exactly all, but lots of) warnings.
- `-ansi` scolds you if your C source is not compliant with the ANSI (1989) standard.
- `-pedantic` scolds you for even more things, which may be legal C, but are considered (by the compiler, at least) to be bad.
- `-o program` says put the executable in `program`

C is similar to Java

C and Java have some similarities:

- Statements continue until a semicolon (";") character
- primitive data types (except C has no `boolean`)
- `if`, `while`, `for`, and `switch` statements are very similar to Java
- C functions resemble Java static methods

C differs from Java

C precedes Java, and isn't object-oriented:

- C doesn't have classes (C structs are a long way off), and no exception mechanism.
- program structure differs (see below)
- C has pointers (addresses) rather than references. These are more easily abused!
- C has no booleans: 0 == false, !0 == true.
- I/O is different
- A simple (single-file) C program is like a Java program where all methods are static

Sample C program

```
/* add.c
 * a simple C program
 */

#include <stdio.h>
#define LAST 10

int main()
{
    int i, sum = 0;

    for ( i = 1; i <= LAST; i++ ) {
      sum += i;
    } /*-for-*/
```

```
    printf("sum = %d\n", sum);

    return 0;
}
```

The main parts are:

- preprocessor directives (notable by the `#` character and the lack of semicolons)
- global variable declarations
- function declarations
- `int main()`
- function definitions

Here's a brief description of those parts:

Comments: These start with `/*` and continue, ignoring newlines, until the next `*/`. Notice that this means there is no nesting of comments. We adhere to ANSI-C standard (until C99 becomes more ubiquitous), so you cannot assume that `//` comments until the end of a line will work. It may turn out that some compilers accept these, but you cannot assume that all compilers (particularly the one used by the marking TA) will.

`#include` directives: includes declarations of functions from header files for standard libraries. For example, `#include <stdio.h>` includes declarations of functions for the standard library that are useful for input/output. Note the distinct absence of semicolon

`#define` directives: Performs textual replacement, which is useful to define constants. For example, this is legal C:

```
#include <stdio.h>
#define BEGIN {
#define END }
int main()
BEGIN
    printf("Hello World!\n");

    return 0;
END
```

`int main()`: An executable program must have a `main` method, which is what's called when the program is executed. `int` indicates that this function returns an integer value to the shell where it is executed. In this course we will normally return 0.

September 13

Our sample C program included a function `printf` from the standard I/O library. You can declare and define your own functions:

```
/* add.c
 * a simple C program
 */

#include <stdio.h>
#define LAST 10
```

```
void Incr(int *num, int i);

int main()
{
    int i, sum = 0;

    for ( i = 1; i <= LAST; i++ ) {
        Incr(&sum, i); /* add i to sum */
    } /*-for-*/
    printf("sum = %d\n", sum);

    return 0;
}

void Incr(int *num, int i) {
    *num = *num + i;
}
```

Several things are going on here

1. There is a declaration of `Incr`.
2. There is a definition of `Incr`.
3. Since I want to change the value of `sum`, I need to pass its address to `Incr`. The `&` can be thought of as the "address-of" operator, and the `*` can be thought of as the "value-at" operator.

The include file `<stdio.h>` provides a declaration, but not a definition, of `printf`. If this is omitted, C will make assumptions about the return type and parameter list for `printf` — and these assumptions may well be wrong!

During the first compilation pass, the C compiler turns `add.c` into object code (machine language), making a note that it will link to `printf` and `Incr`. Linking to `Incr` is internal: its definition is in the same compilation unit (file) as its declaration and use. Linking to `printf` is **implicit**: the compiler links to an enormous library object (the C library) without even asking your permission. Occasionally you'll have to specify that the compiler should link to a non-standard library, such as `libm.o`, which has `math.h` as a header. This is the reason for the `-l m` required at the end of a compilation command that some of you saw during the problem session Wednesday afternoon.

Pointers

C allows you to directly manipulate the byte address in main memory of data and program code. Every type (e.g. `int`, `char`, `double`) has a corresponding pointer type, declared

```
int *ip, k= 3;
ip= &k;
```

This declaration helps by suppressing the (often platform-dependent) details of the data type when doing pointer arithmetic. For example, if the declaration above set `ip == 0xfff0`, what should the value of `ip + 1` be? This operation would be different for a character or long pointer.

When we want a function to change a variable (as a side effect), we generally pass the address of the variable.

```
void swap(int *a, int *b);

/* some intervening code that uses swap */
```

```
void swap(int *a, int *b)
{
    int temp= *a;
    *a= *b;
    *b= temp;
}
```

Convince yourself that `swap` couldn't be implemented if its prototype were `void swap(int a, int b)`.

There are three chapters in KN King "C programming, A Modern Approach" on pointers, and we'll be giving more information in tutorials.

September 18

C has a rather small set of data types:

- `int` represents integers, with the "natural" precision for the platform you're on (typically, but not necessarily 32 bits these days). More or less precision can be specified using the modifiers `short` or `long`. You may also specify `unsigned`, whereas `signed` is the default. A literal such as `4321` is an `int`, if you append an `L` (e.g. `4321L` it is a `long int`. The suffix `U` indicates unsigned. A leading 0 indicates an octal (base 8) integer, a leading 0x or 0X indicates a hexadecimal (base 16) integer.
- `char` is a single byte, representing a single character in the local character set. These may be `signed` or `unsigned`. A character literal is delimited by single quotes (e.g. `'x'`). Certain characters can be represented by an escape sequence, e.g. `\n` for newline, `\t` for tab. `char`s are actually small integers, and can participate in integer arithmetic.
- `double` represents a number with a fractional part with double precision (how much precision is, again, machine-dependent). More precision may be specified with `long`, less by using type `float`. Include a decimal point (e.g. `123.4`) or an exponent (e.g. `1e-2`) to indicate double, add the suffix `F` to indicate `float`

How much precision? This is specified in the header `<float.h>` for the particular machine you are on.

Variables are declared before use. The declaration statement may include initialization:

```
int i, num, k= 3;
char a= 'T', *cp;
int *ip= &k;

num= k+3;
cp= &a;
```

A declaration may have the prefix `const`, which which case warnings (may) occur if there is an attempt to change its value after it is initialized.

September 20

Arithmetic

The arithmetic operators you're familiar with from Java were (pretty much) inherited from C: `+, -, *, /, %`. In C the modulus

If you mix "narrower" and "wider" types in an arithmetic expression, the result is wider:

```
int i;
double d;
... i + d <--- double expression
```

An assignment from a wider to a narrow type is legal. It may generate a warning:

```
int i= 3;
double d= 3.14159;

... i= d <----- truncation may generate warning
```

Use a cast statement to document (and reassure the compiler) that you know what you're doing:

```
int i= 3;
double d= 3.14159;

i= (int) d;
```

Logic

Logical operators from Java were inherited from C: `>`, `>=`, `<`, `<=` have highest precedence, then come `==`, `!=`. All have precedence lower than arithmetic operators. Expressions connected with `&&` or `||` are evaluated from left to right, and evaluation stops as early as it can. An expression is negated with `!`

`if (condition) { statement(s) }` tests whether `condition` is 0 (false) or non-zero (true), and then executes `statements(s)` if it is true. `if (condition) { statement(s) } else { other statement(s) }` executes `statement(s)` if `condition` is non-zero, and `other statement(s)` if `condition` is zero. This resembles Java, once you make the leap from `boolean` to `int` conditions.

Arrays

The commonest aggregate type is an array:

```
int intArray[5], *ip;

intArray[2]= 1073;
*(intArray + 3)= 1074;
ip= &intArray[2];
... ip[1] == intArray[3] <---- this expression evaluates to 1
```

The declaration reserves 5 × `sizeof(int)` contiguous bytes of memory. The identifier `intArray`, without the subscript, contains the address of the zeroeth element. Pointers and arrays may be manipulated in similar ways. Important difference: address corresponding to an array may not be changed:

```
char word[3];

word= "hi"; <--- strings are char arrays terminated by '\0'
                 the constant array on the RHS cannot be assigned
                 to word
```

When you pass an array to a function, it sees a pointer, and it doesn't know the size of the array.

```
int arrayFunc(int *intArray) <------- same as int intArray[])
{
    int k= 2;

    return intArray[k]; <-------- same as k[intArray] --- WHY?
}
```

Another important aggregate type is `struct`. You can read some of Alan Rosenthal's notes on struct

September 25

Input/Output

If you `#include <stdio.h>` you have access to C's standard library of I/O functions. To print a formatted string to standard output (probably your console) you can use `printf`:

```
int i= 67;
double d= 97.8;

printf("Our lucky numbers are %d and %f\n", i, d);
```

The `%d` is a format specification (aka conversion specification), which tells C to replace it with the corresponding integer argument, formatted as a decimal (base 10) integer. Unexpected results if the corresponding argument is not an integer (try a character or a double). The `%f` indicates either a float or a double, and there are (many) details to `printf` that will allow you to control the number of decimal places, alignment, etc.

If you want to get input from your standard input (probably your keyboard), you can use `scanf`:

```
int i;
double d;

printf("Type your lucky integer, then your lucky double: ");
scanf("%d %lf", &i, &d);

printf("Your lucky numbers are %d and %f\n", i, d);
```

The `%d` again matches an integer, and `%lf` matches a double (that's an "l" [ell] not a "1" [one]). If you want to use all the features, you need to look up the arguments to `scanf` in a C manual. In an extreme case, type `man 3 scanf`.

You can input strings delimited by whitespace (similar to tokens):

```
char myString[10];

scanf("%s", myString); <---- why no ampersand?
```

The string to be scanned should have, at most, 9 non-whitespace characters. If you want to include whitespace characters up to a newline (inclusive) use fgets:

```
char inputLine[80];

printf("Type a line: ");
fgets(inputLine, 80, stdin);
printf("You typed: %s\n", inputLine);
```

Don't use `gets(inputLine)`, since no checking is performed if the user of your program types 81 characters...

Here's an example of I/O with some standard C types, as well as some included information about `float` and `int` that will be useful in the next topic:

```
/* fool around with I/O */
#include <stdio.h>
#include <limits.h>
#include <float.h>

int main()
{
    char c;
```

```
    int i, imax= INT_MAX, exp_min = FLT_MIN_EXP,
      exp_max = FLT_MAX_EXP, mant_dig = FLT_MANT_DIG;
    float f;

    printf("Enter a character: ");
    scanf("%c", &c);
    printf("Enter an integer: ");
    scanf("%d", &i);
    printf("Enter a float: ");
    scanf("%f", &f);

    /* what happens if you mix up the arguments? */
    printf("You entered character: %d, integer: %f, and double: %c\n",
           c, i, f);

    return 0;

}
```

How are data represented internally?

Physically data are stored in transistors that can be in an off (0, low voltage) or an on (1, or higher voltage). This allows a natural identification between collections of transistors and binary numbers (sequences of 0s and 1s, interpreted as a number base 2). You should practice a bit converting from familiar decimal (base 10) numbers to binary, and back again.

A `char` in C is represented by 8 bits (**b**inary dig**it**s), or a byte. This can represent 256 distinct values, from 0–255. C specifies that `chars` in the range 0–127 represent ascii values for characters, and the remaining values are implementation dependent.

