SWERC NoteBook

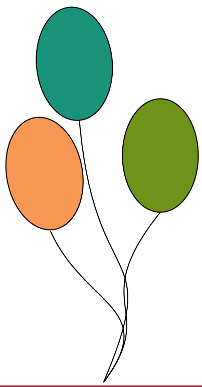
Équipe B.I.O.S

Ismaël BADAOUI, Chems-eddine BENAZIZA, Todd CAUËT

Avril 2022

Ensemble d'algorithmes et techniques de programmation

Et quelques notions de Mathématiques



Université Paris Cité
UFR de Mathématiques-Informatique

2021-2022

Document rédigé par Hugo Demaret

1 Configuration

1.1 C/C++

1.1.1 Header

```
#include <iostream>
#include <stdlib.h>
#include <stdio.h>
```

1.1.2 Vecteur

```
#include <vector>
//Declaration :
std::vector<int> name;
//Debut /Fin (element)
name.begin();
name.back();
//Debut / Fin
name.front();
name.end();
//Ajout /Deletion (fin)
name.push_back(val);
name.pop_back();
//Ajout /Suppression(index)
name.insert(index, val);
name.erase(index);
name.erase(index1,index2);
//Taille
name.size();
//Destruction
name.clear();
```

1.1.3 Deque

```
#include <deque>
//meme methodes que vector, plus :
name.pop_front();
name.push_front();
name.emplace_front();
name.emplace_back();
```

1.1.4 Stack

```
#include <stack>
std::stack<int> pile;
pile.push(val);
pile.pop();
pile.size();
pile.empty();
pile.top();
```

1.1.5 Tri

```
sort(v.begin(),v.end());
```

1.2 Python

1.2.1 Numpy

```
import numpy as np
#creation array
a = np.array([1,2,3])
b = np.array([(1.5,2,3), (4,5,6)], dtype = float)
c = np.array([[(1.5,2,3), (4,5,6)],[(3,2,1), (4,5,6)]], dtype = float)
#Fonction de bases
np.zeros((3,4)) #Create an array of zeros
np.ones((2,3,4),dtype=np.int16) #Create an array of ones
d = np.arange(10,25,5)#Create an array of evenly spaced values (step value)
np.linspace(0,2,9) #Create an array of evenlyspaced values (number of samples)
e = np.full((2,2),7)#Create a constant array
f = np.eye(2) #Create a 2X2 identity matrix
np.random.random((2,2)) #Create an array with random values
np.empty((3,2)) #Create an empty array
#Aide
np.info(np.ndarray.dtype)
#Informations sur l’array
a.shape #Array dimensions
len(a)#Length of array
b.ndim #Number of array dimensions
e.size #Number of array elements
b.dtype #Data type of array elements
b.dtype.name #Name of data type
b.astype(int) #Convert an array to a different type
#Datatype
np.int64 #Signed 64-bit integer types
np.float32 #Standard double-precision floating point
np.complex #Complex numbers represented by 128 floats
np.bool #Boolean type storing TRUE and FALSE values
np.object #Python object type
np.string_ #Fixed-length string type
np.unicode_ #Fixed-length unicode type
#Maths
#operations arithmetiques
g = a - b #Subtraction
np.subtract(a,b) #Subtraction
b + a #Addition
np.add(b,a) #Addition
a/b #Division
np.divide(a,b) #Division
a * b #Multiplication
np.multiply(a,b) #Multiplication
np.exp(b) #Exponentiation
np.sqrt(b) #Square root
np.sin(a) #Print sines of an array
np.cos(b) #Elementwise cosine
np.log(a)#Elementwise natural logarithm
e.dot(f) #Dot product
#Comparaisons
a == b #Elementwise comparison
a< 2 #Elementwise comparison
np.array_equal(a, b) #Arraywise comparison
#Copies
h = a.view()#Create a view of the array with the same data
np.copy(a) #Create a copy of the array
h = a.copy() #Create a deep copy of the array
#Tri
a.sort() #Sort an array
c.sort(axis=0) #Sort the elements of an array’s axis
#Manipulation
#Transposee
```

```
i = np.transpose(b) #Permute array dimensions
i.T #Permute array dimensions
#Changer la forme
b.ravel() #Flatten the array
#Modifier des elements
h.resize((2,6)) #Return a new arraywith shape(2,6)
np.append(h,g) #Append items to an array
np.insert(a,1,5) #Insert items in an array
np.delete(a,[1]) #Delete items from an array
#Combiner les array
np.concatenate((a,d),axis=0) #Concatenate arrays
np.vstack((a,b)) #Stack arrays vertically(row wise)
np.r_[e,f] #Stack arrays vertically(row wise)
np.hstack((e,f)) #Stack arrays horizontally(column wise)
np.column_stack((a,d)) #Create stacked column wise arrays
np.c_[a,d] #Create stacked column wise arrays
#Splitter les array
np.hsplit(a,3) #Split the array horizontally at the 3rd index
np.vsplit(c,2) #Split the array vertically at the 2nd index
#Statistics and probabilities
np.std(my_array) #ecart type
my_array.corrcoef() #coefficient de correlation
np.median(my_array) #mediane
np.mean(my_array) #moyenne empirique
```

