Week 10: Computation and Algorithms (Solutions)

1 Powerful integers

1. This algorithm has a complexity $\Theta(n)$. Indeed, if we consider $T(n)$ its complexity, we have $T(n) \leq 2 + T(n-1)$ which results in linear time.

2. Consider the following algorithm:

Algorithm 1 Binary to Decimal

input: B of length n
output: Decimal representation of B.
for i from 1 to n **do**
  $a \leftarrow A(2, n-i)$ // Compute 2^{n-i} using the Algorithm A from Question 1
  $d \leftarrow d + B[i] * a$
return d

*(Note: consider the start of the code ("**for**...") as line 1.)*

A loop iteration i needs $8 + (n-i)$ instructions: cost 8 for line 1 (cost 3), line 3 (cost 4) and the assignment of the variable a in line 2 (cost 1); and $(n-i)$ for the call of the algorithm A with $n-i$ in line 2. In total this makes $\sum_{i=1}^{n}(8 + (n-i)) = \sum_{i=1}^{n} 8 + \sum_{i=1}^{n} n - \sum_{i=1}^{n} i = 8n + n^2 - \frac{n(n+1)}{2} = 8n + n^2 - \frac{n^2}{2} - \frac{n}{2} = \frac{n^2}{2} + \frac{15n}{2} = \Theta(n^2)$.

The following algorithm solves the same problem but more efficiently.

Algorithm 2 Binary to Decimal

input: B of size n.
output: Decimal representation of B.
$d \leftarrow 0$
for i from 1 to n **do**
  $d \leftarrow 2d + B[i]$
return d

A loop iteration needs 7 instructions. The loop has n iterations and $T(n) = 7n$. This algorithm has a complexity $\Theta(n)$.

3. Consider the following algorithm:

Algorithm 3 Algorithm ***P***

input: $a > 0,\ n \geq 0$
output: a^n
if $n = 0$ **then**
  return 1
if $n \bmod 2 = 0$ **then**
  return $(P(a, n/2))^2$
return $a \times P(a, n-1)$

If we consider $T(n)$ as the complexity of this algorithm, then we have $T(n) \leq 6 + T(\lfloor n/2 \rfloor)$ which gives $O(\log n)$.

2 Find an integer in a matrix

1. We just have to go through the whole matrix:

Algorithm 4 Algorithm ***A***

```
input: A matrix M and an integer k
output: true If and only if k appears in the matrix.
n ← size(M)
for i from 1 to n do
    for j from 1 to n do
        if k = M[i][j] then
            return true
return false
```

The complexity is then $O(n^2)$.

2. We can for this question, make a binary search on each column. Given the binary search algorithm we saw in class, suppose that we have an algorithm **binary search** which, for a given ***sorted*** array with one dimension of size n, and an integer k, finds if this integer appears in the array in time $O(\log n)$. We then write the following algorithm:

Algorithm 5 Algorithm ***B***

```
input: A matrix M and an integer k
output: true if and only if k appears in the matrix.
n ← size(M)
for i from 1 to n do
    if binary_search(M[i], k) then
        return true
return false
```

This algorithm makes n binary searches on arrays of size n, so we have a complexity $O(n \log n)$.

3. Suppose we start with the matrix entry $M[1][n]$. If $M[1][n] = k$ then we have found our solution and the algorithm is finished. Otherwise, there are two possible cases:

   - $M[1][n] < k$. Then, as all the integers of the first line are smaller than $M[1][n]$, we already know that k cannot appear in this line. So we go down one line vertically and go to the matrix entry $M[2][n]$ and start again.
   - $M[1][n] > k$. Then, as all the integers in the last column are greater than $M[1][n]$, we already know that k cannot appear in this column. So we move horizontally one column to the left and go to the matrix entry $M[1][n-1]$ and start again.

   We use this principle every time we arrive at a new entry $M[i][j]$, and compare k to $M[i][j]$. If it is equal, we have found our solution. Otherwise, if k is bigger we can move to entry $M[i+1][j]$ and if k is smaller we move instead to entry $M[i][j-1]$. At all times, we know that all the matrix entries which are either strictly above $M[i][j]$ or strictly to its right cannot contain k. This ensures the algorithm is correct.

   What is the complexity of this algorithm? At each step, we move the 1 "place" down or to the left. Since we start from $i = 1, j = n$, we can only make $2n$ moves like that. The complexity is therefore $O(n)$.

Algorithm 6 Algorithm ***C***

input: A matrix M and an integer k
output: **true** if and only if k appears in the matrix.
$n \leftarrow \text{size}(M)$
$i \leftarrow 1$
$j \leftarrow n$
while $i \leq n$ et $j \geq 1$ **do**
  if $M[i][j] = k$ **then**
    return true
  if $M[i][j] < k$ **then**
    $i \leftarrow i + 1$
  if $M[i][j] > k$ **then**
    $j \leftarrow j - 1$
return false

3 Climbing stairs

1. Suppose that we reach the step $i + 2$. There are two possibilities for the last movement we made to reach the step $i + 2$.

