Sections 5.3 and 5.4

1. (a)

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 1 \\ f(3) &= 1 \\ f(4) &= 1 \\ &\vdots \end{aligned}$$

(b)

$$\begin{aligned} f(1) &= 20 \\ f(2) &= 10 \\ f(3) &= 5 \\ f(4) &= 16 \\ f(5) &= 8 \\ f(6) &= 4 \\ f(7) &= 2 \\ f(8) &= 1 \\ f(9) &= 4 \\ &\vdots \end{aligned}$$

(c)

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= 6 \\ f(4) &= 24 \\ &\vdots \end{aligned}$$

(d)

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 11 \\ f(3) &= 121 \\ f(4) &= 1331 \\ &\vdots \end{aligned}$$

(e)

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 2 \\ f(3) &= \frac{1}{2} \\ f(4) &= 4 \\ f(5) &= \frac{1}{8} \\ f(6) &= 32 \\ f(7) &= \frac{1}{256} \\ &\vdots \end{aligned}$$

3. (a) Let S be the set defined recursively by
   - $2 \in S$
   - If $x \in S$ and $y \in S$ then $x + y \in S$

   (b) Let S be the set defined recursively by
   - $5 \in S$
   - If $x \in S$ and $y \in S$, then $x + y \in S$

   (c) Let S be the set defined recursively by
   - $1 \in S$
   - If $x \in S$ then $3 \cdot x \in S$

   (d) Let S be the set defined recursively by
   - $1 \in S$
   - If $x \in S$ then $(\sqrt{x} + 1)^2 \in S$

   We could also write

   Let S be the set defined recursively by
   - $1 \in S$
   - If $x \in S$ then $x + 2\sqrt{x} + 1 \in S$

5. The set S is the set of pairs (x, y) where $x, y \in \mathbb{N}$ and x is even and y is odd, or x is odd and y is even (i.e. x and y have different parity).

7. Here, $A[1...n]$ is a table of numbers sorted in ascending order!

```
Algo BinarySearch(A[1..n], i, j, x)

if i = j then
    if A[i] = x then
        return i
    else
        return nil
    end if
else
    if x ≤ A[m] then
        index = BinarySearch(A[1..n], i, m, x)
    else
        index = BinarySearch(A[1..n], m + 1, j, x)
    end if
    return index
end if
```

We need to call BinarySearch$(A[1..n], 1, n, x)$.

We could write this algorithm more carefully. For instance, we could first check to see if $A[m] = x$ and return m immediately if that is the case. Here, the objective was to follow the general framework presented in class.

— The base case is when $i = j$, so when the sub-table is of size 1.

— We cut the table into two sub-tables of equal length.

— There is only a single recursive call.

— The merge step does nothing.

9.

```
Algo NbOnes(S[1..n])

if n = 1 then
    return S[1]
else
    m = ⌊m/2⌋
    ℓ = NbOnes(S[1..m])
    r = NbOnes(S[(m + 1)..n])
    total = ℓ + r
    return total
end if
```

— The base case is when $n = 1$, so when the sub-table is of size 1.

— We cut the table into two sub-tables of equal length.

— We make two recursive calls (one for each sub-problem).

— The merge step finds the $total = \ell + r$.

---

11. Algo LengthAlt($S[1..n]$)

if $n = 1$ **then**
  return 1
else
  $m = \lfloor \frac{n}{2} \rfloor$
  $\ell_1 =$ LengthAlt($S[1..m]$)
  $\ell_2 =$ LengthAlt($S[(m+1)..n]$)
  $\ell_3 = 0$
  if $S[m] \neq S[m+1]$ **then**
    • Pass through S from $S[m+1]$ to $S[n]$ to find the longest alternating sub-sequence which starts at $S[m+1]$ and update the value of ℓ_3.
    • Pass through S from $S[m]$ to $S[1]$ to find the length of the longest alternating sub-sequence which ends at $S[m]$ and update the value of ℓ_3.
  end if
  $\ell = \max\{\ell_1, \ell_2, \ell_3\}$
  return ℓ
end if

In the two locations in the algorithm description where it says "pass through" sections of the table to find the length of the longest alternating sub-sequence, we have to use a "for" loop. In this loop, we increment ℓ_3 by 1 each time the following (or preceding, depending on the direction of the pass) in S is different. As soon as we find two identical consecutive elements, we exit the for loop.

— The base case is when $n = 1$, so when the sub-table is of size 1. By default, a sequence of length 1 is considered to be alternating.

— We cut the table into two sub-tables of equal length.