2 Chaînes de caractères

2.1 Anagramme

$\mathcal{O}(nk\log(k))$ en moyenne

```
def anagrams(w):
    w = list(set(w)) #retire les doublons
    d = {}
    for i in range(len(w)):
        s = ''.join(sorted(w[i]))
        if s in d:
            d[s].append(i)
        else:
            d[s] = [i]
    answer = []
    for s in d:
        if len(d[s]) > 1:
            answer.append([w[i] for i in d[s]])
    return answer
```

2.2 Plus long palindrome d'une chaîne

$\mathcal{O}(n)$

```
def manacher(s):
    assert set.isdisjoint({'$', '^', '#'}, s) # Forbidden letters
    if s == "":
        return (0, 1)
    t = "^#" + "#".join(s) + "#$"
    c = 1
    d = 1
    p = [0] * len(t)
    for i in range(2, len(t) - 1):
        #                    -- reflect index i with respect to c
        mirror = 2 * c - i       # = c - (i-c)
        p[i] = max(0, min(d - i, p[mirror]))
        #                    -- grow palindrome centered in i
        while t[i + 1 + p[i]] == t[i - 1 - p[i]]:
```

```
            p[i] += 1
        #                      -- adjust center if necessary
        if i + p[i] > d:
            c = i
            d = i + p[i]
    (k, i) = max((p[i], i) for i in range(1, len(t) - 1))
    return ((i - k) // 2, (i + k) // 2) # extract solution
```

3 Séquences

3.1 Distance de Levenshtein

$\mathcal{O}(nm)$

```
def levenshtein(x,y):
    n = len(x)
    m = len(y)
    A = [[i+j for j in range(m+1)] for i in range(n+1)]
    for i in range(n):
        for j in range(m):
            A[i+1][j+1] = min(A[i][j+1] + 1, A[i+1][j] + 1, A[i][j] + int(x[i] != y[j]))
    return A[n][m]
```

3.2 Plus grand facteur commun

$\mathcal{O}(nm)$

```
def longest_common_subsequence(x,y):
    n = len(x)
    m = len(y)
    A = [[0 for j in range(m+1)] for i in range(n+1)]
    for i in range(n):
        for j in range(m):
            if x[i] == y[j]:
                A[i+1][j+1] = A[i][j] + 1
            else:
                A[i+1][j+1] = max(A[i][j+1], A[i+1][j])
    sol = []
    i, j = n, m
    while A[i][j] > 0:
        if A[i][j] == A[i-1][j]:
            i -= 1
        elif A[i][j] == A[i][j-1]:
            j -= 1
        else:
            i -= 1
            j -= 1
            sol.append(x[i])
    return ''.join(sol[::-1])
```

3.3 Plus longue sous-séquence croissante

$\mathcal{O}(|x| \times \log(|y|))$

```
from bisect import bisect_left


def longest_increasing_subsequence(x):
    n = len(x)
    p = [None] * n
    h = [None]
    b = [float('-inf')] # - infinity
    for i in range(n):
        if x[i] > b[-1]:
```

```
            p[i] = h[-1]
            h.append(i)
            b.append(x[i])
        else:
            #   -- binary search: b[k - 1] < x[i] <= b[k]
            k = bisect_left(b, x[i])
            h[k] = i
            b[k] = x[i]
            p[i] = h[k - 1]
    # extract solution in reverse order
    q = h[-1]
    s = []
    while q is not None:
        s.append(x[q])
        q = p[q]
    return s[::-1] # reverse the list to obtain the solution
```

3.4 Rabin-Karp — Recherche d'un pattern

$\mathcal{O}(len(s) + len(t))$ en temps amortit

```
#a mettre en C++ !
PRIME = 72057594037927931  # < 2^{56}
DOMAIN = 128
def roll_hash(old_val, out_digit, in_digit, last_pos):
    """roll_hash """
    val = (old_val - out_digit * last_pos + DOMAIN * PRIME) % PRIME
    val = (val * DOMAIN) % PRIME
    return (val + in_digit) % PRIME
def matches(s, t, i, j, k):
    for d in range(k):
        if s[i + d] != t[j + d]:
            return False
    return True
def rabin_karp_matching(s, t):
    hash_s = 0
    hash_t = 0
    len_s = len(s)
    len_t = len(t)
    last_pos = pow(DOMAIN, len_t - 1) % PRIME
    if len_s < len_t:              # substring too long
        return -1
    for i in range(len_t):         # preprocessing
        hash_s = (DOMAIN * hash_s + ord(s[i])) % PRIME
        hash_t = (DOMAIN * hash_t + ord(t[i])) % PRIME
    for i in range(len_s - len_t + 1):
        if hash_s == hash_t:                # hashes match
            # check character by character
            if matches(s, t, i, 0, len_t):
                return i
        if i < len_s - len_t:
            # shift window and calculate new hash on s
            hash_s = roll_hash(hash_s, ord(s[i]), ord(s[i + len_t]),
                               last_pos)
    return -1                               #no match
def rabin_karp_factor(s, t, k):
    last_pos = pow(DOMAIN, k - 1) % PRIME
    pos = {}
    assert k > 0
    if len(s) < k or len(t) < k:
        return None
    hash_t = 0
    # First calculate hash values of factors of t
    for j in range(k):
        hash_t = (DOMAIN * hash_t + ord(t[j])) % PRIME
```

```
    for j in range(len(t) - k + 1):
        # store the start position with the hash value
        if hash_t in pos:
            pos[hash_t].append(j)
        else:
            pos[hash_t] = [j]
        if j < len(t) - k:
            hash_t = roll_hash(hash_t, ord(t[j]), ord(t[j + k]), last_pos)
    hash_s = 0
    # Now check for matching factors in s
    for i in range(k):       # preprocessing
        hash_s = (DOMAIN * hash_s + ord(s[i])) % PRIME
    for i in range(len(s) - k + 1):
        if hash_s in pos:    # is this signature in s?
            for j in pos[hash_s]:
                if matches(s, t, i, j, k):
                    return (i, j)
        if i < len(s) - k:
            hash_s = roll_hash(hash_s, ord(s[i]), ord(s[i + k]), last_pos)
    return None
```

