OTE: Ohjelmointitekniikka
Programming Techniques

course homepage: `http://www.cs.uku.fi/~mnykanen/OTE/`

Week 04/2009

Exercise 1. The *Dutch Flag problem* is as follows: There is some function $colour(x)$ which tells the colour of the given element x: RED, WHITE or BLUE. The task is to permute the given input array a so that its RED elements come before its WHITE elements, which in turn come before its BLUE elements[1].

(a) Give a GCL algorithm for this problem.

(b) Argue the correctness of your algorithm.

Solution sketch. Let us denote as $coloured(i, j, c)$ the predicate "all the elements of $a[i \ldots j]$ have the same colour c" which can be expressed with the formula

$$\forall i \leq k \leq j : colour(a[k]) = c.$$

Then we can express our postcondition as

$$coloured(\text{lower}(a), r, \text{RED}) \wedge coloured(r+1, b-1, \text{WHITE}) \wedge coloured(b, \text{upper}(a), \text{BLUE}) \wedge perm(a, A)$$

where A is the original value of a.

We add a new variable w:

$$coloured(\text{lower}(a), r, \text{RED}) \wedge coloured(r+1, w-1, \text{WHITE}) \wedge coloured(b, \text{upper}(a), \text{BLUE}) \wedge perm(a, A) \wedge w = b$$

which means that the elements working area $a[w \ldots b-1]$ between the already WHITE and BLUE areas are still permitted to have any colours. Our strategy and bound is to shrink this working area.

Let us initialize all these three *coloured* statements to TRUE to make them our invariant:

$r\,,\ w\,,\ b\ :=\ \text{lower}(a) - 1\,,\ \text{lower}(a)\,,\ \text{upper}(a) + 1\,;$
do $w \neq b \rightarrow$
  shrink the working area
od

One simple way to shrink the working area is when its first element $a[w] =$ WHITE: Then it is enough to increment w to grow the already WHITE area.

[1] This problem was one of many posed by the late Edsger W. Dijkstra, who was of Dutch origin. The colours come from the Dutch flag.

The case $a[w] = \text{BLUE}$ is also straightforward: Swap it with the element $a[b-1]$ preceding the already BLUE area, and decrement b.

The remaining case is $a[w] = \text{RED}$: Swap it with the element $a[r+1]$ following the already RED area, and increment w. Note that this works whether the already WHITE area is empty or not: If it is empty, then $w = r + 1$ and this swaps a RED element with itself. If it is not empty, then the first element $a[r + 1]$ of the already WHITE area is swapped with the RED element $a[w]$, and so the already WHITE area moves one index further in a.

This gives:

$$
\begin{array}{l}
r\,,\ \ w\,,\ \ b\ :=\ \text{lower}(a) - 1\,,\ \ \text{lower}(a)\,,\ \ \text{upper}(a) + 1\,; \\
\textbf{do}\ \ w \neq b \rightarrow \\
\quad \textbf{if}\ \ a[w] = \text{WHITE} \rightarrow \\
\qquad w\ :=\ w + 1 \\
\quad [\!]\ \ a[w] = \text{BLUE} \rightarrow \\
\qquad a[w]\,,\ \ a[b-1]\,,\ \ r\ :=\ a[b-1]\,,\ \ a[w]\,,\ \ r - 1 \\
\quad [\!]\ \ a[w] = \text{RED} \rightarrow \\
\qquad a[w]\,,\ \ a[r+1]\,,\ \ w\ :=\ a[r+1]\,,\ \ a[w]\,,\ \ w + 1 \\
\textbf{od}
\end{array}
$$

(However, this solution makes more swaps than necessary. Unnecessary swapping can be reduced by examining also the end $a[b-1]$ of the working area in this way, or even adding another WHITE part before it.)

Exercise 2.

(a) How is the solution to the Dutch flag problem of Exercise 1 related to the *partition* procedure within the *quicksort* algorithm?

Solution sketch. After we have selected the pivot x, interpret the colours as follows:

$$
\begin{array}{ll}
\mathit{colour}(y) = \text{RED} & y < x \\
\mathit{colour}(y) = \text{WHITE} & y = x \\
\mathit{colour}(y) = \text{BLUE} & y > x
\end{array}
$$

Then the Dutch flag problem is to partition a and collect all the elements equal to x together as the WHITE middle part $a[r + 1 \ldots b - 1]$ (which is now guaranteed to be nonempty) instead of just the single element $a[m]$.

(b) Give a GCL *quicksort* algorithm which uses the solution to that problem.

Solution sketch. First modify the Dutch flag solution to process only the part $a[l \ldots u]$ being sorted in this recursive call: in its initialization, replace $\text{lower}(a)$ with l and $\text{upper}(a)$ with u.

Then replace the *partition* call with this modified Dutch flag.

Finally replace the recursive calls with $\mathit{quicksort}(l, r, a)$ and $\mathit{quicksort}(b, u, a)$ to skip sorting the WHITE part in vain.

Exercise 3.

(a) Rewrite the *quicksort* algorithm so that the array part sorted by the first recursive call is no longer than the part handled by the second call.

Solution sketch. Replace the recursive calls with the following code, which spells out this requirement:

```
if  m − l ≤ u − m →
    quicksort(l, m − 1, a);
    quicksort(m + 1, u, a)
[]  m − l ≥ u − m →
    quicksort(m + 1, u, a);
    quicksort(l, m − 1, a)
fi
```

(b) Explain how this version of the *quicksort* algorithm can be optimized further to use only logarithmic recursion depth.

Solution sketch. The last call in either **if** branch is a tail call, since it is the last thing this call does. Hence they can be eliminated. The resulting algorithm more than halves the length of the part to sort at each remaining recursive call, and this ensures logarithmic depth.

Exercise 4. The *median* of an unsorted input array $a[0 \dots 2 \cdot N]$ is the element which would appear at the middle position $a[N]$ after sorting a.

(a) Explain how you can simplify the *quicksort* algorithm into one which finds this median element faster than fully sorting the array a.

(HINT: After *partition*ing, just decide in which part $a[N]$ would be.)

Solution sketch. Let us generalize this problem to "what would be in $a[i]$ after sorting a", since it is no harder. Then this median problem is the case $i = (\text{lower}(a) + \text{upper}(a)) \operatorname{div} 2$.

The *quicksort* correctness proof reveals that the element $a[m]$ is not changed after *partition*. Hence we can just *quicksort* until we get $i = m$. Moreover, only the recursive call containing i is needed, the other one can be omitted. Hence:

```
proc ith(value l, i, u: ℤ;
         value a: array of τ; value y:τ);
if  l < u →
    partition(l, u, a, m);
    if  i = m →
        y := a[m]
    []  i < m →
        ith(l, i, m − 1, a, y);
    []  i > m →
        ith(m + 1, i, u, a)
    fi
fi
```

It **abort**s if i is not a valid index of a.

(b) Give an iterative version of this algorithm via tail recursion elimination.

Solution sketch. Just eliminate both these recursive calls, since they are tail calls.