ALGO: Exercises for Tutorial Classes

October 21–25, 2019. EPITA. ING1

Send any mistake to Alexandre Duret-Lutz <adl@lrde.epita.fr>.

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOGN)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): //COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

Picture by Randall Munroe (`http://xkcd.com/1185/`) — Creative Commons by-nc 2.5 License.

Day 1

1.1 Small Loops

How many lines are printed by the following loops? Give your answer as a function of `N`.

1.
   ```
   for (int i = 0; i < N; ++i)
     for (int j = 1; j <= N; ++j)
       puts("Loop 1");
   ```

2.
   ```
   for (int i = 0; i < N; ++i)
     for (int j = N/2; j > 0; --j)
       puts("Loop 2");
   ```

3.
   ```
   for (int i = 0; i < N; ++i)
     for (int j = i; j >= 0; j--)
       puts("Loop 3");
   ```

4.
   ```
   for (int i = 0; i < N; ++i)
     for (int j = 1; j < N; j *= 2)
       puts("Loop 4");
   ```

   Hint: rewrite the inner loop so it increments by 1 while still performing the same number of iterations.

5.
   ```
   for (int i = 0; i < N; ++i)
     for (int j = i; j < N - 2; j++)
       puts("Loop 5");
   ```

6.
   ```
   for (int i = 0; i < N; ++i)
     for (int j = i; j >= 0; j -= 2)
       puts("Loop 6");
   ```

   Hint: rewrite the inner loop so it increments by 1. Then distinguish between odd and even `N`.

1.2 House

How many '`#`' are printed by '`house(n)`'? Give your answer as a function of `n`.

```
void house(unsigned n)
{
  for (unsigned y = 0; y < n; ++y)
    {
      for (unsigned x = 1; x < n - y; ++x)
        putchar(' ');
      for (unsigned x = n - y; x <= n + y; ++x)
        putchar('#');
      putchar('\n');
    }
  for (unsigned y = 1; y < n; ++y)
    {
      for (unsigned x = 1; x < 2 * n; ++x)
        putchar('#');
      putchar('\n');
    }
}
```

1.3 Simple Computations

1. Sum the first 1000 odd natural numbers.
2. Compute $\log_{10}(42)$ to two decimal places by hand. You may use a pocket calculator, but assume that its $\boxed{\text{log}}$ key is broken and cannot be used.
3. ✽ Compute $\log_2(42)$ similarly.

1.4 Binary Trees

Consider non-empty *full* binary trees (i.e., *internal nodes* always have two children, and *leaf nodes* have no children). Let h denote the *height* of the tree (a single-node tree has $h = 0$), n_i the number of internal nodes, ℓ the number of leaves, and $n = n_i + \ell$ the total number of nodes.

1. Express n_i as a function of ℓ.
2. Give lower and upper bounds for:
   - ℓ as a function of h,
   - n_i as a function of h,
   - n as a function of h.

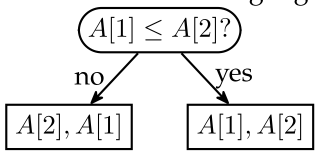
   Keep in mind that (even full) binary trees are not always *balanced*.
3. Deduce some lower bounds for:
   - h as a function of ℓ,
   - h as a function of n_i,
   - h as a function of n.

   Since h is an integer, use $\lfloor\cdot\rfloor$ or $\lceil\cdot\rceil$ when appropriate.

1.5 Decision Tree ✽

Let's construct a tree that represents some sorting algorithm for an array of size n. Each internal node is labeled by a question of the form "$A[i] \le A[j]$?" for some given i and j. The left child is used when the answer is negative, while the right child is used when the answer is positive. Each leaf is labeled by a possible answer of the sorting algorithm, i.e., the order in which values should be rearranged to be sorted. Several leaves may have identical labels.

For instance here is a sorting algorithm for an array of 2 values:



1. Devise a decision tree for sorting an array of 3 values.
2. What is the minimum number of leaves needed to sort an array of n values?
3. Deduce the minimal height of a decision tree for sorting n values.
4. What does this teach us about the complexity of *comparison sorting algorithms* (i.e., sorting algorithms based on comparisons of the values to sort)?