4 Parcours de graphes

4.1 DFS - Depth First Search

$\mathcal{O}(|V| + |E|)$

```
#version iterative pour eviter la recursion limit de python
def dfs_iterative(graph,start,seen):
    seen[start] = True
    to_visit = [start]
    while to_visit:
        node = to_visit.pop()
        for neighbour in graph[node]:
            if not seen[neighbour]:
                seen[neighbour] = True
                to_visit.append(neighbour)
```

4.2 BFS - Breadth First Search

$\mathcal{O}(|V| + |E|)$ (Liste adjacence) $\mathcal{O}(|V|^2)$ (Matrice adjacence)

```
from collections import deque
def bfs(graph, start=0):
    to_visit = deque()
    dist = [float('inf')] * len(graph)
    prec = [none] * len(graph)
    dist[start] = 0
    to_visit.appendleft(start)
    while to_visit: #evalue a faux si vide
        node = to_visit.pop()
        for neighbour in graph[node]:
            if dist[neighbour] == float('inf'):
                dist[neighbour] = dist[node] + 1
                prec[neighbour] = node
                to_visit.appendleft(neighbour)
    return dist, prec
```

4.3 Topological Sort

$\mathcal{O}(|V| + |E|)$

```
def topological_order(graph):
    V = range(len(graph))
```

```
    indeg = [0 for _ in V]
    for node in V:          # compute indegree
        for neighbor in graph[node]:
            indeg[neighbor] += 1
    Q = [node for node in V if indeg[node] == 0]
    order = []
    while Q:
        node = Q.pop()      # node without incoming arrows
        order.append(node)
        for neighbor in graph[node]:
            indeg[neighbor] -= 1
            if indeg[neighbor] == 0:
                Q.append(neighbor)
    return order
```

4.4 Composantes connexes

$\mathcal{O}(|N| + |E|)$

```
#connex : 4-connex
connex = [(i,j+1),(i,j-1),(i+1,j),(i-1,j)]
#connex : 8-connex
connex = [(i,j+1),(i,j-1),(i+1,j),(i-1,j),(i+1,j+1),(i-1,j-1),(i+1,j-1),(i-1,j+1)]
def dfs_grid(grid, i, j, mark, free):
    grid[i][j] = mark
    height = len(grid)
    width = len(grid[0])
    for ni, nj in connex:
        if 0 <= ni < height and 0 <= nj < width:
            if grid[ni][nj] == free:
                dfs_grid(grid, ni, nj, mark, free)
def nb_connected_components(grid, free='#'):
    nb_components = 0
    height = len(grid)
    width = len(grid[0])
    for i in range(height):
        for j in range(width):
            if grid[i][j] == free:
                nb_components += 1
                dfs_grid(grid, i, j, str(nb_components), free)
    return nb_components
```

4.5 Composantes bi-connexe

$\mathcal{O}(m\alpha(m,n))$ avec $\alpha(x,y)$ la réciproque de la fonction de Ackermann

```
def cut_nodes_edges(graph):
    n = len(graph)
    time = 0
    num = [None] * n
    low = [n] * n
    parent = [None] * n      # parent[v] = None if root else parent of v
    critical_children = [0] * n # cc[u] = #{children v | low[v] >= num[u]}
    times_seen = [-1] * n
    for start in range(n):
        if times_seen[start] == -1:           # init DFS path
            times_seen[start] = 0
            to_visit = [start]
            while to_visit:
                node = to_visit[-1]
                if times_seen[node] == 0:      # start processing
                    num[node] = time
                    time += 1
                    low[node] = float('inf')
                children = graph[node]
```

```
                if times_seen[node] == len(children): # end processing
                    to_visit.pop()
                    up = parent[node]         # propagate low to parent
                    if up is not None:
                        low[up] = min(low[up], low[node])
                        if low[node] >= num[up]:
                            critical_children[up] += 1
                else:
                    child = children[times_seen[node]] # next arrow
                    times_seen[node] += 1
                    if times_seen[child] == -1: # not visited yet
                        parent[child] = node    # link arrow
                        times_seen[child] = 0
                        to_visit.append(child) # (below) back arrow
                    elif num[child] < num[node] and parent[node] != child:
                        low[node] = min(low[node], num[child])
    cut_edges = []
    cut_nodes = []                             # extract solution
    for node in range(n):
        if parent[node] is None:               # characteristics
            if critical_children[node] >= 2:
                cut_nodes.append(node)
        else:                                  # internal nodes
            if critical_children[node] >= 1:
                cut_nodes.append(node)
            if low[node] >= num[node]:
                cut_edges.append((parent[node], node))
    return cut_nodes, cut_edges
```

