CSE 1030

Yves Lespérance

Lecture Notes

Week 11 — More on Algorithm Analysis and Correctness

The Dutch National Flag Problem [Dijkstra]

Given an array of `char` containing the characters `'R'` (red), `'W'` (white), and `'B'` (blue) in any order, e.g.

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
 R  B  W  W  B  B  R  W  W  R  R  W  R  B  W
```

write a method to rearrange the array elements so that they appear as in the Dutch national flag, i.e. all reds to the left, all whites in the middle, and all blue to the right:

```
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14
 R  R  R  R  R  W  W  W  W  W  W  B  B  B  B
```

The method's running time should be at most linear in the size of the array.

Solution to Dutch National Flag Problem

```
public static void dnf(char[] a)
{  int r = 0;
   int w = 0;
   int b = a.length - 1;
   while (w <= b)
   {  if (a[w] == 'W')
         w++;
      else if (a[w] == 'R')
      {  if (r != w)
            swap(a, r, w);
         r++;
         w++;
      }
      else // if a[w] == 'B'
      {  swap(a, w, b);
         b--;
      }
}
public static void swap(char[] a, int i, int j)
{  int temp = a[i];
   a[i] = a[j];
   a[j] = temp;
}
```


Running Time of `dnf`

The method's running time is $O(n)$ where $n =$ `a.length`, because the loop does n iterations and does a constant number of operations in each iteration. To see this, note that `b` $-$ `w` $+ 1$ is equal to n initially, and decreases by 1 at each iteration since either `w` is incremented or `b` is decremented.

Proof of correctness of `dnf`

It is easy to make a mistake in such an algorithm, so we should prove that it is correct. To do this, we must identify a *loop invariant*. As discussed in week 4, this is a condition that is preserved by the loop body, i.e., if it is true at the beginning of a loop iteration and the loop's test condition is true, then the invariant will also be true at the end of the loop iteration.

For our example, a suitable invariant is

$$\begin{array}{l} \mathtt{r} \leq \mathtt{w} \wedge \mathtt{a}[0..\mathtt{r}-1] = \mathrm{'R'} \wedge \\ \mathtt{a}[\mathtt{r}..\mathtt{w}-1] = \mathrm{'W'} \wedge \mathtt{a}[\mathtt{b}+1..\mathtt{a.lenght}-1] = \mathrm{'B'} \end{array}$$

where $\mathtt{a}[\mathtt{i}..\mathtt{j}] = c$ means that

$$\forall \mathtt{k}, \mathtt{i} \leq \mathtt{k} \leq \mathtt{j} \rightarrow \mathtt{a}[\mathtt{k}] = c.$$

To show that the method is correct, we show that the invariant is true at the beginning of the loop (trivial given the way the variables are initialized) and preserved by loop iterations. It follows that if the loop terminates, then the invariant is true at the end and the loop's test condition is false. Together, these conditions imply that the array is properly sorted.

To show that the invariant is preserved by the loop body, there are 3 cases to consider: (1) `a[w] == 'W'`, (2) `a[w] == 'R'`, and (3) `a[w] == 'B'`.

Let us show it for case (3); the other cases are similar. We are given that the invariant and the loop's test condition `w <= b` hold before the loop body is executed. We need to show that the invariant still holds after the loop body.

Since $\mathtt{r} \leq \mathtt{w}$ holds beforehand and `r` and `w` are not changed by the body, it must still hold afterwards.

Since $\mathtt{a}[0..\mathtt{r}-1] = \mathrm{'R'}$ holds beforehand and $\mathtt{r} \leq \mathtt{w} \leq b$, then $\mathtt{a}[0..\mathtt{r}-1]$ is not changed by the swap, and so $\mathtt{a}[0..\mathtt{r}-1] = \mathrm{'R'}$ must hold after the body.

$\mathtt{a}[\mathtt{r}..\mathtt{w}-1] = \mathrm{'W'}$ is also preserved by the same argument.

Finally, since $\mathtt{a}[\mathtt{b}+1..\mathtt{a.lenght}-1] = \mathrm{'B'}$ holds beforehand and $\mathtt{a}[\mathtt{w}] = \mathrm{'B'}$, then $\mathtt{a}[\mathtt{b}..\mathtt{a.lenght}-1] = \mathrm{'B'}$ holds after the swap, and thus $\mathtt{a}[\mathtt{b}+1..\mathtt{a.lenght}-1] = \mathrm{'B'}$ must hold after the decrementation of `b`.

We should also prove that the method terminates, and that can be done by an argument similar the one we gave to show the method runs in $O(n)$ time.

Searching

We have seen that a common operation on arrays and collections is sorting them.

Another common operation is *searching* an array to locate a given value: you are given a value, the target, and you must return the index where it appears in the array; if it doesn't appear, you return some value to indicate that it was not there, such as a value like -1 that is not a valid index.

Like for sorting, there are many algorithms to perform searching. Will look at some and do an analysis.

Linear Search

One algorithm for searching an array is simply to start at the beginning and go through each element in sequence; for each element, you compare it with the value you are looking for, the target; you are done when you find the target or you reach the end of the array. This is called *linear search*.

```
public static int linearSearch(int[] a,
   int target)
{  for (int i = 0; i < a.length; i++)
      if (target == array[i])
         return i;
   return -1;
}
```


If you perform linear search on an array of size n, in the worst case you will have to compare the target with all of the array elements, i.e., do n comparisons; on average, you will compare the target with half of the array elements, i.e., do $n/2$ comparisons.

Thus, we say that in the worst case, linear search takes $O(n)$ operations.

Binary Search

When the array you are searching is already *sorted*, then there is a much more efficient algorithm.

You start by comparing the `target` with the element at the middle of the array. If `target == a[mid]`, you just return `mid` and you are done.

Otherwise, there are two cases:

- `target < a[mid]`, in which case `target` can only be between index `0` and `mid - 1`.
- `target > a[mid]`, in which case `target` can only be between index `mid + 1` and `lenght - 1`.

In either case, you've eliminated half of the array; you can continue by applying the same method to the remaining part of the array.

This method is called *binary search*.

Here is an iterative implementation:

```
public static int binarySearch(int[] a,
   int target)
{
   int from = 0;
   int to = a.length - 1;
   while (from <= to)
   {  int mid = (from + to) / 2;
      if (a[mid] == target)
         return mid;
      else if (target < a[mid])
         to = mid - 1;
      else
         from = mid + 1;
   }
   return -1;
}
```

See textbook for a recursive one.

Analysis

Suppose we have an array of size n. In the worst case, we have that:

# of comparisons	# of elements remaining
0	n
1	$n/2$
2	$n/4$
...	
$\log_2 n$	1

So binary search requires in the order of $\log n$ operations.

This is a huge improvement over linear search.

n	$\log_e n$
10	2.3
100	4.6
1000	6.9
1,000,000	13.8
1,000,000,000	20.7

Analysis of Recursive Fibonacci Method

The running time for computing `fibo`(n) can be specified as the recurrence relation:

$$T(n) = \begin{cases} C_2 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + C_1 & \text{otherwise} \end{cases}$$

It can be shown that $T(n)$ is $O((\frac{1+\sqrt{5}}{2})^n)$,
i.e. the running time of the method is *exponential*.

To get an intuition for this, note that the number of recursive calls for `fibo`(n) is close to the number of nodes in a full binary tree of height n, which is

$$1 + 2 + 2^2 + 2^3 + \ldots + 2^{n-1} = 2^n - 1$$

This is called a geometric progression. Since some operations must be performed for each call, the running time must be exponential.