Day 2

You will have to implement the algorithms from exercises 2.1 and 2.2 this afternoon.

2.1 Binary Search

BinarySearch(A, b, e, v) takes a sorted array A, and looks up a value v in the semi-open range of A delimited by indices $b \geq 0$ (included if $b < e$) and e (always excluded).

If there exists an index p such that $b \leq p < e$ and $A[p] = v$, BinarySearch returns p. If there is no such index, BinarySearch returns the index i of the value before which v should be inserted if we wanted to add this value while keeping A ordered. In other words, when v is not found in A between b and e, the returned value i is such that:

$$\begin{cases} b = e \text{ or } v < A[b] & \text{if } i = b \\ A[i-1] < v < A[i] & \text{if } b < i < e \\ b = e \text{ or } A[e-1] < v & \text{if } i = e \end{cases}$$

```
BinarySearch($A, b, e, v$)
if $b < e$ then
    $m \leftarrow \lfloor (b+e)/2 \rfloor$
    if $v = A[m]$ then
        return $m$
    else
        if $v < A[m]$ then
            return BinarySearch($A, b, m, v$)
        else
            return BinarySearch($A, m+1, e, v$)
else
    return $b$
```

1. Prove that $\lfloor (b+e)/2 \rfloor = b + \lfloor (e-b)/2 \rfloor$ when b and e are natural numbers.
2. Although the previous equality is true from a mathematical standpoint, it does not hold anymore if the operations are implemented using a low level type such as `unsigned int`. Suggest two `unsigned int` values `a` and `b` verifying `a<b` but such that `assert((a+b)/2 == (a + (b-a)/2))` would signal a problem. Which one of these two expressions should you use when implementing this algorithm?
3. Give an **upper bound** of $T(n)$, the time complexity of BinarySearch expressed as a function of the array size $n = e - b$. Your expression should be recursive, involving $T(\lfloor n/2 \rfloor)$.
4. Solve this recursive expression by substituting $T(\ldots)$ by its definition until you reach $T(1)$. You may assume that n is a power of 2.
5. Are the recursive calls to BinarySearch in this function *tail calls*?
6. Perform a *tail call elimination* in BinarySearch as if you were a compiler (i.e., using `goto`s), then refactor it to be more readable to the human (getting rid of the `goto`s).
7. What is the complexity of your non-recursive version of `BinarySearch`?

2.2 Insertion Sort + Binary Search

Let's consider again the insertion sort as seen in the lecture, where A is an array of n objects (with indices starting at 0) that can be compared.

```
InsertionSort($A, n$)
for $i \leftarrow 1$ to $n-1$ do
    $key \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j \geq 0$ and $A[j] > key$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow key$
```

1. What is the number of comparisons performed in the worst and best cases? We only want to count comparisons between objects, i.e., the number of times $A[j] > key$ is executed.
2. What is the total number of assignments of objects? (Lines 2, 5, and 7.)
3. Write a variant of InsertionSort that uses BinarySearch to find where to insert key, and then shift all objects to the right of this position to make some room for the new value.
4. ❆ What is the number of comparisons and assignments in this new algorithm? (Did you also account for the number of comparisons in BinarySearch?)

2.3 The Θ Notation

By definition

$$\Theta(g(n)) = \left\{ f(n) \,\middle|\, \begin{array}{l} \exists c_1 > 0, \exists c_2 > 0, \exists n_0 \in \mathbb{N}, \\ \forall n \geq n_0,\ c_1 g(n) \leq f(n) \leq c_2 g(n) \end{array} \right\}$$

For convenience, we often write $f(n) = \Theta(g(n))$ instead of $f(n) \in \Theta(g(n))$. Similarly $n^3 + \Theta(n^2)$ should be understood as some formula for the form "$n^3 + f(n)$" where $f(n) \in \Theta(n^2)$.