4.6 Composantes fortement connexe

$\mathcal{O}(|V| + |E|)$

4.6.1 Kosaraju

```
def kosaraju_dfs(graph,nodes,order,sccp):
    times_seen = [-1] * len(graph)
    for start in nodes:
        if times_seen[start] == -1:
            to_visit = [start]
            times_seen[start] = 0
            sccp.append([start])
            while to_visit:
                node = to_visit[-1]
                children = graph[node]
                if times_seen[node] == len(children):
                    to_visit.pop()
                    order.append(node)
                else:
                    child = children[times_seen[node]]
                    times_seen[node] += 1
                    if times_seen[child] == -1:
                        times_seen[child] = 0
                        to_visit.append(child)
                        sccp[-1].append(child)
def reverse(graph):
    rev_graph = [[] for node in graph]
    for node in range(len(graph)):
        for neighbour in graph[node]:
            rev_graph[neighbour].append(node)
    return rev_graph
def kosaraju(graph):
    n = len(graph)
    order = []
```

```python
    sccp = []
    kosaraju_dfs(graph, range(n), order, [])
    kosaraju_dfs(reverse(graph),order[::-1], [], sccp)
    return sccp[::-1]
```

4.7 2-SAT

Complexité linéaire

```python
def vertex(lit):
    if lit > 0:
        return 2 * (lit - 1)
    else:
        return 2 * (-lit -1) +1
def two_sat(formula):
    n = max(abs(clause[p]) for p in (0,1) for clause in formula)
    graph = [[] for node in range(2*n)]
    for x,y in formula:
        graph[vertex(-x)].append(vertex(y))
        graph[vertex(-y)].append(vertex(x))
    sccp = kosaraju(graph)
    comp_id = [None] * (2*n)
    affectations = [None] * (2*n)
    for component in sccp:
        rep = min(component)
        for vtx in component:
            comp_id[vtx] = rep
            if affectations[vtx] == None:
                affectations[vtx] = True
                affectations[vtx ^ 1] = False
    for i in range(n):
        if comp_id[2*i] == comp_id[2*i+1]:
            return None
    return affectations[::2]
```

4.8 Cycle le plus court

Powergraph : $\mathcal{O}(n^3)$, Shortest Cycle : $\mathcal{O}(|V| \times |E|)$ Path : $\prime(V)$

```python
def path(tree, v):
    P = []
    while not P or P[-1] != v:
        P.append(v)
        v = tree[v]
    return P
def shortest_cycle(graph):
    best_cycle = float('inf')
    best_u = None
    best_v = None
    best_tree = None
    V = list(range(len(graph)))
    for root in V:
        tree, cycle_len, u, v = bfs(graph, root, best_cycle // 2)
        if cycle_len < best_cycle:
            best_cycle = cycle_len
            best_u = u
            best_v = v
            best_tree = tree
    if best_cycle == float('inf'):
        return None                # no cycle found
    Pu = path(best_tree, best_u)   # combine path to make a cycle
    Pv = path(best_tree, best_v)
    cycle = Pu[::-1] + Pv # last vertex equals first vertex
    return cycle[1:]       # remove duplicate vertex
def powergraph(graph, k):
```

```python
    V = range(len(graph))
    # create weight matrix for paths of length 1
    M = [[float('inf') for v in V] for u in V]
    for u in V:
        for v in graph[u]:
            M[u][v] = M[v][u] = 1
        M[u][u] = 0
    floyd_warshall(M)
    return [[v for v in V if M[u][v] <= k] for u in V]
```

4.9 Chemin eulérien

$\mathcal{O}(|V|+|E|)$ (Pour les deux)

4.9.1 Dirigé

```python
def eulerian_tour_directed(graph):
    P = []
    Q = [0]
    R = []
    next = [0] * len(graph)
    while Q:
        node = Q.pop()
        P.append(node)
        while next[node] < len(graph[node]):
            neighbour = graph[node][next[node]]
            next[node] += 1
            R.append(neighbour)
            node = neighbour
        while R:
            Q.append(R.pop())
    return P
```

4.9.2 Non Dirigé

```python
def eulerian_tour_undirected(graph):
    P = []
    Q = [0]
    R = []
    next = [0] * len(graph)
    seen = [set() for _ in graph]
    while Q:
        node = Q.pop()
        P.append(node)
        while next[node] < len(graph[node]):
            neighbour = graph[node][next[node]]
            next[node] += 1
            if neighbour not in seen[node]:
                seen[neighbour].add(node)
                R.append(neighbour)
                node = neighbour
        while R:
            Q.append(R.pop())
    return P
```

