COSC 2123/1285 Algorithms and Analysis
Tutorial 5
Divide and Conquer Algorithmic Paradigm

Objective

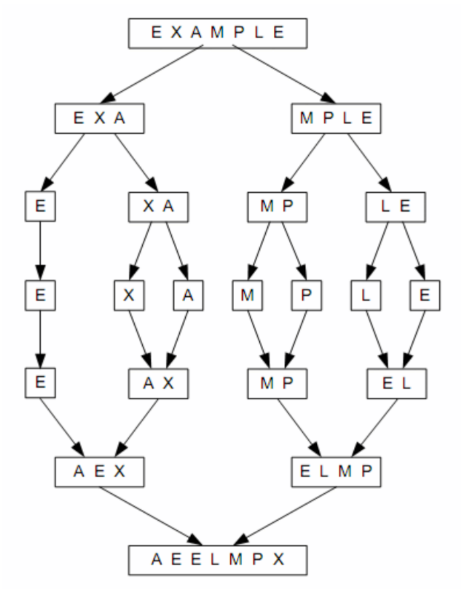
Students who complete this tutorial should:

- Understand the concepts of divide and conquer.
- Be familiar with the way this concept can be applied to sorting problems.

Questions

5.1.6 Apply mergesort to sort the list `E, X, A, M, P, L, E` in alphabetical order.

Answer: Here is a trace of mergesort applied to the input given:



5.2.1 Apply quicksort to sort the list `E, X, A, M, P, L, E` in alphabetical order.

Answer:

0	1	2	3	4	5	6
$\mathbf{E}$	$\overset{i}{X}$	A	M	P	L	$\overset{j}{E}$
$\mathbf{E}$	E	$\overset{j}{A}$	$\overset{i}{M}$	P	L	X
A	E	$\mathbf{E}$	M	P	L	X
$\mathbf{A}$	$\overset{i\,j}{E}$					
$\overset{j}{\mathbf{A}}$	$\overset{i}{E}$					
$\mathbf{A}$	E					
	E					
			$\mathbf{M}$	$\overset{i}{P}$	L	$\overset{j}{X}$
			$\mathbf{M}$	$\overset{i}{P}$	$\overset{j}{L}$	X
			$\mathbf{M}$	$\overset{i}{L}$	$\overset{j}{P}$	X
			$\mathbf{M}$	$\overset{j}{L}$	$\overset{i}{P}$	X
			L	$\mathbf{M}$	P	X
			L			
					$\mathbf{P}$	$\overset{i\,j}{X}$
					$\overset{j}{\mathbf{P}}$	$\overset{i}{X}$
					$\mathbf{P}$	X
						X

5.1.7 Is mergesort a stable sorting algorithm?

Answer: Mergesort is **stable**, as long as in the merging process of two identical elements with indices i and j, with $i < j$, that order is perserved. Assume that we have two elements of the same value in positions i and j, $i < j$, in a subarray before its two (sorted) halves are merged. If these two elements are in the same half of the subarray, their relative ordering will stay the same after the merging because the elements of the same half are processed by the merging operation in the FIFO fashion. Consider now the case when $A[i]$ is in the first half while $A[j]$ is in the second half. $A[j]$ is placed into the new array either after the first half becomes empty (and, hence, $A[i]$ has been already copied into the new array) or after being compared with some key $k > A[j]$ of the first half. In the latter case, since the first half is sorted before the merging begins, $A[i] = A[j] < k$ cannot be among the unprocessed elements of the first half. Hence, by the time of this comparison, $A[i]$ has been already copied into the new array and therefore will precede $A[j]$ after the merging operation is completed.

5.2.3 Is quicksort a stable sorting algorithm?

Answer: Quicksort is **not stable**. As a counterexample, consider its performance on a two-element array of equal values. Another counter example are the 2 "E"s in the EXAMPLE array.

5.1.1

a Write a pseudocode for a divide-and-conquer algorithm for finding a position of the largest element in an array of n numbers.

b What will be your algorithm's output for arrays with several elements of the largest value?

c Set up and solve a recurrence relation for the number of key comparisons made by your algorithm.

d How does this algorithm compare with the brute-force algorithm for this problem?

Answer:

a Call $MaxIndex(A[0 \dots n-1])$ – see Algorithm 1.