1. Prove that for any two positive functions g_1 and g_2 we have:

   a) $\Theta(g_1(n)) + \Theta(g_2(n)) \subseteq \Theta(g_1(n) + g_2(n))$
   b) $\Theta(g_1(n)) + \Theta(g_2(n)) \subseteq \Theta(\max(g_1(n), g_2(n)))$
   c) $\Theta(g_1(n)) \cdot \Theta(g_2(n)) \subseteq \Theta(g_1(n) \cdot g_2(n))$

2. ❆ **Disprove** the following properties:

   a) $g_1(n) = \Theta(g_1(n/2))$
   b) $g_1(n) = \Theta(g_2(n)) \iff \log_2 g_1(n) = \Theta(\log_2 g_2(n))$

2.4 Ternary Search ❆

Propose an implementation of TernarySearch and study its complexity.

Day 3

3.1 Using the "master theorem"

Let us for recall the *master theorem* from the lecture.

Given a recurrence equation such as $T(n) = aT(n/b + \mathrm{O}(1)) + f(n)$ with $a \geq 1,\ b > 1$.

- If $f(n) = \mathrm{O}(n^{(\log_b a)-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
- If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- If $f(n) = \Omega(n^{(\log_b a)+\varepsilon})$ for some $\varepsilon > 0$, **and** if $af(n/b) \leq cf(n)$ for some $c < 1$ and all large values of n, then $T(n) = \Theta(f(n))$.
- In other cases, the theorem does not apply.

Use this theorem to solve the following equations:

1. $T(n) = 3T(n/3) + \Theta(1)$
2. $U(n) = 2U(n/3) + \Theta(1)$
3. $V(n) = V(n/2) + n + 2$
4. $W(n) = W(\lfloor n/2 \rfloor) + W(\lceil n/2 \rceil) + \Theta(\log n)$
5. $X(n) = 2X(n/2) + \Theta(n \log n)$

3.2 Matrix Addition

1. Write an iterative algorithm, ADD(A, B, n), that sums two $n \times n$ matrices.
2. What is the complexity of your algorithm? (The master theorem is useless here.)
3. Argue that it is impossible to do better.

3.3 Naive Block Matrix Multiplication

Input: two $n \times n$ matrices A and B where n is a power of 2.
Output: the $n \times n$ matrix $C = A \times B$.

```
BMUL($A, B, n$)
  if $n = 1$ then
      $C[0][0] \leftarrow A[0][0] \times B[0][0]$
      return $C$
  $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \leftarrow A$        // slice $A$ in four $\frac{n}{2} \times \frac{n}{2}$ blocks
  $\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \leftarrow B$
  $C_{11} \leftarrow$ ADD(BMUL$(A_{11}, B_{11}, \frac{n}{2})$, BMUL$(A_{12}, B_{21}, \frac{n}{2})$, $\frac{n}{2}$)
  $C_{12} \leftarrow$ ADD(BMUL$(A_{11}, B_{12}, \frac{n}{2})$, BMUL$(A_{12}, B_{22}, \frac{n}{2})$, $\frac{n}{2}$)
  $C_{21} \leftarrow$ ADD(BMUL$(A_{21}, B_{11}, \frac{n}{2})$, BMUL$(A_{22}, B_{21}, \frac{n}{2})$, $\frac{n}{2}$)
  $C_{22} \leftarrow$ ADD(BMUL$(A_{21}, B_{12}, \frac{n}{2})$, BMUL$(A_{22}, B_{22}, \frac{n}{2})$, $\frac{n}{2}$)
  return $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$
```

1. Give a recursive equation for the complexity of this algorithm, in function of n. (Assume the slicing matrices and grouping sub-matrices is done using copies, so this is not constant time.)
2. Use the master theorem to find the complexity.
3. How do you think this algorithm compares to the naive multiplication (done with a triple loop)?