   - We can make a simple step of length 1. So we come from the step $i+1$ and we have paid $cost[i+2]$ + the cost of the optimal strategy to reach the step $i + 1$. So $f[i + 2] \leq cost[i + 2] + f[i + 1]$.
   - We can make a step of length 2. We come from the step i we have paid $cost[i + 2]$ + the cost of the optimal strategy to reach the step i. So $f[i + 2] \leq cost[i + 2] + f[i]$.

   So, we have $f[i + 2] \leq cost[i + 2] + f[i]$ and $f[i + 2] \leq cost[i + 2] + f[i + 1]$ which gives

   $$f[i + 2] \leq cost[i + 2] + \min(f[i], f[i + 1])$$

   As there are no other possibilities for the last step, we have, in fact, equality and

   $$f[i + 2] = cost[i + 2] + \min(f[i], f[i + 1])$$

2. With the previous remark we can build the following algorithm:

Algorithm 7 Algorithm ***F***

input: an array $cost$ of size n
output: $f[n]$
$n \leftarrow \text{size}(cost)$
$f \leftarrow \emptyset$ (we initialize the list f of optimal strategies for each case)
$f[1] \leftarrow cost[1]$
$f[2] \leftarrow cost[2]$
for i from 3 to n **do**
  $f[i] \leftarrow cost[i] + \min(f[i - 1], f[i - 2])$
return $f[n]$

3. None are correct. For example, imagine that $n = 3l$ for any l. Suppose that we consider only a subset of all possible strategies: all that take exactly l steps of length 1 and l steps of length 2 (we have $l + 2l = 3l$ steps). How many such strategies are there? To determine a strategy, we just have to determine in which order we make the steps of length 1 or 2 so there are $\binom{2l}{l}$ possibilities (choose l locations for the 1 steps among $2l$ locations in total). So, since $\binom{2l}{l} = \binom{2n/3}{n/3}$ is very large compared to n^{1000}, no option is right.

We can see that despite the exponential number of possible solutions, dynamic programming allows us to solve the problem very quickly.

4. **Additional remark:** Some have suggested the following algorithm to solve the problem: when on a step i, choose the cheaper of the two available steps (steps $i+1$ and $i+2$). One might think that this algorithm gives the best strategy, but this is not the case. Consider the following example :

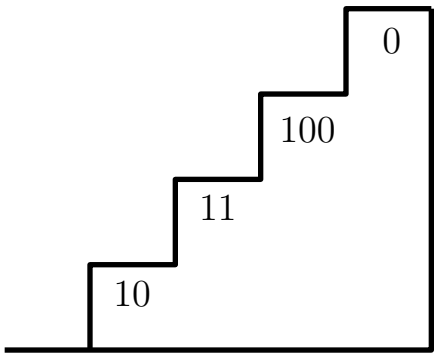

Figure 1: Counter-example

The proposed strategy goes to the 10 cost case first then 11 then 0 for a total cost of 21. While the best strategy would be to go to the 11 case first then 0 for a total cost of 11.

These types of algorithms that make local optimal choices (instead of global optimal choices) are called *greedy algorithms*, and as you can see, they don't always give us the optimal solution!

4 Birthday party

1. We could solve it with a complexity of $O(n \log n)$ by first sorting it with **merge_sort**, but we don't need a sorting algorithm for this problem. Indeed, we can just transverse the array and find the next birthday in linear time ($O(n)$), which is significantly faster than a solution with complexity of $O(n \log n)$. This makes sorting first a worse solution since it is less efficient.

   Below is a solution to this problem in linear time:

Algorithm 8 Algorithm ***G***

```
input: an array birthdays of size n
input: a number current_date
output: a number representing the next birthday
n ← size(birthdays)
earliest_date ← 13 (remember highest possible date is 12.31)
next_date ← 13
for i from 1 to n do
    if birthdays[i] < earliest_date then
        earliest_date ← birthdays[i]
    if birthdays[i] > current_date and birthdays[i] < next_date then
        next_date ← birthdays[i]
if next_date = 13 then
    return earliest_date
return next_date
```

2. We used binary search in exercise 2 (Find an integer in a matrix). We can use a variant with the same complexity of $O(\log n)$ that besides searching for the value also computes its difference with the current list element (and keeps track of both the lowest positive difference between the current date and the element as well as that element's index). Considering only positive difference ensures we are only considering cases where the birthday date is bigger than the current date, which is important for finding the *next* birthday.

   There are three possible outputs to this function: we get the current day (the next birthday is today!), we get the next birthday, or we don't get any date (instead we get an error value like -1, because there was no birthday after the current date), in which case we instead return the first list item (the next birthday is the first birthday next year).