An `int` in C is represented by at least 16 bits, often 32 bits, and occasionally 64 bits. You can figure this out by evaluating `sizeof(int)` on a particular machine, which will tell you how many bytes an `int` occupies.

Suppose you have a 32-bit `unsigned int`. These will represent values from 0–$(2^{32}-1)$, and do arithmetic modulo 2^{32} — in other words, $1 + (2^{32} - 1) == 0$.

Signed `int` (the default) is a bit trickier. The most significant bit (msb) on the left is reserved to represent the sign: 0 for non-negative and 1 for negative. Non-negative `int`s in the range 0–$(2^{31}-1)$ are represented by a 0 followed by their 31-bit binary representation. Negative ints with absolute values in the range 0–2^{31} are represented by their **two's complement** — 2^{32} minus their absolute value. This representation is efficient for hardware (essentially hardware subtraction becomes just addition).

Floats are commonly 32-bit, but you should check `sizeof(float)` on your platform. The first bit indicates the sign (1 for negative, 0 for positive). The next 32-`FLT_MANT_DIG` bits represent the exponent (the constant `FLT_MANT_DIG` is defined in `float.h`), usually to the base (aka radix) 2. Suppose we have an 8-bit exponent. Rather than being limited to exponents in the range 0..255 we shift (or bias) the exponent by subtracting 127, so we can represent exponents in the range -127..128.

The mantissa (which has `FLT_MANT_DIG` bits, or 24 bits in our example) represents the significant digits of our float as a binary number in the range $1 \leq \text{num} < 2$. The exponent is chosen so that $\text{sign} \times 2^{\text{exponent}-127} \times 1.\{\text{mantissa}\}$ equals our float. Notice that all binary numbers that are no smaller than 1 and smaller than 2 have a leading bit 1. In floating point representation, this leading 1 is omitted (it's assumed), allowing us to represent a 24-bit number with 23 bits. For a 32-bit float this gives us 1 bit for the sign, 8 bits for the exponent, and 23 bits for the mantissa.

Perhaps you're wondering: how could you ever represent 0 with a mantissa that has a leading 1? The answer is that there is one further trick. A floating point number with all zeroes in the exponent is interpreted as $2^{-126} \times 0.\{\text{mantissa}\}$. In this way, very small floats (including 0) can be represented.

September 27

Numerical computation involves constant compromising between the theoretically perfect objects we model, and the finite resources of a computer. Consider the function:

$$e^x = 1 + x + x^2/2! + x^3/3! + \cdots$$

The infinite series on the right converges to the true value. In theory you can get as close as you want to e^x by simply calculating enough terms. In practice, your wrist gets tired of the repeated multiplication and addition when you do it by hand. So use a computer.

It turns out that although computers do a fine job of calculating such a function, there are some surprising sources of error that must be guarded against.

Roundoff

Even if your computer had no other source of error in calculating e^x, it would need to fit it into some finite memory location. In what follows we'll assume (to make our calculations easier) that our computer represents floating point numbers using scientific notation and 5 decimal (base 10 digits). So, for example, $11\frac{1}{3}$ would be represented as:

$$1.1333 \times 10^1.$$

This is no different in principle from how floats are usually represented. If we used 5 binary digits instead, we'd represent $11\frac{1}{3}$ as:

$$1.0110 \times 2^3$$

This is basically the IEEE representation, except the latter uses 24 binary digits. In any case, roundoff error afflicts base 2, base 10, and any other conceivable finite representation. Here's how it works.

The decimal (aka base 10) expansion of e^1 is $2.718281828459045\ldots$ If we insist on squashing it into 5 digits, we have two reasonable choices: 2.1782 (truncation) or 2.1783 (rounding). Truncation is computationally easier, since for rounding we must examine the first omitted digit and round up if it is 5 or greater, down otherwise. How bad are our 5-digit approximations? A reasonable measure would be to find the difference between our approximation and the true value, and see what proportion of the true value the discrepancy is. This is called the relative error:

$$\text{relative error} = \frac{\text{approximate value} - \text{true value}}{\text{true value}}.$$

In our 5-digit e^1, the truncated version gives us a relative error of about 0.003%, whereas the rounded version gives a better relative error of -0.00067%. Rounding seems to be more reliable, since it's too high about half the time, too low about half, whereas truncation is always too low. So, we'll stick with rounding (as most floating point arithmetic units do).

So, what's the worst relative error we can expect from rounding? Well, if the first digit that we omit is a 5 followed by zeros, we round up and our **total** error is half the distance between two consecutive numbers. For example, if the true value were 5.43215, to keep only 5 digits we'd round up to 5.4322, and our total error would be 0.00005. Notice that the total error changes if we approximate 5.43215×10^3 by 5.4322×10^3 (the total error gets a thousand times bigger). What we really want is the relative error.

Re-write 5 as 10/2, since what's important is that 5 is half of the base (10). Also, since we're trying to find the maximum relative error, make the denominator as small as possible by having mantissa 1.0000. Then, no matter what exponent k you raise 10 to, the relative error is:

$$\frac{10/2 \times 10^{-5} \times 10^k \times 10/2}{1.0000 \times 10^k}$$

The 10^k cancels out, and you get $\frac{10^{1-5}}{2}$. The same reasoning works for bases other than 10 (just replace 10 by the symbol b for the base in the above expression. The reasoning also works if you have a different number

of digits than five (just replace 5 by t in the above expression. Then the relative error due to roundoff in a base b floating point representation with t digits is:

$$\frac{b^{1-t}}{2}$$

The number b^{1-t} is called the "machine epsilon." It is the smallest number that can be added to 1.0 to produce a different floating point number.

Floating arithmetic

Addition of large and small floating point numbers can lead to loss of information. Again, we're using base 10 with 5 significant digits. Suppose we decide to add 3.4725 *times* 10^0 and 1.1203×10^{-3}. Our floating point arithmetic unit aligns these two numbers (somewhat as you would line them up to add them on paper) by expressing them with the same exponent: 3.4725×10^0 plus 0.0011 *times* 10^0 — the second mantissa lost the significant digits 203 by being squashed into 5 digits! The sum is 3.4736×10^0.

Things would be even worse if the exponent of the second number were smaller. Suppose we added 3.4725×10^0 plus 1.1203×10^{-5}. To align the two numbers, the second one is re-normalized as, well, 0.0000×10^0 — all the significant digits fell off the right hand side. So the sum is 3.4725×10^0. Care is required when adding numbers of greatly different magnitudes.

Stranger still, floating point addition is no longer associative. The following equation says we can group addition according to taste, and the results should be the same:

$$\begin{aligned}&((((1.0000 \times 10^2 + 3.0000 \times 10^{-3}) + 3.0000 \times 10^{-3}) + 3.0000 \times 10^{-3}) + 3.0000 \times 10^{-3})\\ &\quad = (1.0000 \times 10^2 + (3.0000 \times 10^{-3} + (3.0000 \times 10^{-3} + (3.0000 \times 10^{-3} + 3.0000 \times 10^{-3}))))\end{aligned}$$

The floating point calculation on the right is 1.0000×10^2, the one on the right is 1.0001×10^2 — not quite the same.

Subtraction can have catastrophic cancelling when the numbers being subtracted are very close. Suppose the true values we were subtracting were 4.57235000 - 4.57234999. We can only keep 5 digits, and the rules say to round one up and the other down, so we end up with 4.5724 - 4.5723 = 0.0001. However, the true difference is 0.00000001, yielding a relative error of 9999, or 999,900%

October lecture summary

October 2

Course Readings, 14,15, 199–218

Last time we discussed catastrophic cancellation error when subtracting two relatively close numbers. This error can come up in a couple of ways. Suppose you decide to evaluate $\exp(-2)$ using the series:

$$\exp(-10) = 1 - 10 + 10^2/2! - 10^3/3! + 10^4/4! - \cdots$$

Notice that the later terms in this series get very small, and so does the entire series. Once you get past the first dozen terms you are repeatedly subtracting very small values (odd terms like $-10^{23}/23!$) from the partial sum, which is itself small. This gives repeated cancellation error.

Similarly, whenever you compare two floating point numbers for equality, you are vulnerable to cancellation error:

$$1 - \frac{1}{3} = \frac{2}{3}?$$

The line above is certainly an equation, but the floating-point representations of $1 - 1/3$ and $2/3$ may be close but different.

Polynomial evaluation

Suppose you had to evaluate $5x^4 + 3x^3 + 7x^2 + 2x + 9$, given $x = 7$. The straightforward approach would sum up 5*x*x*x*x*x + 3*x*x*x*x + 7*x*x*x + 2*x*x + 9 — 10 multiplications 4 additions. Although multiplication and division don't generate as much error at a single step as catastrophic cancellation (the operations are done in double precision, and then converted to float), the errors accumulate (and multiplication is computationally expensive). A better approach is to re-write the polynomial using Cramer's rule:

$$((((5x + 3)x + 7)x + 2)x + 9$$

This uses 4 multiplications (the degree of the polynomial) and 4 additions. A small re-arrangement makes a large difference (for a degree 100 polynomial, the first method would use 5050 multiplications versus 100 multiplications if re-arranged).

Find roots by bisection

A common computational job is to find the root(s) of a function. That is, for some function $f(x)$, find all the values of x that make $f(x) = 0$. If $f(x) = x^7 - 2$, the root would be $\sqrt[7]{2}$, a bit of a job to find before hand-held calculators.

The first step is to make a guess, or two. If you're lucky and guess two numbers $x_1 < x_2$ with $f(x_1) < 0$ and $f(x_2) > 0$, then you can make the following conclusion: so long as f is continuous, the interval $[x_1, x_2]$ contains at least one root. "Proof" is a picture: the graph of f is above the axis at x_1 and below the axis at x_2, so somewhere in between it must cross the axis (this "proof" of the Intermediate Value Theorem wouldn't get you through Calculus, but it will help you find a root).

The same conclusion holds if $f(x_1) > 0$ and $f(x_2) < 0$. One way to combine the two conditions is: $f(x_1) \times f(x_2) < 0$.

So, we cut the interval in half. Let x_3 be $(x_1 + x_2)/2$. There must be a root in either $[x_1, x_3]$ or $[x_3, x_2]$, so we can repeat our test: if $x_3 \neq 0$ (in which case we'd be done), then either $f(x_1) \times f(x_3) < 0$ or $f(x_3) \times f(x_2) < 0$. We can now restrict our search to half the original interval. Keep doing this until the interval is "small enough." If we decide in advance that we want to be within some tolerance factor t of the root r, then we keep cutting our interval in half until it is smaller than t. Notice that, even if t is small (for example $t = 0.00001$), the fact that $f(r) == 0$, doesn't guarantee that $f(r + t)$ is small — f may have a large slope near r.

If you took the approach that you would keep bisecting until the mid-point (call it m) of your interval had $f(m) < \epsilon$, then you might have an infinite loop. For example, consider finding the root of $x^2 - 2$ on a computer that could represent 5 decimal digits. If you bisected until you had the interval $[1.4142, 1.4143]$, then the mid-point (in 5-digit precision) would be 1.4143... With this approach, you must specify the maximum number of iterations (loops).

If you repeat this process i times, getting an interval $[x_i, y_i]$, and you return $m_i = (x_i + y_i)/2$ as your best approximation of the root r, you can be assured that $|r - m_i| \leq (y_i - x_i)/2$. If we denote the length of kth interval by e_k, then $e_{k+1} = (1/2)e_k^1$, and we say that the bisection method converges linearly (due to the exponent 1).

