2 Divide-and-Conquer

We use quicksort as an example for an algorithm that follows the divide-and-conquer paradigm. It has the reputation of being the fasted comparison-based sorting algorithm. Indeed it is very fast on the average but can be slow for some input, unless precautions are taken.

The algorithm. Quicksort follows the general paradigm of divide-and-conquer, which means it **divides** the unsorted array into two, it **recurses** on the two pieces, and it finally **combines** the two sorted pieces to obtain the sorted array. An interesting feature of quicksort is that the divide step separates small from large items. As a consequence, combining the sorted pieces happens automatically without doing anything extra.

```
void QUICKSORT(int ℓ, r)
  if ℓ < r then m = SPLIT(ℓ, r);
                QUICKSORT(ℓ, m − 1);
                QUICKSORT(m + 1, r)
  endif.
```

We assume the items are stored in $A[0..n-1]$. The array is sorted by calling QUICKSORT$(0, n-1)$.

Splitting. The performance of quicksort depends heavily on the performance of the split operation. The effect of splitting from ℓ to r is:

- $x = A[\ell]$ is moved to its correct location at $A[m]$;
- no item in $A[\ell..m-1]$ is larger than x;
- no item in $A[m+1..r]$ is smaller than x.

Figure 37 illustrates the process with an example. The nine items are split by moving a pointer i from left to right and another pointer j from right to left. The process stops when i and j cross. To get splitting right is a bit delicate, in particular in special cases. Make sure the algorithm is correct for (i) x is smallest item, (ii) x is largest item, (iii) all items are the same.

```
int SPLIT(int ℓ, r)
  x = A[ℓ]; i = ℓ; j = r + 1;
  repeat repeat i++ until x ≤ A[i];
         repeat j-- until x ≥ A[j];
         if i < j then SWAP(i, j) endif
  until i ≥ j;
  SWAP(ℓ, j); return j.
```

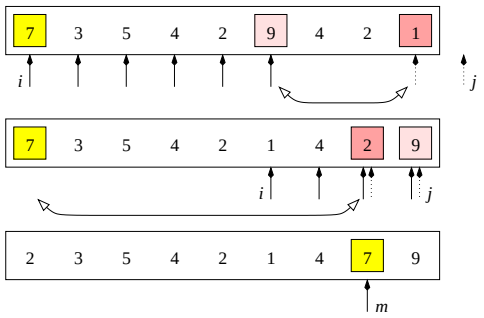

Figure 1: First, i and j stop at items 9 and 1, which are then swapped. Second, i and j cross and the pivot, 7, is swapped with item 2.

Special cases (i) and (iii) are ok but case (ii) requires a stopper at $A[r+1]$. This stopper must be an item at least as large as x. If $r < n-1$ this stopper is automatically given. For $r = n-1$, we create such a stopper by setting $A[n] = +\infty$.

Running time. The actions taken by quicksort can be expressed using a binary tree: each (internal) node represents a call and displays the length of the subarray; see Figure 92. The worst case occurs when A is already

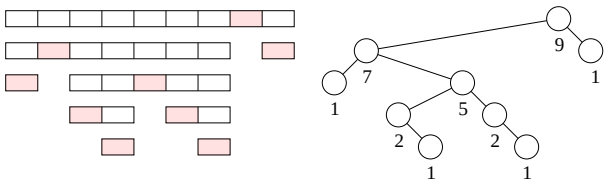

Figure 2: The total amount of time is proportional to the sum of lengths, which are the numbers of nodes in the corresponding subtrees. In the displayed case this sum is 29.

sorted. In this case the tree degenerates to a list without branching. The sum of lengths can be described by the following recurrence relation:

$$T(n) \;=\; n + T(n-1) \;=\; \sum_{i=1}^{n} i \;=\; \binom{n+1}{2}.$$

The running time in the worst case is therefore in $O(n^2)$.

In the best case the tree is completely balanced and the sum of lengths is described by the recurrence relation

$$T(n) \;=\; n + 2 \cdot T\left(\frac{n-1}{2}\right).$$

If we assume $n = 2^k - 1$ we can rewrite the relation as

$$\begin{aligned} U(k) &= (2^k - 1) + 2 \cdot U(k-1) \\ &= (2^k - 1) + 2(2^{k-1} - 1) + \ldots + 2^{k-1}(2-1) \\ &= k \cdot 2^k - \sum_{i=0}^{k-1} 2^i \\ &= 2^k \cdot k - (2^k - 1) \\ &= (n+1) \cdot \log_2(n+1) - n. \end{aligned}$$

The running time in the best case is therefore in $\mathrm{O}(n \log n)$.

Randomization. One of the drawbacks of quicksort, as described until now, is that it is slow on rather common almost sorted sequences. The reason are pivots that tend to create unbalanced splittings. Such pivots tend to occur in practice more often than one might expect. Human and often also machine generated data is frequently biased towards certain distributions (in this case, permutations), and it has been said that 80% of the time or more, sorting is done on either already sorted or almost sorted files. Such situations can often be helped by transferring the algorithm's dependence on the input data to internally made random choices. In this particular case, we use randomization to make the choice of the pivot independent of the input data. Assume $\textsc{Random}(\ell, r)$ returns an integer $p \in [\ell, r]$ with uniform probability:

$$\mathrm{Prob}[\textsc{Random}(\ell, r) = p] = \frac{1}{r - \ell + 1}$$

for each $\ell \le p \le r$. In other words, each $p \in [\ell, r]$ is equally likely. The following algorithm splits the array with a random pivot:

```
int rSplit(int ℓ, r)
  p = Random(ℓ, r); Swap(ℓ, p);
  return Split(ℓ, r).
```

We get a *randomized* implementation by substituting rSplit for Split. The behavior of this version of quicksort depends on p, which is produced by a random number generator.