3.4 Strassen's Block Matrix Multiplication

Same questions for the following algorithm, where SUB performs the substraction of two matrices.

```
SMUL($A, B, n$)
  if $n = 1$ then
      $C[0][0] \leftarrow A[0][0] \times B[0][0]$
      return $C$
  $\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \leftarrow A$
  $\begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} \leftarrow B$
  $M_1 \leftarrow$ SMUL(ADD$(A_{11}, A_{22}, \frac{n}{2})$, ADD$(B_{11}, B_{22}, \frac{n}{2})$, $\frac{n}{2}$)
  $M_2 \leftarrow$ SMUL(ADD$(A_{21}, A_{22}, \frac{n}{2})$, $B_{11}$, $\frac{n}{2}$)
  $M_3 \leftarrow$ SMUL($A_{11}$, SUB$(B_{12}, B_{22}, \frac{n}{2})$, $\frac{n}{2}$)
  $M_4 \leftarrow$ SMUL($A_{22}$, SUB$(B_{21}, B_{11}, \frac{n}{2})$, $\frac{n}{2}$)
  $M_5 \leftarrow$ SMUL(ADD$(A_{11}, A_{12}, \frac{n}{2})$, $B_{22}$, $\frac{n}{2}$)
  $M_6 \leftarrow$ SMUL(SUB$(A_{21}, A_{11}, \frac{n}{2})$, ADD$(B_{11}, B_{12}, \frac{n}{2})$, $\frac{n}{2}$)
  $M_7 \leftarrow$ SMUL(SUB$(A_{12}, A_{22}, \frac{n}{2})$, ADD$(B_{21}, B_{22}, \frac{n}{2})$, $\frac{n}{2}$)
  $C_{11} \leftarrow$ SUB(ADD$(M_1, M_4, \frac{n}{2})$, ADD$(M_5, M_7, \frac{n}{2})$, $\frac{n}{2}$)
  $C_{12} \leftarrow$ ADD$(M_3, M_5, \frac{n}{2})$
  $C_{21} \leftarrow$ ADD$(M_2, M_4, \frac{n}{2})$
  $C_{22} \leftarrow$ ADD(SUB$(M_1, M_2, \frac{n}{2})$, ADD$(M_3, M_6, \frac{n}{2})$, $\frac{n}{2}$)
  return $\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$
```

Note that you do not need to understand why the algorithm is correct to compute its complexity.

3.5 Fast Exponentiation

This computes x^p for $p \in \mathbb{N}$.

```
FASTPOWER($x, p$)
  if $p = 0$ then
        return 1
  if odd($p$) then
        return $x \times$FASTPOWER($x \times x$, $\lfloor p/2 \rfloor$)
  else
        return FASTPOWER($x \times x$, $p/2$)
```

1. Assuming the multiplication used here is done in constant time, what are the worst-case and best-case complexities as a function of p? Deduce the complexity in the general case.
2. This algorithm can be applied to any type that has an associative operation (here $\times$) with a neutral element (here 1), i.e., any type that has a monoid structure. Suggest a variant of FASTPOWER for computing the pth power of a $n \times n$ matrix using SMUL, and compute its complexity as a function a p and n.
3. Can you see how to use FASTPOWER to compute the product of two integers on a computer whose multiplication operation is broken?

Day 4

Today we compress files by just changing how each character is encoded at the bit level.

4.1 Histogram and Fixed-Size Encodings

Let us assume that the file to compress is represented as an array $data[0..n-1]$ of n 8-bit characters.

1. Suggest an algorithm to compute the histogram of characters in $data$. I.e., the function $\textsc{Hist}(data, n)$ should return an array of length 256 giving the number of occurrences of each letter.
2. Give the complexity of Hist as a function of n.
3. What is the complexity of computing the number of different characters occurring in $data$ once the histogram is built?
4. How many bits are necessary to distinguish p characters? Call this value b.
5. We decide that the compressed file we will output starts with 256 bits indicating for each possible character whether it was used in the input. (These 256 bits are enough to find p, b, and the way each character is encoded.) This header is followed by the n characters, each encoded on b bits. Compute the compression ratio as a function of n and p. What is its limit when $n \to \infty$?