There are root finding methods that converge with a higher exponent than 1, but they are not guaranteed to converge, whereas bisection always converges once we've made two appropriate initial guesses.

Newton's method

A much faster, but less certain, root-finder is Newton's method. Suppose you make one initial guess at the root, x_1. You'd like to find x_2 such that $f(x_2) = 0$. If you take the first two terms of the Taylor series for f around x_1, then you have:

$$0 = f(x_2) \approx f(x_1) + f'(x_1)(x_2 - x_1).$$

Re-arranging things, and assuming that $f'(x_1)$ is non-zero, you get:

$$x_2 \approx x_1 - \frac{f(x_1)}{f'(x_1)}.$$

You decide you have to live with the approximation for x_2, and you plug it back into the formula to get x_3, and so on. So long as this method converges, it does so at a quadratic rate: $e_{k+1} \leq \alpha e_k^2$ (Course readings, p. 217). How can you tell whether it converges. A picture gives some intuition, since approximating f by its tangent is risky when the slope is near zero. Another pictorial method comes to us from chaos theory. If we call $x - (f(x))/f'(x))\ g(x)$, convergence means that $g(g(x))$ is getting closer to our root. If we draw the function $g(x)$ and the diagonal $y = x$ through the root, we can see that $g(x)$ is a **contraction** (i.e. gets closer), if $g'(x)$ has magnitude less than 1, or

$$1 > |g'(x)| = \left| \frac{f(x)f''(x)}{[f'(x)]^2} \right| \Longrightarrow [f'(x)]^2 > |f(x)f''(x)|.$$

If this holds true near the root, then Newton's method will converge at a quadratic rate.

October 4

Newton's method used the first two terms of a function's Taylor series, and omitted the others. Page 14 of the Course Notes provides a formula for the error you can expect from this omission, called the truncation error. If you omit all the terms from $f^{(n)}(x)h^n/n!$ on, your error is:

$$f^{(n)}(\theta)h^n/n! \qquad \theta \in [x, x+h]$$

Although this formula doesn't tell us exactly what θ is, it tells us its general neighbourhood. Often we can estimate the maximum size that $f^{(n)}(\theta)$ can be in the interval $[x, x+h]$, and get an idea of how bad our error can be. In the case of Newton's method, this tells us that if x_k is distance h from the root, then x_{k+1} is some constant times h^2 from the root — quadratic convergence.

However, when h is big, you might not have convergence. You might experiment with the innocent function $f(x) = x^3 - 1$ in the interval $[-2, 0]$ to find a couple of points where Newton's method fails to converge.

Numerical integration (aka quadrature)

From a user's perspective, integral calculus has a strange paradox. You're told that for any continuous function, the area under its graph (roughly its integral) exists. However, there is an exact paper-and-pencil method for finding integrals only for a tiny minority of continuous functions, and a large part of the art of calculus consists of guessing which of the various methods (integration by parts, substitution, partial fractions, ...) will work on a particular function. This sort of thing is difficult (probably impossible) to specify in an algorithm for a computer, so we use numerical methods to find a number sufficiently close to a definite integral.

The idea is to divide up the x values our function $f(x)$ is defined on into a group of intervals. Over each interval, we make an estimate of our function whose integral we already know. As we partition into smaller and smaller intervals, we get closer approximations of our desired integral.

If a typical subinterval is $[a, b]$, then the simplest approximation of the area under the graph is the rectangle with base of length $b - a$ and height of f at the midpoint. This leads to the midpoint or rectangle rule:

$$I(f) \approx (b-a) \times f\left(\frac{a+b}{2}\right).$$

A better approximation of our function might be gained by drawing a line segment between $f(a)$ and $f(b)$, called the trapezoid rule:

$$I(f) \approx (b-a) \times \frac{f(a)+f(b)}{2}.$$

If the function might be well-approximated by a quadratic curve (parabola), then we use the two endpoints and the midpoint to determine a unique parabola, and use Simpson's rule:

$$I(f) \approx (b-a) \times \frac{f(a) + 4f([a+b]/2) + f(b)}{6}$$

Which rule to use means making judgements about the nature of the function being approximated, and are beyond the scope of this brief description.

October 9

Readings for the next few lecture are from the course Readings on Graphs.

But first, `structs` and `malloc`

You already seen the aggregate type `struct` in tutorial, which allows varying data types to be grouped together at the same address. A couple of features of `structs` are now relevant.

If we want a `struct` type to be able to refer to something of its own kind, for example in a linked list, we need a declaration of the following form:

```
#include <stdlib.h>

 struct node {
    int data;
    struct node *next;
};

struct node *head = NULL;
struct node *n1, *n2, *n3;
```

The (possibly) odd feature of the declaration of `struct node` is that it includes a pointer to itself. From the point-of-view of the compiler, it ensures that `struct node` has a member that is a pointer to `struct node` before it has even completed the statement (reached the semicolon) creating `struct node`. The somewhat similar declaration replacing `struct node *next` with `struct node next` is **NOT** allowed in C: a structure cannot contain a member of the same type!

Self-referential structures have many uses. Given our declarations above, we could now set up a small linked list:

```
n1 = (struct node *) malloc(sizeof(struct node));
n2 = (struct node *) malloc(sizeof(struct node));
n3 = (struct node *) malloc(sizeof(struct node));

head = n1;
n1->data = 1;
n1->next = n2;
n2->data = 2;
n2->next = n3;
n3->data = 3;
n3->next = NULL; /* <-- indicates end of list */
```

`malloc` allocates `sizeof(struct node)` bytes, and returns a void pointer to it, which we cast to `struct node *`. Under some conditions `malloc` could fail to allocate the required space, in which case it returns the special address `NULL`. We don't check for (nor act on) this possibility, but in A2 `emalloc` does.

The fact that C allows us to declare a pointer to a `struct` whose members have not yet been defined allows other flexibility. For example, we can use `typedef` to declare a type that points to `struct graph`, before `struct graph` itself has been defined, as the following declaration in A2's `graph.h` does:

```
/*
 * "Graph" abstract data type.
 */

typedef struct graph *GRAPH;
```

This allows the user interface in `graph.h` to declare functions with type `GRAPH` as parameters or return type, while the actual implementation of `struct graph` is left to the module defined in `graph.c`, for example

```
/* Colour the graph. */
extern void colour_graph(GRAPH g);
```

Graph definitions

The intuitive notion of a graph is a drawing of (possibly labeled) nodes (often circles) with edges connecting some of them. Graphs turn out to be a model that many problems can be translated to, and often solved. You've already seen a special case of a graph, since a tree is an acyclic, connected graph with one node distinguished as the root (see definitions below).

Here are some formal definitions:

- A graph (denoted G) is a pair of sets (V, E), where V is the set of vertices (or nodes) and E is the set of edges. Vertices are usually the elements of the problem we're interested in. Each edge connects two vertices, so it can be represented as a pair (v_1, v_2).
- We can also denote the vertices of a graph G as $V(G)$ and its edges as $E(G)$. The number of vertices, $|V(G)|$ is the called G's **order**, and the number of edges, $|V(E)|$ is called G's **size**.
- Since edges are ordered pairs, (v_1, v_2) indicates there is an edge from v_1 to v_2, but not necessarily from v_2 to v_1 — if so there must be an edge (v_2, v_1) in E. When this distinction is important, we have a **directed graph**. In this course the edge relationship is usually symmetrical: there is an edge (v_1, v_2) if and only if there is an edge (v_2, v_1). In this case the edge is in fact the set $\{v_1, v_2\}$ (since sets have no order), and we have an **undirected graph**. A vertex may not have an edge to itself (unless we have a **pseudograph**).
- When two vertices share an edge they are "adjacent" (or "neighbours"). The number of neighbours vertex v has is called the **degree** of v (for a directed graph we have in-degree and out-degree). A **path** is a list of vertices where each pair of vertices that are adjacent in the list are also adjacent in the graph.
- A **circuit** is a path that begins and ends on the same vertex without repeating any edges. A **cycle** is a path that begins and ends on the same vertex and doesn't repeat any vertices (and usually includes at least three vertices).
- Vertices v_1 and v_2 are **connected** if there is a path beginning with v_1 and ending with v_2. Graph G is connected if every pair of vertices in G is connected.
- Graph G is **complete** if every vertex is a neighbour of every other vertex. Clearly a complete graph is also connected. If G is a complete, undirected graph, how many edges does it have? It turns out that two complete graphs of the same order (same number of vertices) are equivalent (isomorphic, which is defined later), so it makes sense to talk about **the** complete graph of order 2, K_2, or order 4, K_4, etc.
- A colouring is an assignment of colours to a graph so that
  1. every vertex is assigned a colour.
  2. no two neighbouring vertices are assigned the same colour

  Usually we represent the colours by small integers. If it is possible to colour a graph with k colours, we say the graph is "k colourable." If you know that it's possible with colour a graph with k different colours, then you can certainly colour it with more than k colours: being k colourable doesn't require that all k colours are used, only that k colours are enough.
- The chromatic number of graph G, denoted $\chi(G)$, (that's the greek letter chi) is the minimum number of colours required to colour G. It's easy to come up with an upper bound for $\chi(G)$, since you G is certainly k colourable if $k = |V(G)|$. Usually $\chi(G)$ is less than $|V(G)|$, but in the case of a complete graph G, $\chi(G) = |V(G)|$.
- To prove that $\chi(G) = k$ for a given graph G, you need to prove both that G is **not** $(k-1)$ colourable, and that G **is** k colourable.

October 11

Graph isomorphism

Suppose I have two undirected graphs, each with 3 vertices and 3 edges: G_1 has $V(G_1) = \{a, b, c\}$ and $E(G_1) = \{(a, b), (a, c), (b, c)\}$, and G_2 has $V(G_2) = \{d, e, f\}$, and $E(G_2) = \{(d, e), (d, f), (e, f)\}$. If you draw these graphs, it's pretty clear that they are equivalent: you get G_2 by changing the vertex labels a, b, c on G_1 to d, e, f. But they aren't equal, since their vertex sets and edge sets are different (in a superficial way).

We capture the idea that G_1 and G_2 are equivalent except for their labeling by saying they are **isomorphic**. Two graphs G_1 and G_2 are isomorphic if there is a bijection f such that $f(V(G_1)) = V(G_2)$, $f^{-1}(V(G_2)) = V(G_1)$ **and** $f(E(G_1)) = E(G_2)$. What all the symbols mean is that if you can find a way of matching each vertex of G_1 with a corresponding vertex of G_2 (this matching is your bijection f), and if your matching means that the edges of G_1 are mapped to the edges of G_2, then G_1 and G_2 are isomorphic. Another way of saying the same thing is that v_1 and v_2 are neighbours in G_1 if and only if $f(v_1)$ and $f(v_2)$ are neighbours.

We consider graphs **up to isomorphism**, that is we consider isomorphic graphs to be equivalent, and a property that we prove about one is true of the other. So we talk about **the** complete graph of order 5, K_5. Even though there are many isomorphisms, they are all equivalent.

Graph representation

Two common ways to represent graphs on a computer are as an adjacency list or as an adjacency matrix.

