Algorithms and Data Structures № 4

Introduction

Divide and conquer algorithms

1. Divide the problem into a number of sub-problems (let the dividing time be $D(n)$).
2. Conquer the sub-problems by solving them recursively (there are a subproblems to solve, each of size n/b, if a problem of size n takes $T(n)$ time to solve, then we spend $a \cdot T(n/b)$ time solving subproblems).
3. *Base case:* If the sub-problems are small enough, just solve them by brute force ($\Theta(1)$).
4. Combine the sub-problem solutions to give a solution to the original problem ($C(n)$).

$$T(n) = \begin{cases} \Theta(1) & \text{for } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

Merge sort

- Dance: `https://www.youtube.com/watch?v=XaqR3G_NVoo&t=144s`
- (Pseudo)code - without the **merge** procedure:

```
1 MergeSort(arr, left, right):
2     if left > right
3         return
4     mid = (left+right)/2
5     mergeSort(arr, left, mid)
6     mergeSort(arr, mid+1, right)
7     merge(arr, left, mid, right) // TODO: merge procedure
8 end
```

Problem 1

- Complete the merge sort algorithm
- Extra tasks:
  - `https://www.hackerrank.com/challenges/merge-two-sorted-linked-lists/problem`
  - `https://www.hackerrank.com/challenges/ctci-merge-sort/problem`

Quicksort

Quicksort algorithm - as mergesort - is based on the divive-and-conquer paradigm.

These are the 3 steps applied to *quicksort* a table: A[p..r]:

- **Divide:** A list (a table) A[p..r] is divided (including some element swapping into 2 (possibly empty) sub-lists (sub-tables) A[p..q-1] and A[[q+1..r], such that **each element of A[p..q-1] is not greater than element A[q]**, that, in turn, is not greater than **each element of A[q+1..r]**. Calculating the index q is part of this procedure.
- **Conquer:** The sublist (sub-tables) A[p..q-1] and A[q+1..r] are sorted by recursively calling the quicksort algorithm
- **Merge:** Because the sublists (sub-tables) are already sorted, there is no need to do anything to "merge" the solutions. In fact, as a result the table A[p..r] is already sorted.

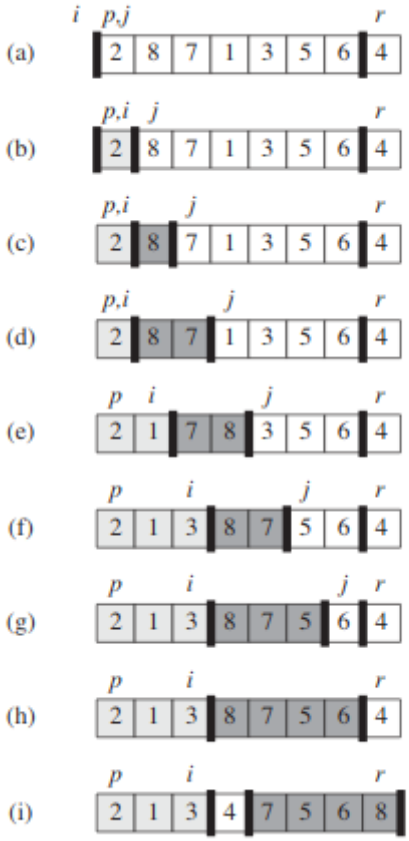
This is a general procedure for Quicksort algorithm:

```
1  QUICKSORT(A, p, r)
2  if p<r
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q-1)
5      QUICKSORT(A, q+1, r)
```

To sort a table A, we call QUICKSORT(A, 1, A.length). The key procedure here is the PARTITION that swaps elements of the tables and returns the index q of division.
The `PARTITION` function according to Lomuto:

```
1  PARTITION(A, p, r)
2  x = A[r]
3  i = p - 1
4  for j = p to r -1
5      if A[j] <= x
6          i = i + 1
7          swap A[i] with A[j]
8  swap A[i+1] with A[r]
9  return i + 1
```

The following illustration shows PARTITION on a 8-elements table:



Problem 1

Illustrate as above the behaviour of PARTITION function for the table:

A = <13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11>.

Problem 2

Implement the Lomuto PARTITION function on the Hackerrank platform: `https://www.hackerrank.com/challenges/quicksort3/problem`.

Hoare Partition

Originally, PARTITION in Quicksort was proposed by Hoare and looks a little bit different (see below). This version is a little bit more involved, but also more efficient – on average, it performs 3 times less swaps than Lomuto partition.

```
1 HOARE_PARTITION(A, p, r)
2 x = A[p]
3 i = p - 1
4 j = r + 1
5 while TRUE
```

```
6       repeat
7           j = j - 1
8       until A[j] <= x
9       repeat
10          i = i + 1
11      until A[i] >= x
12      if i < j
13          swap A[i] with A[j]
14      else return j
```

Oberve the HOARE PARTITION in action: `https://www.youtube.com/watch?v=ywWBy6J5gz8`

Problem 3

Illustrate the behaviour of HOARE_PARTITION function for the table:
A = <13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11>.

Problem 4

Implement the Hoare Partition on Hackerrank: `https://www.hackerrank.com/challenges/quicksort1/problem`.

Problem 5

Hoare partition makes the quicksort algorithm *unstable* which means some elements that didn't have to be swapped may be swapped (e.g. elements of the same key value or elements within a sub-problem at a certain level of recursion). Think of a partition function that would *keep* the order of the elements whenever possible. Hint: you can copy the elements into new sublists/-subtables when creating sub-problems.

- basic task: write pseudocode for such a "stable" partition function
- extra credit: implement the solution on Hackerrank: `https://www.hackerrank.com/challenges/quicksort2/problem`

Complexity analysis of Quicksort

The running time of quicksort depends on whether the partitioning is balanced or unbalanced, which in turn depends on which elements are used for partitioning. If the partitioning is balanced, the algorithm runs asymptotically as fast as merge sort. If the partitioning is unbalanced, however, it can run asymptotically as slowly as insertion sort. Let's informally investigate how quicksort performs under the assumptions of balanced versus unbalanced partitioning

- **Worst case**: Partition of a table size n leads to a subproblem of a size $n-1$ and 0 (or 1 depending on the partition method). If we have this case on each level, then:

$$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$$

- **Best case**: Partition of a table of size n generates subproblems of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil - 1$. Then

$$T(n) = 2T(n/2) + \Theta(n)$$

  From Master Theorem we have that in this case: $T(n) = \Theta(n\,log(n))$.

- **Average / expected case**: In practice, an expected case yields complexity closer to the best, and not the worst case (see a detailed analysis in Cormen book).

The are variants of choosing the pivot element (besides the first and the last element):

- **middle element**: `pivot := A[ floor((hi + lo) / 2) ]`
- **randomized partition**: first select an element randomly, then move it to the first/last position and start `Partition` selecting this newly swapped element:
- **median-of-3**: choose randomly 3 elements and select the middle of them, put it at the beginning/end and call the partition function

Problem 6

Explain why time complexity of PARTITION for a sub-problem of size n is $\Theta(n)$.

Problem 7

Show (by iteration method) that for $T(n) = T(n-1) + \Theta(n)$ the solution is $T(n) = \Theta(n^2)$.

Problem 8

What is the time complexity of QUICKSORT, if all the elements are equal?

Problem 9

Show that the time complexity of QUICKSORT is $\Theta(n^2)$, if all the elements are sorted in a reverse order.