4.2 Prefix Code

We now consider a variable-length code: each character can be encoded using a different number of bits. Obviously we want frequent characters to use short codes, while long codes should be reserved to infrequent characters.

Also our encoding must have the *prefix property*: no code should be prefix of another code. If we do not respect this property, for instance encoding "a" with 11, and "b" with 111, then we cannot tell if 11111 encodes "ab" or "ba".

1. Consider the following encoding:

letter	a	b	c	d	e	f
code	0	101	100	111	1101	1100

   Decode 1101110011000100110 1.

2. Suggest an algorithm that inputs one such "code table" and a string of n bits, and outputs the decoded string.
3. How many encodings with the prefix property exist for p letters?
4. Assuming $p > 1$, argue that if one letter is encoded with p bits or more, there must exist a better encoding.
5. Let's look for an *optimal* encoding. Assume the file to compress has 10000 characters chosen between `abcdefg` with the following frequencies:

letter	a	b	c	d	e	f	g
freq.	15%	42%	7%	11%	13%	9%	3%

   Suggest an encoding of these letters such that the resulting size of all encoded letters is 3062.5 bytes (multiple solutions exist, but you cannot do less on this example).

4.3 Huffman

The Huffman algorithm computes an optimal encoding by representing any encoding as a tree whose leaves are the letters to encode. The code of a letter can be deduced from the path from the root to the leaf labeled by this letter: each left branch represent a 0, and each right branch is a 1. Furthermore, each node of the tree has an attribute *freq* that is the sum of the frequencies of all the letters below.

Lines 1–5 the algorithm build a forest F in which each tree corresponds to a character occurring in the file to compress.

The two trees whose roots have the smallest frequencies are then combined, and the operation is repeated lines 6–11 until there is only one tree left in the forest.

```
HUFFMAN($letters, freqs, p$)
1  for $i \leftarrow 0$ to $p-1$ do
2      $z \leftarrow$NEWLEAF($letters[i]$)
3      $z.freq \leftarrow freqs[i]$
4      INSERT($F, z$)
5  for $i \leftarrow 1$ to $p-1$ do
6      $z \leftarrow$NEWNODE()
7      $z.left \leftarrow$EXTRACTMIN($F$)
8      $z.right \leftarrow$EXTRACTMIN($F$)
9      $z.freq \leftarrow z.left.freq + z.right.freq$
10     INSERT($F, z$)
11 return EXTRACTMIN($F$)
```

1. Execute this algorithm with the frequencies from previous question.
2. Suggest a data structure for F so that ExtractMin and Insert will be efficient.
3. What is the complexity of Huffman as a function of p?
4. Write a recursive procedure that takes the tree built by Huffman and prints the list of letters (in any order) with their associated code. What is the complexity of this procedure?
5. To decode a file, you need to map each code to its character. Suggest a way to save the Huffman tree using at most $10p-1$ bits.
6. If the histogram is passed to Huffman has a sorted list, show that F can be represented more efficiently using two sorted lists (one for trees that are leaves, and the other for larger trees). What does the complexity of Huffman become?
7. Assume the frequencies of the p letters are the p first values of the Fibonacci sequence. What does the Huffman encoding look like?

Day 5

5.1 Frankenstein's QuickSort

We consider a variant of QuickSort where the pivot value chosen by the partition procedure is the median of the values.

```
Input: an array $A[\ell..r-1]$ of integers
Ouput: $A$ is sorted in place
FrankenSort($A, \ell, r$)
if $r - \ell > 1$ then
    $p \leftarrow$ FrankenPart($A, \ell, r$)
    FrankenSort($A, \ell, p$)
    FrankenSort($A, p, r$)
```

```
Input: an array $A[\ell..r-1]$ of integers
Output: an index $p$, and the array $A$ is rearranged
        so that $A[\ell..p-1] \leq A[p..r-1]$
FrankenPart($A, \ell, r$)
$x \leftarrow A[$Median$(A, \ell, r)]$
$i \leftarrow \ell - 1$; $j \leftarrow r$
repeat forever
    do $i \leftarrow i + 1$ until $A[i] \geq x$
    do $j \leftarrow j - 1$ until $A[j] \leq x$
    if $i < j$ then
        $A[i] \leftrightarrow A[j]$
    else
        return $i + (i = \ell)$
```

We assume that Median(A, ℓ, r) returns the *index* of the median of $A[\ell..r-1]$, and that it is allowed to reorder the values of $A[\ell..r-1]$. For some given ℓ and r, let $n = r - \ell$ be the length of the considered sub-array.