4.10 Chemin le plus court

4.10.1 Poids positif ou nul - Dijkstra

$\mathcal{O}(|V|^2)$

```python
from heapq import heappop, heappush
def dijkstra(graph, weight, source=0, target=None):
```

```
    """single source shortest paths by Dijkstra
       :complexity: O(|V| + |E|log|V|)"""
    n = len(graph)
    assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
    prec = [None] * n
    black = [False] * n
    dist = [float('inf')] * n
    dist[source] = 0
    heap = [(0, source)]
    while heap:
        dist_node, node = heappop(heap)    # Closest node from source
        if not black[node]:
            black[node] = True
            if node == target:
                break
            for neighbor in graph[node]:
                dist_neighbor = dist_node + weight[node][neighbor]
                if dist_neighbor < dist[neighbor]:
                    dist[neighbor] = dist_neighbor
                    prec[neighbor] = node
                    heappush(heap, (dist_neighbor, neighbor))
    return dist, prec
def dijkstra_update_heap(graph, weight, source=0, target=None):
    """single source shortest paths by Dijkstra
       :complexity: O(|V| + |E|log|V|)"""
    n = len(graph)
    assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
    prec = [None] * n
    dist = [float('inf')] * n
    dist[source] = 0
    heap = OurHeap([(dist[node], node) for node in range(n)])
    while heap:
        dist_node, node = heap.pop()    # Closest node from source
        if node == target:
            break
        for neighbor in graph[node]:
            old = dist[neighbor]
            new = dist_node + weight[node][neighbor]
            if new < old:
                dist[neighbor] = new
                prec[neighbor] = node
                heap.update((old, neighbor), (new, neighbor))
    return dist, prec
```

4.10.2 Poids arbitraire - Bellman-Ford

$\mathcal{O}(|V| \times |E|)$

```

def bellman_ford2(graph, weight, source):
    """     :complexity: O(|V|*|E|)"""
    n = len(graph)
    dist = [float('inf')] * n
    prec = [None] * n
    dist[source] = 0

    def relax():
        for nb_iterations in range(n-1):
            for node in range(n):
                for neighbor in graph[node]:
                    alt = dist[node] + weight[node][neighbor]
                    if alt < dist[neighbor]:
                        dist[neighbor] = alt
                        prec[neighbor] = node
    relax()
```

```python
    intermediate = dist[:] # is fixpoint in absence of neg cycles
    relax()
    for node in range(n):
        if dist[node] < intermediate[node]:
            dist[node] = float('-inf')
    return dist, prec, min(dist) == float('-inf')
```

4.10.3 Floyd-Warshall

$\mathcal{O}(n^3)$

```python
def floyd_warshall(weight):
    """"O(|V|^3)"""
    for k, Wk in enumerate(weight):
        for _, Wu in enumerate(weight):
            for v, Wuv in enumerate(Wu):
                alt = Wu[k] + Wk[v]
                if alt < Wuv:
                    Wu[v] = alt
    for v, Wv in enumerate(weight):
        if Wv[v] < 0:    # negative cycle found
            return True
    return False
```

4.10.4 A* - Chemin le moins coûteux

```
Input : graph G(V,E) avec node start et node end
Output : Chemin le moins couteux de start a end
ALGORITHME :
Initialisation :
    to_visit = {start}
    visited = {}
    g(start) = 0
    h_cost(start) = heuristic_function(start, end)
    f(start) = g(start) + h(start)
While to_visit not empty:
    m = noeud on top of to_visit with least f
    if m == end :
        return
    remove m from to_visit
    add m to visited
    for each n in child(m):
        if n in visited:
            continue
        cost = g(m) + distance(m,n)
        if n in to_visit and cost<g(n):
            remove n from to_visit if better path
        if n in visited and cost<g(n):
            remove n from visited
        if n not in to_visit and n not in visited:
            add n to to_visit
            g(n) = cost
            h(n) = heuristic_function(n,end)
            f(n) = g(n) + h(n)
```

4.11 Couplages

4.11.1 Bipartite Matching

$\mathcal{O}(|U| \times |E|)$

```python
def augment(u, bigraph, visit, match):
    for v in bigraph[u]:
        if not visit[v]:
```

```
            visit[v] = True
            if match[v] is None or augment(match[v], bigraph,
                visit, match):
                match[v] = u     # found an augmenting path
                return True
    return False
def max_bipartite_matching(bigraph):
    n = len(bigraph)             # same domain for U and V
    match = [None] * n
    for u in range(n):
        augment(u, bigraph, [False] * n, match)
    return match
```

4.11.2 Biparti avec poids - Kuhn-Munkres

$\mathcal{O}(|V|^3)$

```
def kuhn_munkres(G, TOLERANCE=1e-6):
    nU = len(G)
    U = range(nU)
    nV = len(G[0])
    V = range(nV)
    assert nU <= nV
    mu = [None] * nU              # empty matching
    mv = [None] * nV
    lu = [max(row) for row in G] # trivial labels
    lv = [0] * nV
    for root in U:                # build an alternate tree
        au = [False] * nU         # au, av mark nodes...
        au[root] = True           # ... covered by the tree
        Av = [None] * nV          # Av[v] successor of v in the tree
        # for every vertex u, slack[u] := (val, v) such that
        # val is the smallest slack on the constraints (*)
        # with fixed u and v being the corresponding vertex
        slack = [(lu[root] + lv[v] - G[root][v], root) for v in V]
        while True:
            (delta, u), v = min((slack[v], v) for v in V if Av[v] is None)
            assert au[u]
            if delta > TOLERANCE: # tree is full
                for u0 in U:      # improve labels
                    if au[u0]:
                        lu[u0] -= delta
                for v0 in V:
                    if Av[v0] is not None:
                        lv[v0] += delta
                    else:
                        (val, arg) = slack[v0]
                        slack[v0] = (val - delta, arg)
            assert abs(lu[u] + lv[v] - G[u][v]) <= TOLERANCE # equality
            Av[v] = u              # add (u, v) to A
            if mv[v] is None:
                break              # alternating path found
            u1 = mv[v]
            assert not au[u1]
            au[u1] = True          # add (u1, v) to A
            for v1 in V:
                if Av[v1] is None: # update margins
                    alt = (lu[u1] + lv[v1] - G[u1][v1], u1)
                    if slack[v1] > alt:
                        slack[v1] = alt
        while v is not None:       # ... alternating path found
            u = Av[v]              # along path to root
            prec = mu[u]
            mv[v] = u              # augment matching
            mu[u] = v
```

```python
            v = prec
    return (mu, sum(lu) + sum(lv))
```

