Computer Programming: Skills & Concepts (CP)
Searching and sorting

Ajitha Rajan

Monday 13 November 2017

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?
Computers aren't so clever, so we do a simplified version:
Repeatedly chop the array in half to close in on where the element must be. E.g., to search for 17 in:
2 3 5 7 11 13 17 19 23 29
Find the mid-point: $17 > 11$, so narrow to right half:
Find the mid-point: $17 \leq 19$, so narrow to left half:
Find the mid-point: $17 \leq 17$, so narrow to left half:
(yes, we could stop here because we've found it. . .)
Find the mid-point: $17 > 13$, so narrow to right half:
Now we're left with an array of size 1, so either its element is 17 and we've found it, or 17 isn't there.

Binary search

```
int BinarySearch(int n, int a[], int sKey)
/*  Assumes: elements of array a are in ascending order.
 *  Returns TRUE iff sKey is contained in the array, i.e.,
 *  there exists an index i with 0 <= i < n and a[i] == sKey.
 */
{
  /* Precondition:  (n > 0)
                    AND a[0] <= a[1] <= ... <= a[n-1] */
  int i, j, m;
  /*  i will be the start of the sub-array
   *    we're currently chopping;
   *  j will be the end of it (its last element);
   *  m will be the mid-point of it.
   */

  i = 0;
  j = n - 1;
```

```c
  /* Invariant just before (re-)entering loop:  i <= j  AND
   * if sKey is in a[0:n-1] then sKey is in a[i:j] */
  while (i < j) {
    m = (i + j)/2;
    if (sKey <= a[m]) {
      j = m;
    }
    else {
      i = m + 1;
    }
  }
  /* After exiting loop:
   *   (i >= j), by i, j updates, means (i == j).
   *  now 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 proportional to n (i.e., around $c \cdot n$ for some constant c), and the running time of Binary Search is proportional to $\lg(n)$.

Measuring running time on a machine

```c
#include <time.h>
Bool_t flag = FALSE;
int a[24000000];
clock_t start, stop;
double t;
...
start = clock();
flag = LinearSearch(a, 24000000, -5);
stop = clock();
t = ((double)(stop-start))/CLOCKS_PER_SEC;
printf("Time spent by Linear Search was %lf seconds.\n", t);
...
```

On my laptop:

```
Time spent by LinearSearch was 0.069064 seconds.
Time spent by BinarySearch was 0.000001 seconds.
```


Sorting

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

Bubble sort

'Pseudo-code'

```
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]`, ..., `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:  they are
       * the (n - i - 1)-largest elements arranged in
       * positions (i+1), ..., (n-1), in non-descending order
      for (j = 0; j < i; j++) {
         if (a[j] > a[j+1]) {
            swap(&a[j], &a[j+1]);
         }
      }
   }
}
```

The `swap` function used above is the (correct) one from lab 5.

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 $n \lg(n)$.

For general purpose sorting, often use *QuickSort*, which runs in time around $n \lg n$ in most cases, though in bad cases (which?) it can take n^2. Standard C systems provide QuickSort as `qsort`. Occasionally you might know that BubbleSort would be quicker in your application, and want to program it. Anything else is probably specialist.

More about Bubble-Sort can be found in Section 6.7 of 'A Book on C'.

Understanding your loops

These slides are logically small and green: for the mathematically and logically inclined only!

- How can you show that a program is correct?
- One way is to show that certain statements are true at all times in the program (*invariants*)
- In particular, to understand a complex `while`/`for`-loop, it's useful to know what remains true every time you go through it.
- For functions (or other blocks of code) we have *preconditions* (things *assumed* be true before) and *postconditions* (things which *will* be true afterwards given the preconditions).

We'll do a simple example now; then look (in your own time) at the comments in the searching and sorting code, and try to understand what they're saying about invariants.

Power of a number

```
int Power(int n, int k)
/*  Pre-condition: k >= 0. */
/*  On-exit: returns n^k (n raised to the power k). */
{
   int p = 1, i = k;
   /*  Invariant before (re-)entering:
    *  i >= 0  AND  p * n^i == n^k  */
   while (i > 0) {
      p *= n;
      --i;
   }
   /*  After exiting loop: i <= 0  AND p = n^k  */
   return p;
}
```

Warning: `n^k` in the comments is maths notation, not C notation. In C, the ^ symbol is the bitwise exclusive-or operator, something entirely different!

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!

```
/* Invariant before (re-)entering:
   i >= 0  AND  p * n^i == n^k  */
while (i > 0) {
   p *= n;
   --i;
}
/* After exiting loop: i <= 0  AND p = n^k  */
return p;
```

	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	$81 \times 3^0 = 81$

Reading material

Sections of 'A Book on C' that are relevant are:

- A good idea to refresh your memory of arrays (early sections of Chapter 6).
- Section 6.7 has a discussion of `BubbleSort`.