Algorithm 1 $MaxIndex(A[0 \dots n-1])$

```
Input: A portion of array $A[0 \dots n-1]$ between the indices $l$ and $r$ ( $l \leq r$ )
Output: The index of the largest element in $A[l \dots r]$
if $l == r$ then
    return $l$
else
    $temp1 = MaxIndex(A[l \dots \lfloor (l+r)/2 \rfloor])$
    $temp2 = MaxIndex(A[\lfloor (l+r)/2 \rfloor + 1 ... r])$
    if $A[temp1] \geq A[temp2]$ then
        return $temp1$
    else
        return $temp2$
    end if
end if
```

b This algorithm returns the index of the leftmost largest element.

c The recurrence for the number of element comparisons is:

$$C(n) = C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1 \text{ for } n > 1, C(1) = 0$$

The following isn't examinable, but for your interest. Solving it by backward substitutions for $n = 2^k$ yields the following (Note: We use smoothing rule so we can model $n \to n/2$ using $2^k \to 2^{k-1}$):

$$C(n) = C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1$$
$$C(2^k) = C(2^{k-1}) + C(2^{k-1}) + 1 = 2C(2^{k-1}) + 1$$

$$\begin{aligned}
C(2^k) &= 2C(2^{k-1}) + 1 \\
&= 2[2C(2^{k-2}) + 1] + 1 = 2^2C(2^{k-2}) + 2 + 1 \\
&= 2^2[2C(2^{k-3}) + 1] + 2 + 1 = 2^3C(2^{k-3}) + 2^2 + 2 + 1 \\
&= \dots \\
&= 2^iC(2^{k-i}) + 2^{i-1} + 2^{i-2} + \dots + 2 + 1 \\
&= 2^kC(2^{k-k}) + 2^{k-1} + 2^{k-2} + \dots + 2 + 1 \\
&= 2^{k-1} + 2^{k-2} + \dots + 2 + 1 \\
&= \sum_{i=0}^{k-1} 2^i \\
&= 2^k - 1
\end{aligned}$$

Substituting 2^k with n results in $n-1$. Therefore $C(2^k) = C(n) = 2^k - 1 = n-1$. We can verify that $C(n) = n-1$ satisfies, in fact, the recurrence for every value of $n > 1$ by substituting it into the recurrence equation and considering separately the **even** ($n = 2i$) and **odd** ($n = 2i + 1$) **cases**:

Let $n = 2i$, where $i > 0$. Then the left-hand side of the recurrence equation is $n - i = 2i - 1$. The right-hand side is

$$\begin{aligned}
C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1 &= C(\lceil 2i/2 \rceil) + C(\lfloor 2i/2 \rfloor) + 1 \\
&= 2C(i) + 1 = 2(i-1) + 1 = 2i - 1
\end{aligned}$$

which is the same as the left-hand side.

Let $n = 2i + 1$ where $i > 0$. Then the left-hand side of the recurrence equation is $n - 1 = 2i$. The right-hand side is

$$\begin{aligned}
C(\lceil n/2 \rceil) + C(\lfloor n/2 \rfloor) + 1 &= C(\lceil 2i + 1/2 \rceil) + C(\lfloor 2i + 1/2 \rfloor) + 1 \\
&= C(i+1) + C(i) + 1 = (i + 1 - 1) + (i - 1) + 1 = 2i
\end{aligned}$$

which is the same as the left-hand side in this case, too.

d A simple standard scan through the array in question requires the same number of key comparisons but avoids the overhead associated with recursive calls.

5.2.11 *Nuts and bolts* You are given a collection of n bolts of different widths and n corresponding nuts. You are allowed to try a nut and bolt together, from which you can determine whether the nut is larger than the bolt, smaller than the bolt, or matches the bolt exactly. However, there is no way to compare two nuts together or two bolts together. The problem is to match each bolt to its nut. Design an algorithm for this problem with average-case efficiency of $\Theta(n \log n)$.

Answer: Use the partition idea.

Randomly select a nut and try each of the bolts for it to find the matching bolt and separate the bolts that are smaller and larger than the selected nut into two disjoint sets. Then try each of the unmatched nuts against the matched bolt to separate those that are larger from those that are smaller than the bolt. As a result, we've identified a matching pair and partitioned the remaining nuts and bolts into two smaller independent instances of the same problem. The average number of nut-bolt comparisons $C(n)$ is defined by the recurrence very similar to the one for quicksort in Section 5.2 of textbook:

$$C(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(2n-1) + C(s) + C(n-1-s)],\ C(1) = 0,\ C(0) = 0.$$

The solution to this recurrence can be shown to be $O(n \log(n))$.