4.11.3 Biparti avec préférence - Gale-Shapley

$\mathcal{O}(|V|^2)$

```python
def gale_shapley(men, women):
    n = len(men)
    assert n == len(women)
    current_suitor = [0] * n
    spouse = [None] * n
    rank = [[0] * n for j in range(n)] # build rank
    for j in range(n):
        for r in range(n):
            rank[j][women[j][r]] = r
    singles = deque(range(n)) # all men are single and get in the queue
    while singles:
        i = singles.popleft()
        j = men[i][current_suitor[i]]
        current_suitor[i] += 1
        if spouse[j] is None:
            spouse[j] = i
        elif rank[j][spouse[j]] < rank[j][i]:
            singles.append(i)
        else:
            singles.put(spouse[j]) # sorry for spouse[j]
            spouse[j] = i
    return spouse
```

4.11.4 Couverture par sommet minimum

$\mathcal{O}(|U| \times |E|)$

```python

def _alternate(u, bigraph, visitU, visitV, matchV):
    """extend alternating tree from free vertex u.
      visitU, visitV marks all vertices covered by the tree.
    """
    visitU[u] = True
    for v in bigraph[u]:
        if not visitV[v]:
            visitV[v] = True
            assert matchV[v] is not None # otherwise match is not maximum
            _alternate(matchV[v], bigraph, visitU, visitV, matchV)
def bipartite_vertex_cover(bigraph):
    """Bipartite minimum vertex cover by Koenig's theorem
    :param bigraph: adjacency list, index = vertex in U,
                                   value = neighbor list in V
    :assumption: U = V = {0, 1, 2, ..., n - 1} for n = len(bigraph)
    :returns: boolean table for U, boolean table for V
    :comment: selected vertices form a minimum vertex cover,
              i.e. every edge is adjacent to at least one selected vertex
              and number of selected vertices is minimum
    """
    V = range(len(bigraph))
    matchV = max_bipartite_matching(bigraph)
    matchU = [None for u in V]
    for v in V:                     # -- build the mapping from U to V
        if matchV[v] is not None:
            matchU[matchV[v]] = v
    visitU = [False for u in V]     # -- build max alternating forest
    visitV = [False for v in V]
    for u in V:
        if matchU[u] is None:       # -- starting with free vertices in U
```

```
            _alternate(u, bigraph, visitU, visitV, matchV)
    inverse = [not b for b in visitU]
    return (inverse, visitV)
```

4.12 Coupes & Flots

4.13 Dilworth

Complexité : comme Bipartite Matching
Le graphe doit être acyclique

```
def dilworth(graph):
    n = len(graph)
    match = max_bipartite_matching(graph) # maximum matching
    part = [None] * n                     # partition into chains
    nb_chains = 0
    for v in range(n - 1, -1, -1):       # in inverse topological order
        if part[v] is None:               # start of chain
            u = v
            while u is not None:          # follow the chain
                part[u] = nb_chains       # mark
                u = match[u]
            nb_chains += 1
    return part
```

5 Points et polygones

5.1 Points

5.1.1 Points

```
point = [x,y]
```

5.1.2 Cross-product

```
def cross_product(p1, p2):
  return p1[0] * p2[1] - p2[0] * p1[1]
```

5.1.3 Direction

```
def left_turn(a,b,c):
  return (a[0]-c[0]) * (b[1]-c[1]) - (a[1]-c[1]) * (b[0]-c[0]) > 0
  # If floats are used, instead of 0 test if in [0-10E-7,0+10E-7]
```

5.2 Enveloppe convexe

Complexité : $\mathcal{O}(n \log(n))$

```
def andrew(S):
    S.sort()
    top = []
    bot = []
    for p in S:
        while len(top) >= 2 and not left_turn(p,top[-1],top[-2]):
            top.pop()
        top.append(p)
        while len(bot) >= 2 and not left_turn(bot[-2],bot[-1],p):
            bot.pop()
        bot.append(p)
    return bot[:-1] + top[:0:-1]
```