Adjacency list: Vertices are labelled (or re-labelled) from 0 to $|V(G)| - 1$. Corresponding to each vertex is a list (either an array or linked list) of its neighbours.

Adjacency matrix: Vertices are labelled (or re-labelled) with integers from 0 to $|V(G)| - 1$. A two-dimensional boolean array A with dimensions $|V(G)| \times |V(G)|$ contains a 1 at $A[i][j]$ if there is an edge from the vertex labelled i to the vertex labelled j,and a 0 otherwise.

Both representations allow us to represent directed graphs, since we can have an edge from v_i to v_j, but lack one from v_j to v_i. To represent undirected graphs, we simply make sure that are edges are listed twice: once from v_i to v_j, and once from v_j to v_i.

Which representation is best? Both. If graph G has a large portion of its edges, then the adjacency matrix doesn't waste much space, and it indicates whether edge (i, j) exists with one access (rather than following a list). However, if graph G is sparse (not many of its vertex pairs have edges between them), then an adjacency list becomes preferable. For example, if G has 10,000 vertices and only about 20,000 edges, then its adjacency matrix representation will need 10^8 (100 million) entries — 400 megabytes if each took a word. Representing the same G with an adjacency list might require, say, 10,000 words for the node plus 20,000 words for the list of neighbours: 30,000 words or 120K. The difference may make one representation feasible and the other infeasible.

Depth-first search (DFS)

There are various ways to traverse (visit all the nodes) of a graph systematically. A couple of these ways (depth-first and breadth-first) give us some information about graph structure (e.g. connectedness).

In depth-first search the idea is to travel as deep as possible from neighbour to neighbour before backtracking. What determines how deep is possible is that you must follow edges, and you don't visit any vertex twice.

To do this properly we need to keep track of which vertices have already been visited, plus how we got to (the path to...) where we currently are, so that we can backtrack. We could keep track of which nodes were visited in a boolean array, and a stack to push nodes onto that we mean to visit (the course Readings have a recursive algorithm for DFS which takes a slightly different approach). Here's some pseudocode:

```
DFS(G,v)   ( v is the vertex where the search starts )
   Stack S := {};   ( start with an empty stack )
```

```
    for each vertex u, set visited[u] := false;
    push S, v;
    while (S is not empty) do
       u := pop S;
       if (not visited[u]) then
          visited[u] := true;
          for each unvisited neighbour w of u
             push S, w;
       end if
    end while
END DFS()
```

It would probably be useful to keep track of the edges we used to visit the vertices, since these edges would span the vertices visited. One way to do this is with another array `predecessor[u]` which indicates which vertex u was reached from. When we are processing the neighbours of, say, vertex u, for each neighbour (say v) of u that we push onto the stack, we set `predecessor[v]` to u. Eventually we end up with a **tree**: an acyclic, connected graph of all the vertices that can be reached from our starting point.

What happens if our original graph G isn't connected? Then `DFS(G,v)` won't visit any vertices that aren't connected to its starting point. You'll need an outer loop that iterates over unvisited vertices, and then calls `DFS(G,V)`.

The end result is a **forest** (a collection of trees) representing the connected components of G.

Breadth-first search

Breadth-first search means we visit all of vertex v's neighbours before we visit the neighbours' neighbours. One way to achieve this is to add all of v's neighbours to a queue, and then visit each element of the queue(adding that element's neighbours to the tail of the queue) in FIFO order. There is pseudocode for this algorithm in the Course readings that numbers each vertex with a unique number according to when it is visited (as does the Readings' DFS algorithm).

October 16

Earlier we noted that a path from v_1 to itself is called a circuit (sometimes simple circuit) if it repeats no edges, and a cycle (sometimes simple cycle) if it repeats no vertices except v_1. A graph is **acyclic** if it contains no cycles with 2 or more edges. A connected, acyclic graph with a distinguished vertex (a vertex that we choose and keep track of) is called a **tree** (the distinguished node is the **root**).

One sort of information that can be added to a graph is edge weight: with each edge (v_i, v_j), associate a number. Various applications might use **weighted graph** to represent properties, for example inter-city distances, average packet time between internet hosts, etc.

Our two graph representations can be coerced into adding this information: an adjacency matrix can store the edge weight for edge (v_i, v_j) in `matrix[i][j]`. In this scheme, `matrix[i][i]` would be 0, and `matrix[i][j]` could contain some special value (say ∞) if there is no edge (v_i, v_j). Similarly, an adjacency list could add a member to each list element to record the edge weight.

Shortest distance

A reasonable question to have about a weighted graph is "what's the shortest distance from A to B?" (assuming that A and B are nodes on this graph). Translating the question into graph theory, this means that if G is a weighted graph with vertices u and v, we want to know which path from u to v has the minimum cumulative weight (minimum sum of edge weights in the path). For example, if G were a weighted graph representing cities in southern Ontario and Quebec, with the weights being road distances between cities, then it would be reasonable to ask what the shortest distance from Toronto to Montreal is (probably not taking the QEW to Hamilton, the 403 to London, and then the 401 to Montreal).

Some conditions need to be set. Although we may be able to tolerate negative weights on some edges of G, negative cycles (even a cycle with only 2 vertices) are not allowed (or else there would be no minimum!).

Notice that this means that any undirected graph with a negative weight, say $w(i,j) = -1$, has a negative cycle (you can step back and forth from i to j and pick up a negative one each time).

Here's an algorithm that works for non-negative weights, by Edgster Dijkstra:

Dijkstra's algorithm

```
DIJKSTRA ( V, E, s )
   for each v in V
      d[v] := oo;   ("infinity")
      pred[v] := NULL;
   end for
   d[s] := 0;
   S := {};
   V' := V;
   while ( V' is not empty ) do
      find a vertex u in V' such that d[u] is minimum;
      V' := V' - {u};
      S := S U {u};
      for each edge e = (u,v) in E
         if ( v is not in S ) and ( d[v] > d[u] + w(u,v) ) then
            d[v] := d[u] + w(u,v);
            pred[v] := u;
         end if
      end for
   end while
END
```

At each iteration of the while loop we add a vertex u to the solution set S, and never change its distance ($d[u]$) again. You should worry that, perhaps, there is some shorter distance from s to u using some vertex that hasn't yet been added to S. Convince yourself that this can't happen.

October 18

Floyd-Warshall all-pairs shortest path

Here's an algorithm that can tolerate negative edges (but not negative cycles, even with only 2 vertices) and returns a table of the shortest distance between all pairs of vertices.

The idea is to first find all the minimal distances for pairs without using any intermediate vertices (this is easy to calculate, since it will be ∞ if no edge exists between the pair, the edge weight otherwise). We then relax the restriction, step-by-step: first allow paths that use node 0 as an intermediate, and see whether the minimum distances can be adjusted downwards, then paths that are allowed to use nodes 0 and 1, and so on until all paths are considered. We denote the minimum distance from v_i to v_j that considers intermediate nodes from the set $\{0, \ldots, k\}$ as $D[i,j,k]$ (special case, $D[i,j,-1]$ doesn't allow any intermediate nodes).

Now we have a way of building up results. If we already know $D[i,j,m]$, for $m \in \{0, \ldots, k-1\}$, and every pair (i,j), then the shortest path from i to j either passes through k or it doesn't. In the first case, the portion of this minimal path from i to k is itself minimal and doesn't use k (convince your self of this), as is the portion from k to j, so $D[i,j,k] = D[i,k,k-1] + D[k,j,k-1]$. In the second case, $D[i,j,k]$ equals $D[i,j,k-1]$.

Since we can easily calculate $D[i,j,-1]$ for all pairs (i,j), we can build up to $D[i,j,n]$ (where $n = |V|$) as follows (the third parameter is suppressed, but corresponds to the outer loop parameter k in the bottom triply-nested loop):

```
FLOYD-WARSHALL( V, E )
   for  i := 0  to  n-1  do
      for  j := 0  to  n-1  do
```

```
            if  i == j  then
               D[i,j] := 0;
            else if  (i,j) is in E  then
               D[i,j] := w(i,j); /* <-- weight function w */
            else
               D[i,j] := oo;
            end if
         end for
      end for
      for  k := 0  to  n-1  do
         for  i := 0  to  n-1  do
            for  j := 0  to  n-1  do
               if  D[i,k] + D[k,j] < D[i,j]  then
                  D[i,j] = D[i,k] + D[k,j];
               end if
            end for
         end for
      end for
   END
```

Here's something to worry about. How do you know that $D[i,k]$ and $D[k,j]$ correspond to $D[i,k,k-1]$, $D[k,j,k-1]$, and haven't been updated with minimum distances that pass through k?

An easy calculation shows that Floyd-Warshall has $O(n^3)$ complexity, which is consistent with using Dijkstra n times (and much simpler). How would you recover a shortest path, given this table of distances?

October 23

Paradox lost

In class last time I presented a small undirected graph that defeated Dijkstra and (I claimed) would work with Floyd-Warshall. Unfortunately, the graph contained a negative cycle (a trivial one that got under my radar), hence the shortest-path problem is not well-defined. The fix is to make it a directed graph. The directed graph both defeats Dijkstra and works for Floyd-Warshall. Here are the resulting adjacency matrices (the cases $k = 0$ and $k = 3$ are omitted, the first because vertex 0 has no incoming edges, the second because vertex 3 has no outgoing edges).

$k = -1$

	0	1	2	3
0	∞	10	∞	∞
1	∞	∞	4	3
2	∞	∞	∞	-6
3	∞	∞	∞	∞

$k = 1$

	0	1	2	3
0	∞	10	14	13
1	∞	∞	4	3
2	∞	∞	∞	-6
3	∞	∞	∞	∞

$k = 2$

	0	1	2	3
0	∞	10	14	8
1	∞	∞	4	-2
2	∞	∞	∞	-6
3	∞	∞	∞	∞

Graph problems often need to find out whether graph G contains a cycle. One way to answer this question is to use modify DFS (Readings 388) to see whether our search ever re-visits a vertex. Note that the modification in the course Readings considers two vertices with an undirected edge between them to be a cycle. Usually we're interested only in cycles with 3 or more distinct vertices, so this algorithm would have to be modified.

This approach runs into problems with a directed graph. Since, in a directed graph there may be a path from v_1 to v_2, but not from v_2 to v_1, the fact that v_2 has already been visited doesn't necessarily indicate a cycle. Consider the example from Readings p. 376 (Figure 8.4(b)). Although a has already been visited, and found all the edges "downstream" from it, there is an edge from g to a that the simple DFS modification would flag as a cycle. We need to set a special value (perhaps ∞) if a node has been visited **and** all nodes reachable from it have been visited. Another way to do the same thing is to record arrival/departure numbers.