— We make two recursive calls (one for each sub-problem).

— The merge step finds the longest alternating sequence that starts in the first half and ends in the second half of S. Then, this sequence is compared to the two sequences found in the recursive calls.

13. Here, we assume that the table $A[1..n]$ given as input has $n \geq 2$. The algorithm returns two parameters : the smallest element and the second smallest element of $A[1..n]$.

Algo Two($A[1..n]$)

if $n = 2$ **then**
  $n_1 = \min\{A[1], A[2]\}$
  $n_2 = \max\{A[1], A[2]\}$
  return (n_1, n_2)
else if $n = 3$ **then**
  $n_1 = \min\{A[1], A[2], A[3]\}$
  Let n_2 be the second smallest element of $\{A[1], A[2], A[3]\}$.
  return (n_1, n_2)
else
  $m = \lfloor \frac{n}{2} \rfloor$
  $(n_{1,left}, n_{2,left}) = \text{Two}(A[1..m])$
  $(n_{1,right}, n_{2,right}) = \text{Two}(A[(m+1)..n])$
  $n_1 = \min\{n_{1,left}, n_{2,left}, n_{1,right}, n_{2,right}\}$
  Let n_2 be the second smallest element of $\{n_{1,left}, n_{2,left}, n_{1,right}, n_{2,right}\}$
  return (n_1, n_2)
end if

Do you see why we need to return both parameters ?

— The base case is when $n \leq 3$, so when the sub-table is of size 3 or smaller. If $n = 2$, it's easy to find the smallest element and then the second smallest element. If $n = 3$, it's easy to find the smallest element. For the second smallest element, we have to compare the 3 numbers to find the second smallest. We need at most 3 comparisons. Do you see why the base case is $n \leq 3$ and not $n = 2$?

— We cut the table into two sub-tables of equal length.

— We make two recursive calls (one for each sub-problem).

— The merge step calculates the smallest element and second smallest element among the 4 candidates returned : $n_{1,left}$, $n_{2,left}$, $n_{1,right}$ et $n_{2,right}$. This is done with at most 5 comparisons.

15. Given two numbers a and b, we could first find $\gcd(a, b)$ and then return $\frac{a\,b}{\gcd(a,b)}$. In the previous exercise we found an algorithm to calculate $\gcd(a, b)$ recursively.

Let's try to calculate the lcm without using the gcd. First, how do we find the lcm without a recursive algorithm ? The positive multiples of a are $\{a, 2a, 3a, 4a, 5a, ...\}$ and the positive multiples of b are $\{b, 2b, 3b, 4b, 5b, ...\}$. So the lcm of a and b is the smallest

number that is found in the set

$$\{a, 2a, 3a, 4a, 5a, ...\} \cap \{b, 2b, 3b, 4b, 5b, ...\}.$$

On peut donc calculer le ppcm de la façon suivante.

Algo LCM(a,b)

$x = a$
$y = b$
while $x \neq y$ **do**
    if $x < y$ **then**
        $x = x + a$
    else
        $y = y + b$
    end if
end while
return x

Now let's see how to write this recursively. If it were a table of n numbers, it would be easy to divide the input in two. Here, the input is only two numbers. The simplest way is therefore to translate the "While" loop into a recursive algorithm. We obtain the following algorithm.

Algo LCMRecur(a,b,x,y)

if $x = y$ **then**
    return x
else
    if $x < y$ **then**
        $answer$ = LCMRecur(a,b,$x + a$,y)
    else
        $answer$ = LCMRecur(a,b,x,$y + b$)
    end if
    return $answer$
end if

Il s'agit d'appeler LCMRecur(a,b,a,b).

— The base case is when $x = y$. In this case, we have found the lcm of a and b.

— This algorithm passes through the sets $\{a, 2a, 3a, 4a, 5a, ...\}$ and $\{b, 2b, 3b, 4b, 5b, ...\}$ to find the smallest number in their intersection. With each recursive call, we exclude one candidate. There isn't really a merge step.

---

17. Algo Sum($A[1..n]$)

if $n = 1$ **then**
  return $A[1]$
else
  $m = \lfloor \frac{n}{2} \rfloor$
  $left = \text{Sum}(A[1..m])$
  $right = \text{Sum}(A[(m+1)..n])$
  $total = left + right$
  return $total$
end if

— The base case is when $n = 1$. Then the only element in the table gets returned.

— We cut the table into two sub-tables of equal length.

— We make two recursive calls (one for each sub-problem).

— The merge step adds the sum of elements in the left sub-table to the sum of elements in the right sub-table.