5.3 Points les plus proches

```
def dist(p, q):
    return hypot(p[0] - q[0], p[1] - q[1]) # Euclidean dist.
def cell(point, size):
    """ returns the grid cell coordinates containing the given point.
    size is the side length of a grid cell
    beware: in other languages negative coordinates need special care
    in C++ for example int(-1.5) == -1 and not -2 as we need
    hence we need floor(x / pas) in C++ using #include <cmath>
    """
    x, y = point                    # size = grid cell side length
    return (int(x // size), int(y // size))
def improve(S, d):
    G = {}                          # maps grid cell to its point
    for p in S:                     # for every point
        a, b = cell(p, d / 2)       # determine its grid cell
        for a1 in range(a - 2, a + 3):
            for b1 in range(b - 2, b + 3):
                if (a1, b1) in G:   # compare with points
                    q = G[a1, b1]   # in surrounding cells
                    pq = dist(p, q)
                    if pq < d:      # improvement found
                        return pq, p, q
        G[a, b] = p
    return None
def closest_points(S):
    shuffle(S)
    assert len(S) >= 2
    p = S[0]                # start with distance between
    q = S[1]                # first two points
    d = dist(p, q)
    while d > 0:            # distance 0 cannot be improved
        r = improve(S, d)
        if r:               # distance improved
            d, p, q = r
        else:               # r is None: could not improve
            break
    return p, q
```

5.4 Aire d'un polygone

Complexité linéaire *Uniquement pour les polygones simples. Réduire à des composantes simples sinon.* Voir partie Mathématiques.

```
def area(p):
    A = 0
    for i in range(len(p)):
        A += p[i-1][0] * p[i][1] - p[i][0] * p[i-1][1]
    return A/2
```

5.5 Polygone simple

$\mathcal{O}(n \log(n))$

```
def is_simple(polygon):
    """complexity: O(n log n) for n=len(polygon)"""
    n = len(polygon)
    order = list(range(n))
    order.sort(key=lambda i: polygon[i])      # lexicographic order
    rank_to_y = list(set(p[1] for p in polygon))
    rank_to_y.sort()
    y_to_rank = {y: rank for rank, y in enumerate(rank_to_y)}
    S = RangeMinQuery([0] * len(rank_to_y)) # sweep structure
    last_y = None
```

```
    for i in order:
        x, y = polygon[i]
        rank = y_to_rank[y]
        right_x = max(polygon[i - 1][0], polygon[(i + 1) % n][0])
        left = x < right_x
        below_y = min(polygon[i - 1][1], polygon[(i + 1) % n][1])
        high = y > below_y
        if left:                    # y does not need to be in S yet
            if S[rank]:
                return False        # two horizontal segments intersect
            S[rank] = -1            # add y to S
        else:
            S[rank] = 0             # remove y from S
        if high:
            lo = y_to_rank[below_y] # check S between [lo + 1, rank - 1]
            if (below_y != last_y or last_y == y or
                    rank - lo >= 2 and S.range_min(lo + 1, rank)):
                return False        # horiz. & vert. segments intersect
        last_y = y                  # remember for next iteration
    return True
```

5.6 Rectangle avec des points

$\mathcal{O}(n^2)$
Combien de rectangles peut-on former dans un ensemble de points

```
def rectangles_from_points(S):
    """
    :param S: list of points, as coordinate pairs
    :returns: the number of rectangles
    """
    answ = 0
    pairs = {}
    for j, _ in enumerate(S):
        for i in range(j):     # loop over point pairs (p,q)
            px, py = S[i]
            qx, qy = S[j]
            center = (px + qx, py + qy)
            dist = (px - qx) ** 2 + (py - qy) ** 2
            signature = (center, dist)
            if signature in pairs:
                answ += len(pairs[signature])
                pairs[signature].append((i, j))
            else:
                pairs[signature] = [(i, j)]
    return answ
```

5.7 Plus grand rectangle dans un histogramme

Complexité linéaire

```
def rectangles_from_histogram(H):
    """Largest Rectangular Area in a Histogram
    :param H: histogram table
    :returns: area, left, height, right, rect. is [0, height] * [left, right)
    """
    best = (float('-inf'), 0, 0, 0)
    S = []
    H2 = H + [float('-inf')] # extra element to empty the queue
    for right, _ in enumerate(H2):
        x = H2[right]
        left = right
        while len(S) > 0 and S[-1][1] >= x:
            left, height = S.pop()
            # first element is area of candidate
```

```
            rect = (height * (right - left), left, height, right)
            if rect > best:
                best = rect
        S.append((left, x))
    return best
```

5.8 Rectangle sur une grille

$\mathcal{O}(n)$

```
def rectangles_from_grid(P, black=1):
    """Largest area rectangle in a binary matrix
    :param P: matrix
    :param black: search for rectangles filled with value black
    :returns: area, left, top, right, bottom of optimal rectangle
             consisting of all (i,j) with
             left <= j < right and top <= i <= bottom
    """
    rows = len(P)
    cols = len(P[0])
    t = [0] * cols
    best = None
    for i in range(rows):
        for j in range(cols):
            if P[i][j] == black:
                t[j] += 1
            else:
                t[j] = 0
        (area, left, height, right) = rectangles_from_histogram(t)
        alt = (area, left, i, right, i-height)
        if best is None or alt > best:
            best = alt
    return best
```

6 Ensembles

6.1 Rendu de monnaie

Problème NP-Complet.

```
def coin(x, R):
    b = [False] * (R+1)
    b[0] = True
    for xi in x:
        for s in range(xi, R+1):
            b[s] |= b[s - xi]
    return b[R]
```

