Eliminating tail recursion

A recursive call is “tail recursion” if it is the last thing done before a return.

Many compilers remove tail recursion. (See optional slides.)

But we can do it ourselves by source-level transformation

```
procedure $f(p)$
    if $e$ then
        $S$
        $f(a)$
    else
        $T$
    end if
end $f$
```

```
procedure $f(p)$
    start: if $e$ then
        $S$
        $p := a$
        goto start
    else
        $T$
    end if
end $f$
```

```
procedure $f(p)$
    while $e$ do
        $S$
        $p := a$
    end while
    $T$
end $f$
```

For example we can optimize quickSort to get

```
procedure quickSort( var $a$ : array $\langle T \rangle$ ; $p, r$ : Int )
implements sort($a, p, r$)
    while $r - p > 1$ do
        val $i$ := any value from $\{p, ..r\}$
        val $x := a(i)$
        var $q$
        partition( $a, p, r, x, q$ )
        quickSort( $a, p, q$ )
        $p := q + 1$
    end while
end quickSort
```

Better yet we can ensure that the depth of recursion never exceeds $\lceil \log_2(r-p) \rceil$ by only using recursion for the smaller part of the array

```
procedure quickSort( var a : array ⟨T⟩ ; p, r : Int )
implements sort(a, p, r)
    while $r - p > 1$ do
        val $i :=$ any value from $\{p, ..r\}$
        val $x := a(i)$
        var $q$
        partition( $a, p, r, x, q$ )
        if $q - p < r - q - 1$ then
            quickSort( $a, p, q$ )
            $p := q + 1$
        else
            quickSort( $a, q + 1, r$ )
            $r := q$
        end if
    end while
end quickSort
```

In my Java implementation, this allowed me to sort 10^5 items.

Eliminating all recursion from quicksort

Quicksort (of an array) is an entirely top-down, "divide and conquer" algorithm in that once a problem instance is divided into smaller subinstances, there is no need to return to the original instance.

Therefore we can maintain a set of instances yet to be solved: the $WorkSet$.

```
procedure quickSortNR( var $a$ : array $\langle T \rangle$ )
implements sort($a, 0, a$.length)
    var $WorkSet$ : Set $\langle$Int $\times$ Int$\rangle$ := $\{(0, a.\text{length})\}$
    {inv: if we sort every segment in $WorkSet$ then $a$ will be sorted}
    while $WorkSet \neq \emptyset$ do
        var $p, r$
        $(p, r)$ := any element of $WorkSet$
        $WorkSet := WorkSet - \{(p, r)\}$
        if $r - p > 1$ then
            val $i$ := any value from $\{p, ..r\}$
            val $x := a(i)$
            var $q$
            partition( $a, p, r, x, q$ )
            $WorkSet := WorkSet \cup \{(p, q), (q + 1, r)\}$
        end if
    end while
end quickSortNR
```

By representing $WorkSet$ as a stack and pushing the "smaller" subinstance second, we can ensure that $|WorkSet|$ never exceeds $\lceil \log_2(a.\text{length}) \rceil$

Aside. Note that this algorithm is parallelizable.

General pattern for top-down

Top-down recursive

```
procedure $p(x)$
    if $x$ is is a leaf instance then
        solve $x$ by direct means
    else
        do some work on $x$
        break $x$ into smaller child instances $x_0, x_1, ..x_n$
        for $i \leftarrow \{0, ..n\}$ do $p(x_i)$
    end if
end $p$
```

Top-down workset algorithm

```
procedure $p(x)$
    postcondition $R$
    var $WorkSet := \{x\}$
    inv by doing all the tasks in the WorkSet, $R$ will be
    established.
    while $WorkSet \neq \emptyset$ do
        val $y :=$ any element of $WorkSet$
        $WorkSet := WorkSet - \{y\}$
        if $y$ is a leaf instance then
            solve $y$ by direct means
        else
            do some work on $y$
            break $y$ into smaller child instances $y_0, y_1, ..y_n$
            $WorkSet := WorkSet \cup \{y_0, y_1, ..y_n\}$
        end if
    end while
end $p$
```

Very few algorithms are purely top down.

Exercise: Find a variant expression for this loop or otherwise show it terminates.

Eliminating recursion from bottom-up

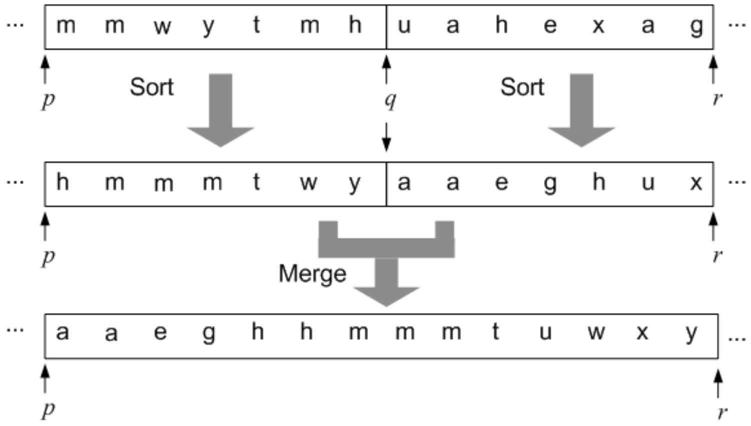
MergeSort is a bottom-up, “conquer and combine” algorithm

```
procedure mergeSort( var $a$ : array $\langle T \rangle$ ; $p, r$ : Int )
implements sort( $a, p, r$ )
    if $r - p > 1$ then
        var $q :=$ any number in $\{p + 1, .., r - 1\}$
        // For efficiency we pick $q$ near the middle
        $\{p < q < r\}$
        mergeSort( $a, p, q$ )    mergeSort( $a, q, r$ )
        merge( $a, p, q, r$ )
    end if
end mergeSort
```

where

```
procedure merge( var $a$ : array $\langle T \rangle$ ; $p, q, r$ : Int )
    precondition $p \le q \le r$ and segment $a[p, ..q]$ is sorted
    and $a[q, ..r]$ is sorted
    changes $a$ (but only permuting $a\,[p, ..r]$)
    postcondition segment $a[p, ..r]$ is sorted
```

Exercise: Implement $merge$.