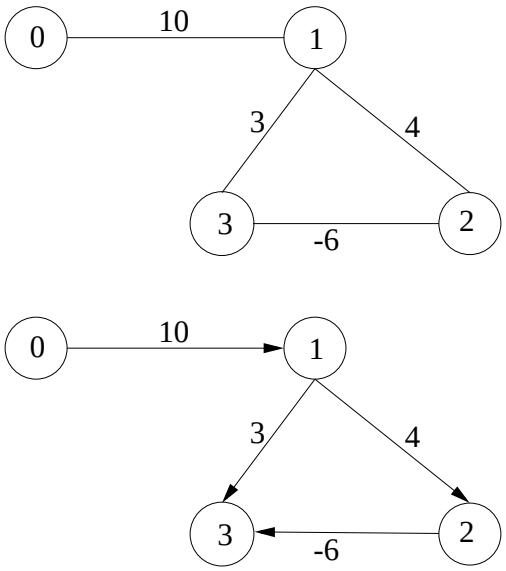


Figure 1: The top (undirected) graph doesn't have a well-defined shortest path between pairs, because you can hop back and forth between nodes 2 and 3, making any path as short as you like. The bottom (directed) graph defeats Dijkstra's algorithm (with 0 as the source), but is okay with Floyd-Warshal

October 25

CPP macros and conditional compilation

Earlier in the course we saw that the C pre-processor performs textual substitution. The preprocessor can do more than this, it can conditionally compile (or ignore) portions of code, and it can expand macros to behave like functions (with no compiler checking of parameter and return types). Many standard library functions are implemented as macros. A useful example is the `assert` function, declared in `assert.h`. If your code contains a statement such as:

```
assert(n > 0);
```

...stating some condition you intend to be true at all times, then if the condition is ever false your program exits, prints the file and line number, plus stating that the condition failed.

You can create your own macros. The following example defines `DD1` to print the file, line number, and then a formatted string (argument `a`) that uses an argument (`b`). This version of `DD1` is defined only if `DEBUG` is defined, otherwise `DD1` becomes an empty macro (resulting in an empty, but legal, statement). When compiling you can make sure `DEBUG` is defined by including `-DDEBUG` in the `gcc` command (e.g. `gcc -DDEBUG myprog.c`

```
/* experiment a bit with macros */
#include <stdio.h>

#ifdef DEBUG
#define DD1(a,b) printf("%s: %d " a, __FILE__, __LINE__, b);
#else
#define DD1(a,b)
#endif

int main()
{
     int i= 5;
```

```
#ifdef DEBUG
     printf("Debugging on!\n");
#else
     printf("Debugging off!\n");
#endif
     DD1("The value of i is %d\n", i);

     return 0;
}
```

If the replacement list for a macro contains an operator (e.g. "*"), you should enclose the list with parentheses. Similarly, if the arguments for the macro are expressions, they should be enclosed in parentheses (See K.N. King, chapter 14). Here's an example

```
#ifdef CARELESS
    #define MAX(x,y)  x>y?x:y
#else
    #define MAX(x,y)   ((x)>(y)?(x):(y))
#endif
```

With `CARELESS` defined, you might get surprising results from, say `3 * MAX(2,3)`.

Next Monday's tutorial will give you more details about preprocessor and conditional compilation.

C++ overview

The last two assignments in this course will be implemented in C++. We don't teach C++ in lecture (although there will be a number of tutorials dealing with it) so you'll need a manual (consider Stroustrup's *The C++ Programming Language* or Satir and Brown *C++, the Core Language*.

Having said that, let's consider the differences and similarities between C and C++.

Some similarities:

- C++ is an extension (superset) of C: most C programs are also C++ programs.
- Basic data types (`int, char, float, void`) are the same in C and C++
- You declare variables and functions in the same way.
- A program still has `main()` outside any class, which is the starting point for execution. There may be other functions outside any class.
- preprocessor directives are inherited from C (`#include`, etcetera).

Some differences:

- C++ has an additional streamed I/O facilitiy (`cout, cerr` etc.). We won't go there, but keep using `stdio.h`
- dynamic memory allocation adds features to make it easier.
- C++ has classes
- C++ allows the `//` form of comment (lasts until the end of the currentline).
- `struct foo ...;` creates a type `foo` — declarations `foo a;` and `struct foo a;` both work.
- You can mix declarations and other statements.

Dynamic allocation

In addition to C's `malloc`-like functions, and `free`, C++ has `new` and `delete`. These can provide convenient dynamic allocation. There is no garbage collection.

```
int *a, *b;

a = new int;  // new provides a pointer
*a = 100;
b = new int[*a]; // array of 100 ints
b[3] = 7;

delete a; // frees the allocated memory
delete [] b; // weird, eh?  frees the 100 ints.
```

Classes

One early name for C++ is "C with classes" (there are other Cs with classes, e.g. Objective C). The ability to group data and behaviour (functions) into a class should be familiar from java, and exists in C++. Also, the ability to inherit behaviour and data from base classes exists in C++. Classes look a lot like C structures (even including the final semicolon) but

- They can contain functions (not just pointers to functions) that "know" about their data (variables)
- they can hide both data and functions

Here's a C++ stack class:

```
class stack
{
public:
    void push(int x);
    int pop();
    int empty();
    stack() {size = 0;}

private:
    int size;
    int a[100];
};
```

Notice the final semicolon!

The class stack mainly contains declarations of functions except for the constructor, `stack()`. Conventionally only small functions are implemented inside the class, since each instance of the class will contain in-line code for functions defined between the left and right braces. Most of the functions are defined in the **namespace** of the class:

```
bool stack::empty()
{
    return size == 0;
}
```

October 30

In order to predict the outcome of processes we usually model them, that is we construct some ideal representation that concentrates on the important details and neglects the others. For example, to figure out when a body in free-fall will hit the ground we construct a model that describes the behaviour of gravity,

air resistance, and other important factors but neglect the (probably small) effect that the body's choice of clothing might have on their descent.

Sometimes we can apply analytical solutions, for example an equation or system of equations that can be solved in a few steps, to make predictions. To get information from a system where we lack an analytic solution we need to **simulate** or model the system's components: the entities in the system, the relationships between them, and events that change the state of the system.

Simulations come in a few varieties. Discrete simulations have entities that take on finitely many states, continuous simulations allow their entities to range over infinitely many values. Static simulations don't have time as a variable, dynamic simulations do have time as a variable. Deterministic simulations apply rules in the same way each time, stochastic simulations have a random component to their application of some rules. Here are some examples:

1. Randomized integration: could be discrete or continuous, static, and stochastic (perhaps faster than the numerical methods learned).
2. Simulate the trajectory of a projectile: continuous, dynamic, stochastic (perhaps easier than solving all the related equations).

For this course we'll concentrate on dynamic, discrete, probablistic simulations. We'll need some way of keeping track of the simulation time, and generating events at various times. There are a couple of ways of doing this.

Time-driven simulation

In a time-driven simulation we have a variable recording the current time, which is incremented in fixed steps. After each increment we check to see which events may happen at the current time point, and handle those that do. For example, suppose we want to simulate the trajectory of a projectile. At time zero we assign it an initial position and velocity. At each time step we calculate a new position and velocity using the forces acting on the projectile. Time-driven simulation is suitable here because there is an event (movement) that happens at each time step.

How do know when to stop the simulation? We can use either the criterion of time reaching a certain point, or the model reaching a certain state, or some combination of the two.

Here's a general algorithm for time-driven simulation:

1. Initialize the system state and simulation time
2. while (simulation is not finished)
   (a) Collect statistics about the current state
   (b) handle events that occurred between last step and now
   (c) Increment simulation time

November lecture summary

November 6

Event-driven simulation

If events aren't guaranteed to occur at regular intervals, and we don't have a good bound on the time step (it shouldn't be so small as to make the simulation run too long, nor so large as to make the number of events unmanageable), then it's more appropriate to use an event-driven simulation. A typical example might be simulating a lineup at a bank, where customers don't arrive at regular time intervals, and may be deterred by a long lineup.

This approach uses a list of events that occur at various time, and handles them in order of increasing time. Handling an event may alter the list of later events. The simulation makes time "jump" to the time of the next event.

How do we stop? Again, we can stop when time reaches a certain point, or when the system reaches a certain state. Here is a generic event-driven algorithm:

1. Initialize system state
2. Initialize event list
3. While (simulation not finished)
   (a) Collect statistics from current state
   (b) Remove first event from list, handle it
   (c) Set time to the time of this event.

How is the list of events managed? It should be ordered by increasing time (a priority heap might be efficient). We don't generate all the events in the list at the beginning (this would be analogous to knowing the entire sequence of states of the simulation at the outset). Instead we initialize the simulation with certain events, with their associated times. Certain events may be handled by scheduling later events, which are inserted at the appropriate place in the event list.

As stated above, we could stop when time reaches or exceeds a certain point, or once the system reaches a certain state (the bank is lined up for two blocks...). Sometimes we want the stopping condition itself to be randomized: we can schedule a random pseudo-event which doesn't change the state of the model, but simply stops the simulation.

Barber simulation

A single-chair barbershop. From when it opens in the morning until it closes, customers arrive at random times. If the barber is not busy, he serves a customer immediately, otherwise they must wait in a queue (FIFO order). The time needed to serve each customer is also random.

The entities are:

- Server (idle or busy)
- customer (arrival and service times)

The events:

- customer arrives
- customer departs

The arrival and departure events encapsulate everything we care to know about a customer, so we don't **need** to track customers explicitly.

System state:

- Simulation time (starts at 0)
- event list (starts empty)
- server status (initially idle)
- customer queue (initial value empty)

The first two elements are properly part of the simulation program, rather than the system we're modelling. The last two are part of the barbershop simulation. There is always a single event list, since it represents the flow of time and all events end up there. There may be multiple event lists, depending on the system we're modelling.

Initialization:

- Schedule the first customer

Arrival event:

- If the server is idle, start service immediately (change server status to busy and schedule an end of service event). Otherwise the customer waits in queue (in some models they may "balk" and decide they don't have time to wait).
- Schedule the next customer arrival at random, given the desired distribution (more about this later).

Departure event:

- Change server status to idle
- If the customer queue is not empty, start service on the first customer in the queue (change the server status back to busy and schedule an end of service event).

Statistics

- Average delay: keep track of how long each customer waits (even if the waiting time is zero). At the end of the simulation, compute the average waiting time per customer (assuming ¿= 1 customer).
- Average server utilization: the fraction of the total time that the server was busy. Add up all the time the server was busy, divided by the total simulation time.
- Average number of customers in the queue: this could be a weighted sum: 0*(fraction of time the queue is empty) + 1*(fraction of time the queue has 1 person) + 2*(fraction of time the queue has 2 people) + ...
- To compute all of these statistics, we need to keep track of the following additional quantities during the simulation (the total simulation time is already stored in the system state): the total number of customers for the duration of the simulation, the total time spent waiting for all customers, the total amount of time that the server was busy, and the weighted sum of the number of customers waiting in the queue.
- At the end of the simulation, the server finishes working the the current request (if any), ignores anyone in the waiting list, except to record the size of the waiting list. No further statistics about the length of delay are collected, since this is unknowable (everybody goes home).

October 8

In many simulations we want a value to be calculated "randomly," that is chosen from some distribution of values where each particularly choice is unknowable in advance. Since we're simulating this on a computer where there has to be **some** rule for calculating the values, we usually use a pseudo-random sequence of values that obey our desired distribution of values, and instead of "unknowable" we settle for hard-to-know in advance.

Some of the distributions we simulate are

uniform: Each value in some range has an equal chance of occurring (for example we expect each number on a single fair die to occur in 1/6 of the rolls)

normal: Two or more independent events are summed (e.g. two dice), yielding a familiar bell-shaped curve.

exponential: The range of values is infinite, but we determine in advance what the mean (average) value is. Streetcars may arrive every 10 minutes on average, but this includes cases where five arrive bunched together plus the cases when equipment failure suspends service for several hours.