Average analysis. We assume that the items in $A[0..n-1]$ are pairwise different. The pivot splits A into

$$A[0..m-1], \quad A[m], \quad A[m+1..n-1].$$

By assumption on function rSplit, the probability for each $m \in [0, n-1]$ is $\frac{1}{n}$. Therefore the average sum of array lengths split by QuickSort is

$$T(n) = n + \frac{1}{n} \cdot \sum_{m=0}^{n-1} \left(T(m) + T(n-m-1)\right).$$

To simplify, we multiply with n and obtain a second relation by substituting $n-1$ for n:

$$n \cdot T(n) = n^2 + 2 \cdot \sum_{i=0}^{n-1} T(i), \qquad (1)$$

$$(n-1) \cdot T(n-1) = (n-1)^2 + 2 \cdot \sum_{i=0}^{n-2} T(i). \qquad (2)$$

Next we subtract (2) from (1), we divide by $n(n+1)$, we use repeated substitution to express $T(n)$ as a sum, and finally split the sum in two:

$$\begin{aligned} \frac{T(n)}{n+1} &= \frac{T(n-1)}{n} + \frac{2n-1}{n(n+1)} \\ &= \frac{T(n-2)}{n-1} + \frac{2n-3}{(n-1)n} + \frac{2n-1}{n(n+1)} \\ &= \sum_{i=1}^{n} \frac{2i-1}{i(i+1)} \\ &= 2 \cdot \sum_{i=1}^{n} \frac{1}{i+1} - \sum_{i=1}^{n} \frac{1}{i(i+1)}. \end{aligned}$$

Bounding the sums. The second sum is solved directly by transformation to a telescoping series:

$$\begin{aligned} \sum_{i=1}^{n} \frac{1}{i(i+1)} &= \sum_{i=1}^{n} \left(\frac{1}{i} - \frac{1}{i+1}\right) \\ &= 1 - \frac{1}{n+1}. \end{aligned}$$

The first sum is bounded from above by the integral of $\frac{1}{x}$ for x ranging from 1 to $n+1$; see Figure 3. The sum of $\frac{1}{i+1}$ is the sum of areas of the shaded rectangles, and because all rectangles lie below the graph of $\frac{1}{x}$ we get a bound for the total rectangle area:

$$\sum_{i=1}^{n} \frac{1}{i+1} < \int_1^{n+1} \frac{\mathrm{d}x}{x} = \ln(n+1).$$

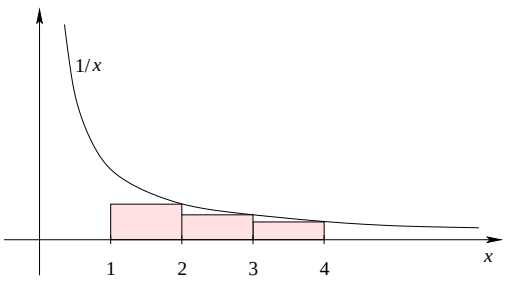

Figure 3: The areas of the rectangles are the terms in the sum, and the total rectangle area is bounded by the integral from 1 through $n+1$.

We plug this bound back into the expression for the average running time:

$$\begin{aligned} T(n) &< (n+1) \cdot \sum_{i=1}^{n} \frac{2}{i+1} \\ &< 2 \cdot (n+1) \cdot \ln(n+1) \\ &= \frac{2}{\log_2 e} \cdot (n+1) \cdot \log_2(n+1). \end{aligned}$$

In words, the running time of quicksort in the average case is only a factor of about $2/\log_2 e = 1.386\ldots$ slower than in the best case. This also implies that the worst case cannot happen very often, for else the average performance would be slower.

Stack size. Another drawback of quicksort is the recursion stack, which can reach a size of $\Omega(n)$ entries. This can be improved by always first sorting the smaller side and simultaneously removing the tail-recursion:

```
void QUICKSORT(int ℓ, r)
  i = ℓ; j = r;
  while i < j do
    m = RSPLIT(i, j);
    if m − i < j − m
      then QUICKSORT(i, m − 1); i = m + 1
      else QUICKSORT(m + 1, j); j = m − 1
    endif
  endwhile.
```

In each recursive call to QUICKSORT, the length of the array is at most half the length of the array in the preceding call. This implies that at any moment of time the stack contains no more than $1 + \log_2 n$ entries. Note that without removal of the tail-recursion, the stack can reach $\Omega(n)$ entries even if the smaller side is sorted first.

Summary. Quicksort incorporates two design techniques to efficiently sort n numbers: divide-and-conquer for reducing large to small problems and randomization for avoiding the sensitivity to worst-case inputs. The average running time of quicksort is in $\mathrm{O}(n \log n)$ and the extra amount of memory it requires is in $\mathrm{O}(\log n)$. For the deterministic version, the average is over all $n!$ permutations of the input items. For the randomized version the average is the expected running time for *every* input sequence.