1. Justify that a comparison-based Median needs at least $n - 1$ comparisons regardless of the implementation.

2. Explain (in two lines) how to implement Median with $\Theta(n \log n)$ comparisons.

3. Justify that the following Median implementation effectively returns the index of the median of $A[l..r-1]$:

```
Median($A, \ell, r$)
$m \leftarrow \lfloor(\ell + r)/2\rfloor$
for $i \leftarrow \ell$ to $m$ do
        $min \leftarrow i$
        for $j \leftarrow i + 1$ to $r - 1$ do
                if $A[j] < A[min]$ then $min \leftarrow j$
        $A[i] \leftrightarrow A[min]$
return $m$
```

4. Give, as a function of n, the complexity of Median above.

5. Give, as a function of n, the complexity of FrankenPart when using this Median.

6. Give a recursive equation (of the form $T(n) = aT(n/b) + f(n)$) for the complexity of FrankenSort.

7. Solve this equation using the master theorem. How does this algorithm compare to QuickSort?

8. Let us modify the main algorithm as follows, without changing the definition of Median:

```
FrankenSort2($A, \ell, r$)
if $r - \ell > 1$ then
    $p \leftarrow$ Median($A, \ell, r$)
    FrankenSort2($A, p + 1, r$)
```

   Explain why it is still correct, i.e., why the array returned by FrankenSort2 is sorted althought the recursion is only done on the second half, and FrankenPart is not even called...

9. What is the complexity of FrankenSort2.

10. Since the recursive call to FrankenSort2 is a tail call, perform a tail call elimination to rewrite FrankenSort2 as an iterative procedure. Then inline the code of Median, and finally, maybe give FrankenSort2 its real name.

5.2 A QuickSort-based Selection

This exercise is almost the converse of the previous one: instead of using Median in QuickSort, we derive a Median implementation from QuickSort... More generally we want to *select* the rank-i value (i.e., the ith smallest value) in an array of size n. Rank 0 is the minimum, rank $n - 1$ is the maximum, and rank $\lfloor n/2 \rfloor$ is the median.

1. Suggest an algorithm to find the rank-i value in a n-sized array, and give its complexity.

2. Execute the following algorithm on a few examples, and explain why it returns the rank-i value.

```
Input: a non-empty array $A[\ell..r-1]$ of integers,
       and a rank $i < r - \ell$
Output: the rank-$i$ value
Selection($A, \ell, r, i$)
if $r - \ell = 1$ then return $A[l]$
$m \leftarrow$ RandomizedPartition($A, \ell, r$)
$k \leftarrow m - \ell$
if $i < k$
     then return Selection($A, \ell, m, i$)
     else return Selection($A, m, r, i - k$)
```

```
Input: a non-empty array $A[\ell..r-1]$ of integer
Output: an index $p$, and the array $A$ is rearranged
        so that $A[\ell..p-1] \leq A[p..r-1]$
RandomizedPartition($A, \ell, r$)
$x \leftarrow A[$Random$(\ell, r)]$
$i \leftarrow \ell - 1$; $j \leftarrow r$
repeat forever
    do $i \leftarrow i + 1$ until $A[i] \geq x$
    do $j \leftarrow j - 1$ until $A[j] \leq x$
    if $i < j$ then
        $A[i] \leftrightarrow A[j]$
    else
        return $i + (i = \ell)$
```

3. Assuming Random runs in $\Theta(1)$, compute the complexity of Selection in the worst case.

4. This algorithm has an average-case complexity of $\Theta(n)$, we will prove it during the lecture.