As well as these distributions we distinguish **discrete** distributions (there is a finite set of possible values), and **continuous** distributions (infinitely many values along some segment of the real line).

We describe probability that a value, or range of values, will occur by inventing a random variable X, a probability density function $f_X(x)$ indicating the likelihood that X will be x (note the upper/lower case distinction), setting $f_X(x)$ to some value in $[0, 1]$, and insisting that the sum of $f_X(x)$ for all x is 1. If $X == x$ is impossible, then $f_X(x)$ is zero. If $X == x$ is certain, then $f_X(x)$ is 1. Most cases are between these extremes.

Discrete distributions

Suppose our random variable X can take only values $v_1, v_2, \ldots, v_N$. We insist that $0 \leq f_X(v_i) \leq 1$, and

$$\sum_{i=1}^{N} f_X(v_i) = 1.$$

If each v_i is equally likely to occur, then these conditions lead to a uniform distribution with $f_X(v_i) == 1/N$. A binomial distribution expresses how often we expect a particular value to occur out of n events. If $f_X(e) = p$ (the probability that our chosen event occurs once is p), then the probability of exactly k occurrences is:

$$f_X(k) = \binom{n}{k} p^k (1-p)^{n-k}.$$

The reasoning is that in a sequence of n events there are $n$choose$k$ ways to select exactly k event es, and the probability of each selection is the product of the ps with the product of the $(1-p)$s (the probability that e doesn't occur).

This is a discrete normal distribution.

Continuous distributions

Things are slightly less intuitive when our event can be selected from a segment of points on the real line. For example, I usually tell anyone who cares that I'm 166 centimeters tall. However, I'm sure that a precise enough measure would determine that I'm not **exactly** 166.000000... cm tall. Adult human heights are distributed over some interval between 1 meter and 3 meters (to be generous), but the chance that some particular exact height is achieved is basically zero.

So we change our point of view and talk about the probability that our value falls within some particular range. We define $f_X(x)$ so that its values are always non-negative and its integral from a to b represents the probability that X takes a value in $[a, b]$:

$$\int_a^b f_X(t)dt = \text{probability that } X \text{ is in } [a, b].$$

From the continuous point of view, the uniform distribution over the interval $[a, b]$ needs an f_X whose integral is 1 and whose value for any two equally-wide subintervals of $[a, b]$ is the same:

$$f_X(x) = \begin{cases} 1/(b-a) & x \in [a, b] \\ 0 & \text{otherwise} \end{cases}$$

Suppose you want a distribution that models the distribution of times between events that happen at rate r (reciprocal of mean time between events), but you have no upper limit on the maximum possible time. A distribution that works is the exponential distribution:

$$f_X(t) = \begin{cases} re^{-rt} & t \geq 0, \\ 0 & \text{otherwise} \end{cases}.$$

Generating a distribution

A subtle point is that f_X describes the distribution we want, but it doesn't directly help create it. Programming languages often provide a way of generating a sequence of pseudo-random integers in the range $0 \ldots m$ by starting at some integer x_0 and then continuing using the relation:

$$x_{n+1} = (ax_n + b) \bmod m.$$

With a good choice of a, b, and m this will generate a sequence of integers that satisfy some criteria of randomness, and not repeat any for m steps. If m is chosen fairly large, we can divide by it to scale these

values into $[0,1)$ (the half-open interval from zero to 1). But there's more to do to generate our desired distributions.

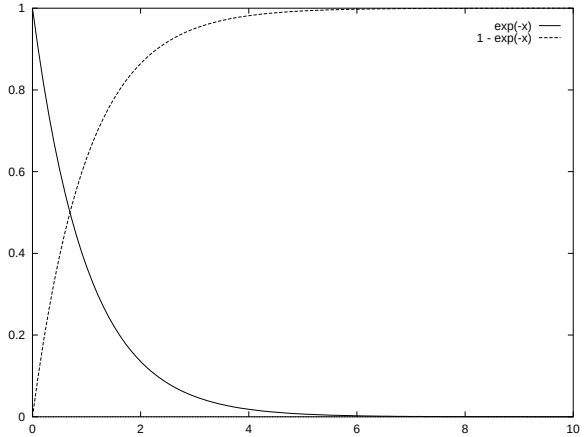
We need a way to associate each number we generate in $[0,1)$ with a number in our desired distribution. The first step is to construct the cumulative distribution function $F_X(x)$ which represents the probability that $X \leq x$:

$$F_X(x) = \int_{-\infty}^{x} f_X(t)dt.$$

F_X has the nice property that its values are increasing and in the interval $[0,1]$. If we apply it to our exponential distribution, we get:

$$F_X(x) = \int_0^x re^{-rt}dt = r(e^{-rt}/(-r))\Big|_0^x = 1 - e^{-rx}.$$

Here's the picture:



Now we can **invert** $F_X(x)$ — turn the graph sideways! For each $y \in [0,1)$ we generate, we can solve $y = 1 - e^{-rx}$ in order to find the corresponding x. Rearrange the equation and take ln (natural log) of both sides, and you have $x = -\ln(1-y)/r$. We don't have to worry about taking $\ln(0)$, since we rigged things so that y is never 1. A more straight-forward task is to invert a uniform continuous distribution on the interval $[a,b]$. If $x \in [a,b]$ the cumulative distribution is $F_X(x) = (x-a)/(b-a)$, so now we solve for $y = (x-a)/(b-a)$, or $x = a + y(b-a)$.

November 13

Dynamic Programming

You've already seen problems that can be reduced to subproblems with the same structure — recursion comes to mind. A classical sequence whose value is expressed recursively is the Fibonacci sequence:

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}.$$

Given this recursive definition (a recurrence relation), it is very efficient in programmer labour to write down an algorithm:

```
int fib(int n)
{
    if (n < 2) {
        return n;
```

```
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```

However, for fairly modest n this algorithm turns out to be inefficient for the computer. Consider `fib(8)`: `fib(7)` gets called once, `fib(6)` gets called twice, ...the complexity ends up being exponential.

A more reasonable way to calculate `fib(n)` is from the bottom up:

```
int fib(int n)
{
    int *f, answer;
    if (n < 2)
        return n;
    f = new int[n+1];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];
    answer = f[n];
    delete f;
    return answer;
}
```

Even given the `new/free` operations to allocate and free memory, this is a much better algorithm, being $O(n)$. It is also an example of Dynamic Programming.

Another algorithm that fits the same mold is the Floyd-Warshall all-pairs shortest distance. At the heart of that algorithm was the following recurrence:

$$D[i,j,k] = \min\{D[i,j,k-1], D[i,k,k-1] + D[k,j,k-1]\}.$$

If you were to use recursion to solve this problem, you'd need two recursive calls to evaluate the minimum. Each of those would require two recursive calls ... and you'd have exponential complexity.

The Dynamic Programming approach would be to start at the bottom: $D[i,j,-1]$ is defined to be the weight of edge (i,j), and we can easily build up a three-dimensional array starting from these values, and using the recurrence. The solution presented in our course readings uses a two-dimensional array (saving space), although the algorithm is still $O(n^3)$.

What are the principles that tie these approaches together under the heading Dynamic Programming? Given a problem that you want to find an optimal solution for:

1. Characterize the optimal subproblem structure.
2. Define an array (table) that contains the value you want to optimize for, plus all the relevant subproblems.
3. Write down a recursive definition for building the array.
4. Compute the solution from known values.

Knapsack problem

Your knapsack (or possibly your back) has the capacity to hold C kilograms (C is a nonnegative integer). You are presented with a collection of n categories of goodies that are free for the taking, category i items have weight w_i and value v_i (weights and values also nonnegative integers). Fill your knapsack so that you can carry off (C is not exceeded) the maximum value. Since you're allowed to take more than one item of the same type, you can express your answer as a list of indices $i_1, i_2, \ldots, i_k$ such that $w_{i_1} + \cdots + w_{i_k} \leq C$, and you maximize $v_{i_1} + \cdots + v_{i_k}$.

Here's an example you may be able to solve without any high-powered algorithm. Suppose $C = 11$, $w_1 = 3$, $w_2 = 4$, $w_3 = 5$, $w_4 = 8$, and $v_1 = 2$, $v_2 = 3$, $v_3 = 4$, $v_4 = 5$.

Solving this using greedy algorithms (add the most valuable item that fits, etc.) doesn't yield an optimal solution. Follow the DP approach to try to identify how an optimal solution is related to optimal solutions for subproblems.

If $i_1, \ldots, i_{k-1}, i_k$ is an optimal solution, then we can at least say that $i_i, \ldots, i_{k-1}$ must be an optimal solution for a knapsack with capacity $C - w_{i_k}$. This suggests we can define an array $V[]$, where $V[j]$ is the maximum value that can be carried in a knapsack with capacity j. If we can find $V[C]$, then we're done. What we already know yields a recurrence:

$$V[n] = \begin{cases} 0 & n = 0 \\ \max\{V[j-1], \max_{w_i \le j}\{v_i + V[j - w_i]\}\} & \text{otherwise} \end{cases}.$$

If you use the straight-forward approach of writing a recurrence program, you will have an extremely expensive (in running time) algorithm. Instead, compute the values bottom-up as follows (`C` is as above, `v[i]` is v_i, and `w[i]` is w_i):

```
V[0] = 0;
for (int j = 1; j <= C; j++) {
    V[j] = V[j-1];
    for (int i = 0; i < n; i++) {
        if (w[i] <= j && v[i] + V[j-w[i]] > V[j]) {
            V[j] = v[i] + V[j-w[i]];
        }
     }
}
```

Now you know the **value** of the loot you can carry away in your knapsack, but you still need to work out the actual composition. One way to do this is by creating an extra array `L[j]`, containing the last choice of article added to get `V[j]`.

November 15

Wednesday we looked at how to find the highest value that can be packed into a knapsack with capacity C, n items with weights $w_1, \ldots, w_n$, and values $v_1, \ldots, v_n$. That algorithm told us how find the highest **value** we can pack into the knapsack, but now which items to pack. Add an array $L[j]$ (the index of the last item packed into an optimal packing not exceeding capacity j), and we can track the choices of what fills up an ideal knapsack:

