Computer Science 102
Lab 13

Efficient methods of searching -

Suppose an online telephone book is comprised of structures of the following form:

```
struct entry_type
{
   char name[60];   // e.g. Westall,  James M.
   char num[10];    // e.g. 8645551212
};
```

and it consists of 1048576 = 1024 x 1024 = 2^ 20 entries as shown

```
struct entry_type phone_book[1024 * 1024];
```

Now suppose your mission is to write the function

```
struct entry_type *lookup(char *target_name);
```

The naive approach to doing this is:

```
struct entry_type *lookup(char *target_name);
{
   int i;
   for (i = 0; i < (1024 * 1024); i++)
   {
      if (strcmp(target_name, phone_book[i]) == 0)
          return(&phone_book[i]);
   }
}
```

For this approach, the length of the search (number of array elements that must be compared against target_name) is:

- best case: 1 (You were asked to lookup the first element)
- worst case: 1024 * 1024 (You were asked to lookup the last element)
- average case: 1024 * 1024 / 2

If the phonebook is in random order, this is the best possible result.

However if the phonebook is sorted by name (as a real phonebook is) a more clever algorithm can produce the following results.

- best case: 1 you found the element on the first look
- worst case: 20 (instead of 1024 * 1024)
- average case: left as an exercise (but obviously between 1 and 20)

This better algorithm is known as binary search and belongs to the class of algorithms known as *divide and conquer.* It begins by comparing the "middle element", phone_book[1024 * 1024 / 2].name with target_name.

If target_name > phone_book[1024 * 1024 / 2].name then the whole first half of the phone book is known NOT to contain the target name. The next step is to compare target_name with phone_book[3 * 1024 * 1024 / 4].name. At each step, the remaining range is divided by 2 and the next probe point is set to the middle of the remaining range.

Because the range of elements to be searched is divided by 2 at each step, the number of elements that must be tested is bounded by log2(the table size). *Therefore an element may be looked up in a sorted table of 4 BILLION elements in only 32 comparisons.*

The bisection algorithm

Given a function f(x), a common problem in numerical computing is to find a *value x for which f(x) = 0*. If

- f(x) = 2 * x + 7 or
- f(x) = x^2 -2x + 1,

you can readily solve for x using algebra.

But if the function is something like

- f(x) = x * e^x - 10 * x,

there will be no algebraic solution and if the function is something like

- f(x) = x^2 + 1

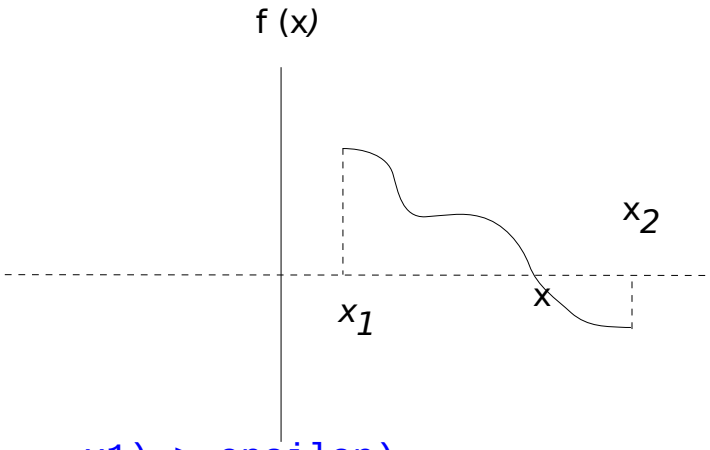
there may be NO real number solution at all!

The *bisection* algorithm

If the function f(x) is continuous and we can find points x1 and x2 so that f(x1) * f(x2) < 0 (the function values have different signs at x1 and x2), then we are guaranteed by the intermediate value theorem that there there exists an x with

- x1 < x < x2 for which f(x) = 0

The bisection algorithm is a divide and conquer algorithm that can be used to find the the x value for which f(x) ~ 0.




```
while ((x2    x1) > epsilon)
{
    compute xm = the midpoint between x2 and x1
    if (f(x1) * f(xm)) < 0)
                replace x2 with xm
            else
                replace x1 with xm
}
```

In this lab you will you will implement the bisect function that can find numerically a value for which an arbitrary continuous function takes on a value of approximately 0.

```
/**/
/* This function attempts to find a zero of a function */
/* using the bisection method                          */

int bisect(
double (*f)(double),    // external function whose zero we seek
double *minval,         // pointer to minimum of search range
double *maxval,         // pointer to maximum of search range
double epsilon);        // tolerance
```

Your function should begin by ensuring that *maxval > *minval and that f(*minval) and f(*maxval) have different signs. If not *bisect* should return 0.

Otherwise your function should perform bisection until

```
     (*maxval - *minval) < epsilon
```

In this case the *value returned should be the number of times the interval is bisected*. Your bisection function should directly modify the values *minval and *maxval so that on return the caller can use *minval as the root of the equation f(x) = 0.

```
int main()
{
   int code;
   double min;
   double max;

   min = 0.5;
   max = 2.0;

   code = bisect(f1, &min, &max, 1.0e-10);
   printf("code = %5d min = %16.6le f(min) = %16.6le \n",
             code, min, f1(min));
```

In this lab you will submit a single file, bisect.c that includes *ONLY* the new bisect() function constructed as part of this lab. It must not contain the main() function nor the f1() function!

```
sendlab.102.labsection#  lab#  bisect.c
```