Lecture 13:
Writing Efficient Programs

How can you speed up the following program ?

```
#include <stdio.h>
void main()
{
 int n, i;
 scanf("%d", &n);
    /*n must be >2 */

 for(i=2; i< n/2 +1; i++)
   if ((n%i)== 0) break;

 if ((n%i)==0)
     printf("0\n");
 else
     printf("1\n");
}
```


A more efficient version

Observation: If $n = a \cdot b$ then one of a and b is $< n + 1$

```
#include <stdio.h>
#include <math.h>
void main()
{
 int n, i, sqrtn;
 scanf("%d", &n);
 sqrtn = sqrt(n);
 for(i=2; i< sqrtn +1; i++)
   if ((n%i)== 0) break;

 if ((n%i)==0)
     printf("0\n");
 else
     printf("1\n");
}
```

Estimating the running times of programs

Rules of thumb:

- each basic operation (+, -, *, /,...) takes the same amount of time.
- running time of a for/while loop
  = number of iterations *x* number of basic operations per iteration

Running time is generally measured as a proportion of the "size" of the input : $O(n)$, $O(n^2)$...

- **Remember:** Careful design may greatly boost performance
- **Warning:** Don't sacrifice readability and clarity for tiny performance gain

Efficiency of Sorting Programs

Bubble Sort:

do the following for i=0 to n-1:
"bubble" the smallest value among A[i], A[i+1], ... A[n-1] into A[i].

```
void swap(int *x, int *y)
{int t;
 t = *x; *x =*y; *y=t;
}
void bubblesort(int A[], int n)
{ int i, j;
  for(i=0; i<n; i++)
   for(j=i;j <n-1; j++)
     if (A[j] > A[j+1])
        swap(&A[j], &A[j+1]);
}
```

Running Time?

Quicksort (divide and conquer strategy):

- pick a "random" value *val* among A[0], A[1], .. A[n-1]
- partition A according to *val*
  (put all numbers that're <= *val*
  before all numbers that're > *val*)
- recursively sort the numbers that're <= *val*
  and the numbers that're > *val*

```
void quicksort(int A[], int lo, int hi)
{ int mid, val;
  if (hi-lo <=0)
    return;
  if (hi -lo ==1)
    { if (A[hi]< A[lo])
        swap(&A[hi], &A[lo]);
      return;}

  val= A[lo + rand()%(hi -lo)];
  mid = partition(A, val, lo, hi);
  quicksort(A, lo, mid);
  quicksort(A, mid+1, hi);
}
void main()
{int a[N], i;
 ...
 quicksort(a, 0, N-1);
 ...
}
```


Partition( )

```
/*   assumes hi> lo +1; returns mid such  */
/*    that D[i]<=val for all i<=mid       */

int partition(int D[], int val, int lo, int hi)
{
 int i, c1=0, c2, *B;
 B = calloc(hi-lo +1, sizeof(int));

for(i=lo; i<=hi; i++)
   if(D[i]<=val)
    {B[c1]=D[i]; c1++;}

c2=c1;
for(i=lo; i<=hi; i++)
   if(D[i] >val)
    {B[c2]=D[i]; c2++;}

for(i=lo;i<=hi; i++)
    D[i]=B[i-lo];

free(B);
return lo+ c1-1 ;
}
```

Analysis of quicksort's running time

Time to partition n elements = $O(n)$.

Assume: Since *val* was randomly chosen, approximately 1/2 the array elements are <= *val* .

So,

time to "quicksort" n numbers

= $O(n) + 2 \times$ (*time to "quicksort" n/2 numbers*)

=

=

= $O(n \log n)$

A Data Management Problem

Goal: Design a program that keeps track of a list of numbers for the user.

Desired Functionality: Given any number, program should be able to

- Check if it is present in the current list
- Insert it in the list if it is not already in there
- Delete it from the list if it is in there

Applications: Databases, On-line phonebooks, ...

Array-based solution: Store in sorted order

Trivial search algorithm: "travel" down the array

```
int search(int a[], int value, n)
{ int i=0;
  while((i<n)&&(a[i]< value))
        i=i+1;
  if a[i] == value
        return i;
  else return -1;
}
```

More efficient: binary search

```
int binsearch(int a[], int value, n)
{int low=0; int high = n-1; int mid;
 while(high >=low)
  { mid = (low + high)/2;
   if (value == a[mid]) return mid;
   if (value < a[mid])
       high =mid-1;
   else
       low = mid +1;
   }
 return -1;
 }
```


running time of binsearch()

Each step divides the range of the search into 1/2.
After $log_2 n$ steps the range becomes 1.

($log_2 n$ = number of digits in binary representation of n)

For $n = 1000,000$
time taken by one call to search()
= time taken by $50,000$ calls to binsearch()

Question: What about recursive binary search?

```
int rbinsearch(int a[],int val,int low, int high)
{ int mid = (low + up)/2;
  if (a[mid] ==val) return mid;
  if (low==high) return -1;
  if (a[mid] < val)
     return rbinsearch(a, val, mid+1, high);
  else
     return rbinsearch(a, val, low,mid);
}
```

insert

```
/* returns index where val is inserted */
int insert(int a[], int val, int n)
{
 int i;
 if (binsearch(a, val,n)> 0)
    return -1; /*val is already present*/
 else
    for(i=n-1; a[i]>val; i--)
       a[i+1] = a[i];

     a[i+1]=val;
     return i+1;
    }
}
```

Moves around lots of data each time we insert/delete!

Next time: Better solution using "trees"