# Lab 04

I. El-Shaarawy, & M. Touny

October 22, 2023

## Contents

## Instructor Notes

**Lemma.** $\lfloor \log n \rfloor + 1 = \lceil \log(n+1) \rceil$.

We know $n = 2^k + r$ for some $k \geq 0$ and $0 \leq r < 2^k$, By *Euclid's Theorem* and *Archimedean Property*. Then

$$k + 1 = \log 2^{k+1} \geq \log(2^k + r + 1) > \log(2^k + r) \geq \log 2^k = k$$

Thus, $\lceil \log(n+1) \rceil = \lceil \log(2^k + r + 1) \rceil = k + 1$ and $\lfloor \log(n+1) \rfloor = \lfloor \log(2^k + 1) \rfloor = k$.

**Lemma.** Given $n$, If we repeatedly apply the operation $\lfloor n/2 \rfloor$ Then we reach 1 after exactly $\lfloor \log n \rfloor + 1$.

Consider $n$ but in binary representation $(d_1 d_2 \ldots d_k)_2$, where $d_1 = 1$. Then by definition $(d_1 d_2 \ldots d_k)_2/2$ yields a quotient $(d_1 \ldots d_{k-1})$ and remainder $d_k$. Since we are taking floor, We can safely ignore $d_k$. It is easy to we reach $d_1 = 1$ after exactly $k-1$ operations. But we know $k = \lfloor \log n \rfloor$.

## Exercises

### 4.1.4

*Hints*

- Consider the fact, for a fixed element $k$, All subsets either contain $k$, or does not contain $k$.
- Given all subsets not containing $k$, What do we generate when we append $k$ to each subset?

*Solution*

Top-down

```
def generateSubsets(A[0..n-1])
  # base case, empty subset
  if A.length == 0
    return [ [ ] ]

  lastElement = A[n-1]

  # smaller instance solution
  subsetsWithNoLast = generateSubsets(A[0..n-2])

  # generate new solutions from smaller instance
  subsetsWithLast = []
```

```
    for subset in subsetsWithNoLast
      subsetsWithLast.append( subset + [lastElement] )

    # concatenate solutions
    return subsetsWithNoLast + subsetsWithLast
```

Bottom-up (Iterative Improvement)

```
def generateSubsets(A[0..n-1]):
    n = A.length
    allSubsets = [ [ ] ]

    for i in 0..n-1:
        newSubsets = []
        for subset in allSubsets:
            newSubsets.append( subset + [ A[i] ] )
        allSubsets = allSubsets + newSubsets

    return allSubsets
```

## 4.1.10

**Homework.**

## 4.2.3

**(a)** In matrix implementation $\Theta(|V|^2)$, and in adjacency list implementation $\Theta(|V| + |E|)$. Careful analysis won't be shown as it is outside the scope of the lab, especially that students lack data structures foundations.

**(b)**

*Hints*

- Consider a stack data structure
- Think in terms of recursion, Given a solved smaller instance, How do we augment it to reach a greater instance?

*Solution*

```
# a node is inserted in stack, only after calling its subgraph
# Input: node, visited nodes list, stack
# Output: NULL
def dfs(node, visited, stack):
  visited.add( node )
```

```
  for neighbor in graph[node]:
    if neighbor not in visited:
      dfs(neighbor, visited, stack)

  stack.insert(node)

# Input: directed graph in adjacency list implementation
# Output: Topological order of the graph
def topologicalSortDfs(graph G):
  visited = set() # no multiple occurences in sets
  stack = []

  # can be omitted if we assumed graph's connectivity
  # and given a unique root (tree)
  for node in G(V):
    if node not in visited:
      dfs(node)

  return stack
```

Another simpler implementation not based on DFS as a bonus answer. Preferred to students over DFS based implementation.

```
# Input: directed graph in adjacency list implementation
# Output: Topological order of the graph
def topologicalsortRecursive(graph G):
  visited = set() # multiple occurences in sets
  stack = []

  for node in G(V):
    if node not in visited:
      visited.add(node)
      topologicalSortRecursive(graph[node], visited, stack)
      stack.insert(0, node)

  return stack
```

### 4.2.8

**Homework.**

### 4.3.7

*Hints*

- For each bit string of size $n-1$, If we added 0, What do we generate?
- Combine adding 0 and 1.

*Solution*

```
# Input: Positive integer n
# Output All bit strings of length n
def generateAllBitStrings(n):
  # base case
  if n == 1:
    return ["0", "1"]
  else
    # smaller instance solution
    smallerInstanceStrings = generateAllBitsStrings(n-1)

    # generate n instance from smaller instance

    nInstanceWithZero = []
    for bitString in smallerInstanceStrings
      nInstanceWithZero.append(bitString + "0")

    nInstanceWithOne = []
    for bitString in smallerInstanceStrings
      nInstanceWithOne.append(bitString + "1")

    return nInstanceWithZero + nInstanceWithOne
```

## 4.3.10

**Homework.**

## 4.4.2

*Hints*

- Consider n separation, in case it is odd, and in case it is even.
- If odd, subtract from it only 1, to get an even number
- Since we are taking floor, We only need to care about the new even number. I.e we won't count.

*Solution*

```
def floorLog2Recursive(n):
```

```
  # Base case
  # log2(1) = 0
  if n == 1:
    return 0

  # n is even
  if n % 2 == 0:
    return 1 + floorLog2Recursive(n/2)

  # n is odd
  else
    return 0 + floorLog2Recursive( (n-1)/2 )
    # Since we consider floor, the remainder does not count
```

## 4.4.9

**Homework.**

## 4.5.12

**Homework.**

## 4.5.13

*Hints*

- Given the target $t >$ cell $c$, for some cell in the matrix. Which elements of the matrix can we exclude from the search?
- Consider the case if the cell $c$ is at the corner.
- Try to reduce the problem size by 1.

*Solution*

Recursive implementation

```
# Input: n x n Matrix, and target value t
# Output: tuple (row, column) of the element found, or -1 if not found
def searchMatrixRecursive(matrix M[0..n-1, 0..n-1], target t, row, col):
  if row >= n or col < 0:
    return -1

  # Base case
  if M[row][col] == t:
    return (row, col)
```

```
  # Call smaller instances
  else M[row][col] < t:
    return searchMatrixRecursive(M, t, row + 1, col)
  else:
    return searchMatrixRecursive(M, t, row, col - 1)

def searchMatrix(Matrix M[0..n-1, 0..n-1], target t)
  # initialize with row = 0 and column = n-1
  return searchMatrixRecursive(M, t, 0, n-1)
```

Upperbounded by $2n = \mathcal{O}(n)$ by the recurrence $T(q) = T(q-1) + 1$, where $q = n + n$, the sum of columns and rows number.

Bottom-up implementation (iterative improvement)

```
# Input: n x n matrix and target value t
# Output: tuple (row, column) of the element found, or -1 if not found
def searchMatrixBottomUp(matrix M[0..n-1, 0..n-1] , target t):
  row = 0
  col = n-1

  while row < n and col >= 0:
    if M[row][col] == t:
      return (row, col)

    if M[row][col] < t:
      row = row + 1
    else:
      col = col - 1

  return -1
```

Upperbounded by $\sum_{i=1}^{2n} 2 = 2(2n) = \mathcal{O}(n)$, the sum of columns and row numbers.

P.S. It might be more elegant to consider three-comparison as a single operation. For our students we omit this discussion.