```
V[0] = 0;
for (int j = 1; j <= C; j++) {
    V[j] = V[j-1];
    L[j] = L[j-1];
    for (int i = 0; i < n; i++) {
        if (w[i] <= j && v[i] + V[j-w[i]] > V[j]) {
            V[j] = v[i] + V[j-w[i]];
            L[j] = i;
        }
     }
}
```

Now we can work backwards from $L[C]$. We know we added item with index $L[C]$, and that item had weight $w[L[C]]$, so the previous item must have been $L[C - w[L[C]]]$, and so on.

Matrix chains

Suppose you (or your computer) have to multiply a chain of matrices:

$$A \times B \times C \times D \times E.$$

Matrix multiplication of a pair of matrices is well-defined (if it's defined at all), and for more than two, multiplication is associative:

$$(((AB)(CD))E) = (A((BC)(DE))).$$

You get the same answer, either way, but the amount of work (measured in the number of multiplications) may well differ. Suppose A is a 10×100 matrix, B is a 100×5 matrix, and C is a 5×50 matrix. If we group the multiplication as $((AB)C)$, then the amount of work is proportional to $10 \times 100 \times 5 + 10 \times 5 \times 50$, or 7500 multiplications. On the other hand, if we grouped the product as $(A(BC))$ we'd be faced with $100 \times 5 \times 50 + 10 \times 100 \times 50$, or 75000 multiplications — 10 times as many! So, order counts.

How much work is it to check out all the ways of parenthesizing a product? Try it out with just a few matrices (no more than 10), and you'll find the number of groupings grows rapidly — it's proportional to 4^n for n matrices. The exact number of groupings is a challenging thing to calculate, it's called the Catalan number.

It's not practical to check something proportional to 4^n before we even get down to the business of calculating the matrix product. Let's look at the structure of the problem. Suppose we (magically) have an optimal grouping for multiplying n matrices. Examine the last matrix multiplication performed:

$$((A_1 \times \cdots A_k)(A_{k+1} \times \cdots A_n)).$$

If the entire product is optimal (uses the fewest possible multiplications), then the subproducts $(A_1 \cdots A_k)$ and $(A_{k+1} \cdots A_n)$ must also be optimal. Otherwise, if there were a better way to calculate either subproduct, it would lead to a more efficient grouping to calculate the entire product. So, our Dynamic Programming solution will keep track of the best ways of calculating subproducts, and combine these into the best ways of calculating the entire product. We can define the following array, for $1 \leq i \leq j \leq n$:

$$C[i][j] = \text{minimum cost of multiplying } A_i \cdots A_j.$$

Clearly $C[i][i]$ is zero, for all i, since there is no cost in the "product" of one matrix. If $j > i$, then we get the recurrence relation:

$$C[i][j] = \min_{i \leq k < j} \{C[i][k] + C[k+1][j] + r_i r_{k+1} c_j\}.$$

... where $r_i r_{k+1} c_j$ is the product of the number of rows in matrix i, the number of rows in matrix $k+1$, and the number of columns in matrix j.

November 20

Let's write the algorithm using the convention that we begin counting from 0, so we consider matrices A_0 through A_{n-1}, A_0 has dimensions $s_0 \times s_1$, A_1 has dimensions $s_1 \times s_2$, etc. The most natural approach would be a recursive function (supposing we have an array with $s_0, \ldots s_n$):

```
int rec_matrix_product(int i, int j, int *s) {
    if (i == j) {
       return 0;
    } else {
       int C = rec_matrix_product(i, i, s) +
               rec_matrix_product(i+1, j, s) +
               s[i]*s[i+1]*s[j+1];
       for (int k = i+1; k < j; k++) {
          int c = rec_matrix_product(i, k, s) +
                  rec_matrix_product(k+1, j, s) +
```

```
                s[i]*s[k+1]*s[j+1];
            if (c < C)   C = c;
        }
        return C;
    }
}
```

If we solve the recurrence for the running time of the above code, we find that $T(n)$ is still exponential, so we haven't made any progress from 4^n! We need to solve this problem "bottom-up." If we start along the diagonal where $i == j$, the problem is easy, since $C[i][i]$ is zero. Then we work on the diagonal where $j == i + 1$, and so on:

```
for (int i = 0; i < n; i++)
    C[i][i] = 0;
for (int len = 1; len < n; len++) {
    for (int i = 0; i < n - len; i++) {
        int j = i + len;
        C[i][j] = C[i][i] + C[i+1][j] + s[i]*s[i+1]*s[j+1];
        for (int k = i+1; k < j; k++) {
            if (C[i][k] + C[k+1][j] + s[i]*s[k+1]*s[j+1] < C[i][j])
                C[i][j] = C[i][k] + C[k+1][j] + s[i]*s[k+1]*s[j+1];
        }
    }
}
```

Now we know the **cost** of a best grouping, but we need to recover grouping itself. You could work backwards from the optimal cost, and find two subcosts that would yield that number, but it's probably more straightforward just to keep track of the best groupings at each step:

```
for (int i = 0; i < n; i++) {
    C[i][i] = 0;
    B[i][i] = i;
}
for (int len = 1; len < n; len++) {
    for (int i = 0; i < n-len; i++) {
        int j = i + len;
        C[i][j] = C[i][i] + C[i+1][j] + s[i]*s[i+1]*s[j+1];
        B[i][j] = i;
        for (int k = i+1; k < j; k++) {
            if (C[i][k] + C[k+1][j] + s[i]*s[k+1]*s[j+1] < C[i][j]) {
                C[i][j] = C[i][k] + C[k+1][j] + s[i]*s[k+1]*s[j+1];
                B[i][j] = k;
            }
        }
    }
}
```

Optimal binary search trees

Suppose you have data organized into a BST by keys. It seems wise to keep the most frequently accessed keys near the root, and the rarely accessed keys further down. In fact, you can quantify this wisdom by decreeing that it takes one step to get to the root, two steps to get to its children, etc., and (if you know the frequency, say f_i of key K_i) you can calculate the cost of accessing K_i as $f_i \times K_i$. If you sum this up over all $f_i \times K_i$, you get the **weighted internal path length** of your BST. Inspection shows that this may be different for two BSTs with identical keys, so structure matters.

Suppose you're given an ordered list of keys, $K_1 < K_2, \ldots, K_n$, and each key has an associated frequency $f_1, \ldots, f_n$. We want to build a BST that minimizes the work we expect to do accessing each key. Suppose we know that a BST with a minimal weighted internal path has root K_j, then it follows that its left subtree (with keys $K_1, \ldots, K_{j-1}$) and its right subtree (with keys $K_{j+1}, \ldots, K_n$) must also have minimal weighted internal path length. This leads to the recurrence, where $C[i][j]$ represents the lowest possible cost of a BST with keys $K_i, \ldots, K_k$, and $w(i,j,k)$ is the cost added by making the node with key K_j the root for subtrees $K_i, \ldots, K_{j-1}$ and $K_{j+1}, \ldots, K_k$.

$$C[i][k] = \begin{cases} 0 & k < i \\ f_i & k == i \\ \min i \le j \le k \{C[i][j-1] + C[j+1][k] + w(i,j,k)\} & \text{otherwise} \end{cases}$$

Notice that when we make the node with key K_j the root of subtrees with nodes $K_i, \ldots, K_{j-1}$ and $K_{j+1}, \ldots, K_k$, we increase all the paths to nodes in the subtrees by 1, so we add $f_i + \cdots + f_{j-1}$ and $f_{j+1} + \cdots + f_k$ to our cost, plus the cost (f_j) of getting to the new root note. That gives us a formula for $w(i,j,k)$ (which, curiously, doesn't depend on j).

You should be able to modify the code for matrix multiplication to find an optimal BST, using a triply-nested loop (exercise). Some care is needed in manipulating the indices. This problem is magnified in Assignment 4, where we re-tool the indices in the pursuit of generic code.

November 22

Current genetics and biochemistry have given prominence to the problem of quickly identifying patterns in sequences of amino acids and other biological objects. Clearly you'd want algorithms for rapidly matching strings and substrings, but other subtler patterns are important. Consider the problem of sequences, subsequences, and supersequences (this example if from *Algorithms* by Cormen, Leiserson, and Rivest).

We can consider strings to be sequences, for example "ACCTGAA" can be thought of as a sequence of characters with the first character 'A', the second character 'C', the fifth character 'G', and so on (I'm using a 1-origin instead of 0-origin for a reason that will become clear in a moment).

Suppose string `X = "ACCTGAA"` and string `Y = "ATA"`. Then we say that `Y` is a subsequence of `X` (or `X` is a supersequence of `Y`), since we can produce `Y` by deleting characters 2,3, 5, and 7 from `X` (there's another deletion that also works). Clearly, by this rule, the empty sequence `""` is a subsequence of any other sequence.

Given two sequences, $X = \langle x_1, x_2, \ldots, x_m \rangle$, and $Y = \langle y_1, y_2, \ldots, y_n \rangle$, what is a longest common subsequence of X and Y? Brute force would mean we would list every subsequence of X — each element is either in or out of each subsequence, leading to 2^m possibilities — and then check whether each is a subsequence of Y. Exponential algorithms get old really fast.

However, the problem has an optimal-substructure property. Define X_i, the ith prefix of X, to be the sequence $\langle x_1, \ldots, x_i$ (the special case X_0 is the empty sequence — that's the reason for the 1-origin). Now we can make the following observation about any Longest Common Subsequence (LCS) $Z = \langle z_1, \ldots, z_k \rangle$ of $X = \langle x_1, \ldots, x_m \rangle$ and $Y = \langle y_1, \ldots, y_n \rangle$. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}. If $x_m \neq y_n$ and $z_k \neq x_m$, then Z is an LCS of X_{m-1} and Y_n. Symmetrically, if $x_m \neq y_n$ and $z_k \neq y_n$, then Z is an LCS of X_m and Y_{m-1}.

Define a two-dimensional array $c[i][j]$ == the length of an LCS of X_i and Y_j. You have the following recursive formula:

$$c[i][j] = \begin{cases} 0 & i == 0 \text{ OR } j == 0 \\ c[i-1][j-1] + 1 & i,j > 0 \text{ AND } x_i == y_j \\ \max(c[i][j-1], c[i-1][j] & i,j > 0 \text{ AND } x_i \neq y_j \end{cases}.$$

The recursive definition immediately gives us the row $c[0][j]$, and column $c[i][0]$, for $0 \le i \le m$ and $0 \le j \le n$. We then proceed from top-to-bottom and left-to-right to fill in $c[i][j]$. You can write (exercise) a doubly-nested loop that iterates over i and j and finds, simultaneously, the length of an LCS of X_i and Y_j and the last character of such an LCS. This has complexity $O(mn)$ — much nicer than exponential.

You can also pose (and solve) the problem of a shortest common supersequence of X and Y, in an analogous way.

A weirder (to me) result comes from asking: in how many distinct ways is Y a subsequence of X. In other words, how many distinct deletions of letters from X yield Y. In our previous example, `X = "ACCTGAA"`, and `Y = "ATA"`, we'd find that Y occurs as a subsequence of Y twice. The empty sequence occurs exactly once as a subsequence of any other sequence.

Counting these subsequence is clearly not possible. Suppose X is a sequence of 64 'A's and Y is a sequence of 32 'A's. Then Y occurs as a subsequence of X in as many ways as you can choose 32 'A's to delete: 64-choose-32 — greater than 2^32.

To make this concrete, suppose you apply to legally change your name to a sequence of characters taken from the set $\{A, C, T, G\}$. You quietly obtain DNA samples from friends and acquaintances. Now you count the number of times your new name (let's call it `TACCAG`) occurs in other people's DNA, and you bill them a dollar for each use of your intellectual property. You might be fabulously rich, but you're faced with the prospect of an exponential algorithm to count the number of licenses to charge for.

But, optimal substructure comes to the rescue. If we denote your new name as X, your friend's DNA sequence as Y, and $C[i][j]$ as the number of times X_i occurs as a subsequence of Y_j, we get the following cases

1. If $i == 0$, then $C[i][j]$ is 1, since the empty sequence occurs exactly once as a subsequence of any sequence.
2. If $i > j$, then $C[i][j]$ is 0, since a sequence can't be a subsequence of a shorter sequence.
3. If $j \geq i > 0$ AND $X_i \neq Y_j$, then $C[i][j]$ is the same as $C[i][j-1]$.
4. If $j \geq i > 0$ AND $X_i = Y_j$, then $C[i][j] = C[i][j-1] + C[i-1][j-1]$, since you count the number times X_i occurs as a subsequence of Y_{j-1} (without using y_j), plus the number of ways X_{i-1} occurs as a subsequence of Y_{j-1}, and then match x_i with y_j.

This also leads to a $O(nm)$ algorithm. Start sending those bills today!

November 27

A reasonable question might be whether there is some easier way to calculate the optimal grouping of matrices (optimal BST structure, etc.) than the Dynamic Programming solution I presented.

In particular, a class of algorithms called greedy algorithms attempt to build up a solution to an optimization question one step at a time, never reconsidering a step once it has been taking (perhaps myopic algorithms would be a better name than greedy algorithms).

For example, in the case of grouping matrices for optimal multiplication, you might wonder whether you would get the optimal grouping by always choosing the top-level parenthesization so as to minimize (or, alternatively, maximize) the number of multiplications in the last step.

In the first case, consider matrices A_0, with dimensions 2×3, A_1, with dimensions 3×4, and A_2, with dimensions 4×3. If we group them so that the last matrix multiplication is minimal, we get:

$$(A_0(A_1A_2))$$

...for a total of 54 multiplications. But the grouping

$$((A_0A_1)A_2)$$

...yields 48 multiplications. So that greedy algorithms doesn't work.

The opposite approach (maximize the number of multiplications at the top level) runs into trouble with A_0 and A_1 having the same dimensions as previously, but A_2 having dimensions 4×2. The greedy criterion leads to

$$((A_0A_1)A_2)$$

...with 40 multiplications, versus 36 multiplications if we group the matrices:

$$(A_0(A_1A_2)).$$

So greediness doesn't work so far for matrix grouping. How about BST structure? A likely choice is to place the most frequent keys as high as possible, but that doesn't seem to work for key 'A' has frequency 3, key 'B' has frequency 2, and key 'C' has frequency 2.

Of course, these counterexamples don't prove that there doesn't exist any greedy algorithm for solving these problems, I've just dealt with the first ones that come to mind.

C strings

I present this material out-of-sequence since we needed some of the earlier topics in order to make assignments do-able. However, a few comments on how to treat strings in C is needed.

When you're finished, you should be able to say what the following code does, and why:

