

Computer Programming: Skills & Concepts (CP1)

Searching and sorting

1st November 2010

Correctness of a Program

- ▶ How can you show that a program is correct?
- ▶ Similar to a mathematical proof: show that certain statements are true at all times in the program (“Invariants”)
- ▶ Brain teaser: Is it possible to write a program that checks any other program, if it is correct?

Power of a number

```
int Power(int n, int k)
/* Assumes k >= 0. Returns n^k: n raised to the power k. */
{
    int p = 1, i = k;
    /* Precondition: i >= 0 */
    while (i > 0) {
        /* Invariant: i >= 0 AND p * n^i == n^k */
        p *= n;
        --i;
    }
    /* p = n^k */
    return p;
}
```

Example: $n = 3$, $k = 4$. The answer should be $3^4 = 81$.

The computation progresses as follows. Initially, $i = k$ and $p = 1$. Note that $p \times n^i$ is invariant!

	i	p	$p \times n^i$
Initial	4	1	$1 \times 3^4 = 81$
Iteration 1	3	3	$3 \times 3^3 = 81$
Iteration 2	2	9	$9 \times 3^2 = 81$
Iteration 3	1	27	$27 \times 3^1 = 81$
Iteration 4	0	81	$81 \times 3^0 = 81$

Searching an array

```
typedef enum {FALSE, TRUE} Bool_t;

Bool_t LinearSearch(int n, int a[], int sKey)
/* Returns TRUE iff (if and only if) sKey is contained
 * in the array, i.e., there exists an index i with 0 <= i < n
 * such that a[i] == sKey.
 */
{
    int i;
    for (i = 0; i < n; ++i) {
        if (a[i] == sKey) return TRUE;
    }
    return FALSE;
}
```

variant:

- ▶ Could use return type int with #DEFINE for TRUE, FALSE (see BinarySearch)

Binary search

Sometimes we quickly want to find an entry in an array.

It helps if the array is sorted.

How do you search for a name in a telephone book?

Binary search

```
int BinarySearch(int n, int a[], int sKey)
/* Assumes the elements of the array a are in ascending order.
 * Returns TRUE iff sKey is contained in the array, i.e.,
 * there exists an index i with  $0 \leq i < n$  and  $a[i] == sKey$ .
 */
{
    int i, j, m;

    i = 0;
    j = n - 1;
    /* Precondition:  $a[0] \leq a[1] \leq \dots \leq a[n-1]$  */
```

```
while (i < j) {
    /* Invariant:  i <= j  AND
    * if sKey is in a[0:n-1] then sKey is in a[i:j] */
    m = (i + j)/2;
    if (sKey <= a[m])
        j = m;
    else
        i = m + 1;
}
/* EITHER a[i] == sKey OR sKey is not in a[0:n-1] */
return a[i] == sKey;
}
```

- ▶ Note how we return true/false ...

Running time

The (*worst-case*) *running time* of a function (or *algorithm*) is defined to be the maximum number of steps that might be performed by the program as a function of the *input size*.

- ▶ For functions which take an array (of some basic type) as the input, the length of the array (n in lots of our examples) is usually taken to represent size.
- ▶ The running time of Linear Search is $\Theta(n)$ (ie, around $c \cdot n$ for some constant c), and the running time of Binary search is $\Theta(\lg(n))$ (proportional to $\lg(n)$).
- ▶ This *size* is conceptual and *not* what gets measured by the `sizeof` command in C (`sizeof` applied to an array is not length)

Measuring running time on a machine

```
#include <time.h>
Bool_t flag = FALSE;
int a[880000];
double start, stop, t;
...
start = clock();
flag = LinearSearch(a, 880000, -5);
stop = clock();
t = (stop-start)/CLOCKS_PER_SEC;
printf("Time spent by Linear Search was %lf seconds.\n", t);
...
```

Machines getting faster ... on DICE, this method measures *both* linsearch (and binsearch) at 0.000000 secs on arrays up to length 400000!

Sorting

Given an array of integers (or any *comparable* type), re-arrange the array so that the items appear in increasing order.

Bubble sort

“Proto loop”

```
for (i = n - 1; i >= 1; --i) {  
    /* Rearrange the contents of array elements a[0], ..., a[i],  
     * so that the largest value appears in element a[i].  
     */  
}
```

“Method”:

- ▶ Find the largest item, and move it to the end;
- ▶ **repeat** for 2nd largest item, and so on ...

Bubble sort (cont'd)

The task of rearranging the contents of array elements $a[0], a[1], \dots, a[i]$ so that the largest value appears in element $a[i]$, may be handled by the following simple loop:

```
for (j = 0; j < i; ++j) {  
    if (a[j] > a[j+1]) swap(&a[j], &a[j+1]);  
}
```

(The largest value supposedly “bubbles” up the array into its appropriate position.)

Bubble sort code

```
/* Sorts a[0], a[1], ..., a[n-1] into ascending order. */
void BubbleSort(int a[], int n)
{
    int i, j;
    for (i = n - 1; i >= 1; --i) {
        /* Invariant: The values in locations to the right of a[i]
         * are in their correct resting places: that is, they are
         * the n - i - 1 largest elements, and they are correctly
         * ordered among themselves.
         */
        for (j = 0; j < i; ++j) {
            if (a[j] > a[j+1]) swap(&a[j], &a[j+1]);
        }
    }
}
```

The function used above is the following function from lecture 11.

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

Running time of Bubble Sort

The (worst case) running time of Bubble Sort is proportional to n^2 . why?

There are better sorting algorithms . . . for example *MergeSort* or *HeapSort* run in time proportional to $n \lg(n)$.

($\lg(n)$ denotes “log to the base-2”)

A more dramatic example

Sometimes the gap between a good and bad algorithm can be dramatic. Consider the problem of testing whether an n -bit number is prime.

- ▶ The obvious brute force method requires $2^{n/2}$ integer divisions
 - ▶ *why?*
 - ▶ This is completely infeasible if $n = 200$ (say).
- ▶ On the other hand, a (non-obvious) algorithm for primality testing which take time *polynomial* in n was discovered in 2002 (Agrawal-Kayal-Saxena)

(Needed for RSA public-key cryptosystem.)