A non-recursive merge-sort

Consider any directed accylic graph (DAG) T of pairs such that

- $(0, a.\mathrm{length})$ is in the DAG
- Pairs $(i, i+1)$ are leaves, for all $i \in \{0, ..a.\mathrm{length}\}$
- Every nonleaf (p, r) has exactly 2 children (p, q) and (q, r), for some q, such that $p < q < r$.

```
procedure mergeSortNR( var $a$ : array $\langle T \rangle$ )
    var $Solved : Set := \emptyset$
    inv All pairs in $Solved$ represent sorted regions of the array
    while $(0, a.\mathrm{length}) \notin Solved$
        let $(p, r) \notin Solved$ such that $(p, r)$ is a leaf
            or both children of $(p, r)$ are in Solved
        if $(p, r)$ is a leaf
            do nothing
        else
            Let $q$ be such that $(p, q)$ and $(q, r)$ are the children
            of $(p, r)$
            merge( $a, p, q, r$ )
        end if
        $Solved := Solved \cup \{(p, r)\}$
    end while
end $p$
```

General pattern for bottom-up

Bottom-up recursive conquer and combine

The general form of a recursive, bottom-up algorithm

```
procedure $p(x)$
    if $x$ is a leaf instance then
        solve $x$ by direct means
    else
        break $x$ into smaller child instances $x_0, x_1, ..x_n$
        for $i \leftarrow \{0, ..n\}$ do $p(x_i)$
        combine the solution for the children to solve $x$
    end if
end
```

As with the top-down algorithm, the algorithm defines a DAG of instances.

Bottom-up nonrecursive conquer and combine

If we can anticipate which instances will be in the DAG, we can solve the instances nonrecursively bottom-up.

```
procedure $p(x)$
    Consider a DAG $T$ of instance that contains instance $x$.
    var $Solved : Set := \emptyset$
    inv: All instance in $Solved$ are solved
    while $x \notin Solved$
        pick an instance $y \notin Solved$ that all of $y$'s children are in $Solved$
        if $y$ is a leaf
            solve $y$ directly
        else
            combine the solutions to $y$'s children so that $y$ is solved
        end if
        $Solved := Solved \cup \{y\}$
    end while
end $p$
```

Layer-by-layer bottom-up

Often bottom up problems can be solved one layer at a time, starting with the leaves. This will often remove the need to keep track of solved instances

```
procedure $p(x)$
    Consider a DAG $T$ of subinstances that contains instance $x$.
    give each node of $T$ a natural ‘layer number’ so that children have lower numbers than parents.
    var $k := 0$
    inv: all nodes numbered below $k$ have been solved
    while the root $x$ is not solved
        solve all instances with $k$ as layer number
        $k := k + 1$
    end while
end $p$
```

Layer-by-layer merge-sort.

The DAG is a tree such that

- Layer 0 consists of intervals $(0, 1)$, $(1, 2)$, $\cdots$
- Layer 1 consists of intervals $(0, 2)$, $(2, 4)$, $\cdots$
- Layer 2 consists of intervals $(0, 4)$, $(4, 8)$, $\cdots$
- etc

Since layer 0 is already solved, we start with solving layer 1

```
procedure mergeSortNR( var $a$ : array $\langle T\rangle$ )
implements sort( $a, 0, a.\mathrm{length}$ )
    var $grain := 1$
    // inv.: each of the segments $a[0, ..grain]$,
    // $a[grain, ..2 \times grain]$, $a\,[2 \times grain, ..3 \times grain]$, etc. on
    // up to and including $a[\left\lfloor \frac{a.length}{grain} \right\rfloor \times grain, ..a.\mathrm{length}\}]$
    // is sorted, and $grain > 0$
    $\{grain > 0 \wedge a$ is a permutation of $a_0 \wedge \forall i \in \mathbb{N}\cdot$
          $a[\mathrm{cap}(i \times grain), .. \mathrm{cap}\,((i + 1) \times grain)]$ is sorted,
          where $\mathrm{cap}(j) = \min(j, a.\mathrm{length})\}$
    while $grain < a.\mathrm{length}$ do
        var $p := 0$
        while $p < a.\mathrm{length}$ do
            val $q := \min(p + grain, a.\mathrm{length})$
            val $r := \min(q + grain, a.\mathrm{length})$
            merge( $a, p, q, r$ )
            $p := r$
        end while
        $grain := grain \times 2$
    end while
end mergeSortNR
```

In mergesort we are able to anticipate the subinstances that need to be solved prior to solving the superinstances.

However the tree of sub-instances used by the nonrecursive merge-sort may differ from the recursive version.

- Usually the recursive version attempts to balance the split. E.g. if $a.\mathrm{length} = 17$, the final merge is between regions of lengths 8 and 9.
- The bottom up version always merges regions of length $=$ some power of 2. E.g. if $a.\mathrm{length} = 17$, the final merge is between regions of lengths 16 and 1.

Note that this algorithm's inner loop is parallelizable.