```
char*s="char*s=%c%s%c;main(){printf(s,34,s,34);}";main(){printf(s,34,s,34);}
```

Can you do something similar without the magic number 34?

C doesn't truly have strings, it has arrays of `char` terminated by the special zero character `'\0'`. Once you've left the scope where an array of `char` was declared, C can't distinguish between `char s[5]` and `char *s`.

Here are a few C string functions you should become familiar with (see K.N. King for details). You need `stdio.h'` for the first two, and `string.h` for the others.

printf You've already used this. The first argument is a (possibly) formatted string. The remaining arguments are addresses of objects, that are inserted into the formatted string wherever a conversion specifier (%-something) is found.

scanf Use this to read space-delimited strings from standard in. It will read to the next whitespace or the string delimited `'\0'`, so be sure you have enough room to hold expected input:

```
#include <stdio.h>

int main()
{
    char s1[80+1], *s2;
    scanf("%80s", s1); /* okay */
    scanf("%80s", s2); /* not okay */
    printf("Strings %s, %s\n", s1, s2);
    return 0;
}
```

strlen(char *s) This returns the number of characters before the first occurrence of `'\0'` at address `s`.

strcpy(char *s1, const char *s2) Copy characters from `s2` to `s1`. It copies up to the first `'\0'` in `s2`, and has no way of checking whether there is enough room in `s1` to accommodate these. User beware. Behaviour is undefined if `s1` and `s2` overlap.

strcmp(char *s1, char *s2) The comparison `s1 == s2` doesn't work (you're comparing pointers). What you really want to know is whether the characters preceding the first `'\0'` in `s1` are the same as the corresponding characters in `s2`. This function tells you more: is `s1` greater than (positive value returned), equal to (0 returned), or less than (negative value returned) `s2` in alphabetical order.

November 29

Until this point you've used `scanf` and `printf` from `<stdio.h>`. This works well when you have one standard input and one standard output, and can be adapted to reading and/or writing from files where needed, using a unix shell's IO redirection.

```
$ prog <infile /* redirect stdin to infile */
$ prog >outfile /* redirect stdout to outfile */
$ prog <infile >outfile /* do both */
```

Inevitably you'll want more. You may want to read or write to more than one file, or perhaps put error messages somewhere different than other output.

We're assuming that the files you're interested in are streams of bytes to be interpreted as characters, broken into lines by an OS-specific end-of-line character(s) (in CSC209 you'll discuss binary files). You can declare a file pointer, and then open a file attached to that pointer for reading:

```
FILE *fp;

if ((fp = fopen("/path/to/my/file", "r")) == NULL) {
     /* file can't be opened --- do something */;
};
```

The `r` indicates that data will be read from this file, but not written to it. The value of `fp` should be tested, since it will be `NULL` if the file could not be opened. You may also open a file for writing with the argument `w` (clobbers previous instances of this file, if it exists, creates it otherwise), or appending with `a` (appends to the end of a file, if it exists, creates it otherwise). There options to combine reading with writing or appending, but that's too exciting for today's lecture.

You might also want to change which file an already-open stream is attached to. The file-pointers `stdin`, `stdout`, and `stderr` are open by default, and your program can redirect them to named files:

```
if (freopen("/path/to/myerror", "w", stderr) == NULL) {
    /* error, handle it */;
}
```

If this was successful, any output to `stderr` is sent to `myerror`.

Before doing anything useful with a file pointer, you should know how to close a file that's attached to a file pointer. This operation ensures that any data buffered for the file is flushed to it, and satisfies any OS constraints on how many files may be open simultaneously:

```
fclose (fp);
```

With file pointers, you can write output to specific streams. `fprintf` behaves like `printf`, except you specify which file is being written to (`printf` writes to `stdout` by default):

```
fprintf(fp, "The number is: %d\n", num);
fprintf(stderr, "The number is not: %d\n", num-1);
fprintf(stdout, "The number could be: %d\n", num * num);
```

The first statement writes to whatever stream is attached to `fp`, the second to standard error. The third is identical to `printf` (without the file-pointer argument).

Symmetrically, there is `fscanf` to read space-delimited strings from a file:

```
while(fscanf(fp, "%d", &i) != EOF) {
    /* do something with i */;
}
```

Since `scanf` returns the number of of items matched and assigned to variables, the loop above could have replaced `!= EOF` with `== 1`.

If you want to read in lines, use `fgets` **NOT** `gets`:

```
char *buf[80+1];
while (fgets(buf, sizeof(buf), fp) != EOF) {
    /* process buf */;
}
```

This stores either the first `sizeof(buf) - 1` characters, or up to (and including) the first newline from the stream attached to `fp` in `buf`.

You may need to get your input character-by-character (if you can't get what you want from `scanf` or `fscanf`):

```
char ch;
while ((ch = getc(fp)) != EOF) {
    /* do something with ch */;
}
```

There is (nearly identically) `fgetc`, which has the advantage/disadvantage of being defined as a function (`getc` is usually a macro).

Here's a fragment of code to merge two files containing sorted lists of integers (represented as strings) (this may well be the starting point for either next Monday's tutorial or next Wednesday's tutorial):

```
    FILE *fp0, *fp1;
    int x, y;
    int eof0, eof1;

    /* open fp0 and fp1 for reading ... */

    eof0 = fscanf(fp0, "%d", &x);
    eof1 = fscanf(fp1, "%d", &y);
    while (eof0 != EOF && eof1 != EOF) {
        if (x < y) {
            printf("%d\n", x);
            eof0 = fscanf(fp0, "%d", &x);
        } else {
            printf("%d\n", y);
            eof1 = fscanf(fp1, "%d", &y);
        }
    }
    while (eof0 != EOF) {
        printf("%d\n", x);
        eof0 = fscanf(fp0, "%d", &x);
    }
    while (eof1 != EOF) {
        printf("%d\n", y);
        eof1 = fscanf(fp1, "%d", &y);
    }
```

December lecture

December 6

Here are some things I've been asked to review.

- How do we create dynamic arrays of dimension two or higher in C/C++?

  There are two very distinct approaches. In the FAQ for A2 you were shown how to manipulate a one-dimensional array of size `n*n` as a two-dimensional array. This approach has the advantage that it

uses fewer calls to `malloc` (or `new`), but it leaves the programmer with the responsibility for working out the array indexing arithmetic — `A[i][j]` is represented as `A[i*n +j]`. This becomes even more detailed as you extend to higher dimensions (you might want to experiment with how you'd created a three-dimensional dynamic array).

A different approach is to make your two-dimensional array an array of pointers. This involves more calls to `malloc` or `new`, but it may seem more natural to the programmer, and extends simply to higher dimensions. It also allows you to have rows of different lengths, which might be suitable for, say, Pascal's Triangle or an array of strings (i.e., a two-dimensional array of `char`). Here's some example code:

```
int **buildTable(int n)
{
    int **table = new int*[n];
    for (int i = 0; i < n; i++) {
        table[i] = new int[n];
        for (int j = 0; j < n; j++)
            table[i][j] = 7;
    }
    return table;
}

int **Pascal(int n)
{
    int **table = new int*[n];
    for (int i = 0; i < n; i++) {
        table[i] = new int[i+1]; /* row i has i+1 elements */
        table[i][0] = 1;
        table[i][i] = 1;
        for (int j = 1; j < i; j++)
            table[i][j] = table[i-1][j-1] + table[i-1][j];
    }
    return table;
}

int *buildTable2(int n)
{
    int *table = new int[n*n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            table[i*n + j] = 7;
    }
    return table;
}

int main()
{

    int **p = Pascal(5);
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < i+1; j++)
            printf("%d\t", p[i][j]);
        printf("\n");
    }
```

```
    return 0;
}
```

- If class `B` is derived from class `A` and has a member function with the same name, which function gets called, and when.

  It depends. A function of the same name in the derived class hides the function in the base class, even if the signatures (e.g. parameter list) differ. You can refer explicitly to the base classes version by using `::` (the namespace operator). A reference to the base class will look for a match only there, unless you modify the function with `virtual`. Here is some code that illustrates some of this behaviour (you should experiment with examples of your own):

```
#include <stdio.h>

class A {
public:
    void func1(int i);
    virtual void func2(int i);
};

void A::func1(int i)
{
    printf("func1 in A\n");
}

void A::func2(int i)
{
    printf("func2 in A\n");
}

class B : public A {
public:
    void func1(int i, int j);
    void func2(int i);
};

void B::func1(int i, int j)
{
    printf("func1 in B\n");
}

void B::func2(int i)
{
    printf("func2 in B\n");
}

int main()
{
    A a, *ap;
    B b, *bp;

    a.func1(1);
    b.func1(1,2);
    /* b.func(1) won't work--- no matching signature */
```

```
    b.A::func1(1);
    a = b;
    a.func1(1);
    /* a.func1(1,2); can't find this signature in A */
    ap = &b;
    bp = &b;
    ap->func1(1);
    /* ap->func1(1,2); can't find this signature in A */
    ap->func2(1);
    a.func2(1); /* calls base class version */
    /* bp->func1(1); can't find this signature in B */
    bp->func1(1,2);
    return 0;
}
```