6.2 Sac à dos

Problème NP-Complet.

```
def knapsack(p, v, cmax):
    n = len(p)
    Opt = [[0] * (cmax + 1) for _ in range(n+1)]
    Sel = [[False] * (cmax + 1) for _ in range(n+1)]
    #cas de base
    for cap in range(p[0], cmax +1):
        Opt[0][cap] = v[0]
        Sel[0][cap] = True
    # cas d'induction
    for i in range(1,n):
        for cap in range(cmax+1):
            if cap >= p[i] and Opt[i-1][cap - p[i]] + v[i] > Opt[i-1][cap]:
                Opt[i][cap] = Opt[i-1][cap-p[i]] + v[i]
```

```
                Sel[i][cap] = True
            else:
                Opt[i][cap] = Opt[i-1][cap]
                Sel[i][cap] = False
    cap = cmax
    sol = []
    for i in range(n-1, -1, -1):
        if Sel[i][cap]:
            sol.append(i)
            cap -= p[i]
    return (Opt[n-1][cmax], sol)
```

6.3 k-somme

$\mathcal{O}(nR)$

```
def subset_sum(x, R):
    b = [False] * (R + 1)
    b[0] = True
    for xi in x:
        for s in range(R, xi - 1, -1):
            b[s] |= b[s - xi]
    return b[R]
```

6.4 Valeurs les plus proches

```
def closest_values(L):
    assert len(L) >= 2
    L.sort()
    valmin, argmin = min((L[i] - L[i - 1], i) for i in range(1, len(L)))
    return L[argmin - 1], L[argmin]
```

6.5 Union d'intervalles

$\mathcal{O}(n \log(n))$

```
def intervals_union(S):
    E = [(low, -1) for (low, high) in S]
    E += [(high, +1) for (low, high) in S]
    nb_open = 0
    last = None
    retval = []
    for x, _dir in sorted(E):
        if _dir == -1:
            if nb_open == 0:
                last = x
            nb_open += 1
        else:
            nb_open -= 1
            if nb_open == 0:
                retval.append((last, x))
    return retval
```

6.6 MiniMax

$\mathcal{O}(b^m)$

```
minimax(state, depth, player)
  if (player = max) then
    best = [null, -infinity]
  else
    best = [null, +infinity]
  if (depth = 0 or gameover) then
    score = evaluate this state for player
```

```
    return [null, score]
  for each valid move m for player in state s do
    execute move m on s
    [move, score] = minimax(s, depth - 1, -player)
    undo move m on s
    if (player = max) then
      if score > best.score then best = [move, score]
    else
      if score < best.score then best = [move, score]
  return best
end
```

7 Calculs

7.1 PGCD

```
def pgcd(a,b):
    return a if b == 0 else pgcd(b,a%b)
```

7.2 Coefficients de Bézout

```
def bezout(a,b):
    if b == 0:
        return (1,0)
    else:
        u,v = bezout(b,a%b)
        return (v, u - (a//b) *v)
def inv(a,p):
    return bezout(a,p)[0]%p
```

7.3 Coefficients binomiaux

```
def binom(n,k):
    prod = 1
    for i in range(k):
        prod = (prod * (n-i)) // (i+1)
    return prod
def binom_modulo(n,k,p):
    prod = 1
    for i in range(k):
        prod = (prod * (n-1) * inv(i+1,p)) %p
    return prod
```

7.4 Inverse

```
def inv(a,p):
    return bezout(a,p)[0] %p
```

7.5 Nombres premiers

Crible d'Eratosthène : $\mathcal{O}(n\log(\log(n)))$
Gries Misra : $\mathcal{O}(n)$

```
def eratosthene(n):
    P = [True] * n
    answ = [2]
    for i in range(3, n, 2):
        if P[i]:
            answ.append(i)
            for j in range(i * i, n, i):
```

```
                P[j] = False
    return answ
def gries_misra(n):
    primes = []
    factor = [0] * n
    for x in range(2, n):
        if not factor[x]:     # no factor found
            factor[x] = x     # meaning x is prime
            primes.append(x)
        for p in primes:      # loop over primes found so far
            if p > factor[x] or p * x >= n:
                break
            factor[p * x] = p # p is the smallest factor of p * x
    return primes, factor
```

8 Mathématiques

8.1 Mesure

8.1.1 Distance

Définition :

- $\delta(x, y) \geq 0, \delta(x, y) = 0$ ssi $x = y$
- $\delta(x, y) = \delta(y, x)$ (Symétrique)
- Satisfait l'inégalité triangulaire : $\delta(x, z) \leq \delta(x, y) + \delta(y, z)$

Distance de Manhattan : $\delta(a, b) = |x_b - x_a| + |y_b - y_a|$

8.2 Géométrie

8.2.1 3D

Caractéristique d'Euler χ :
$\chi = s - a + f$, si $\chi = 2$ alors le polyhèdre est de rang 0 (pas de trou)
Formule usuelles :

- Sphère : Volume : $\frac{4}{3}\pi r^3$ — Surface : $4\pi r^2$
- Cylindre droit : Volume $\pi r^2 h$ — Surface : $2\pi r(r + h)$
- Cone circulaire droit : Volume $\frac{1}{3}\pi r^2 h$ — Surface : $\pi r(r + s)$
- Prisme triangulaire : Volume Al ou $\frac{1}{2}bhl$ — Surface : $bh + 2ls + lb$
- Prisme : Volume Ah — Surface : $2A + (h \times p)$
- Pyramide : Volume : $\frac{1}{3}Ah$
- Tétrahèdre : Volume : $\frac{b^3}{6\sqrt[2]{2}}$ — Surface : $\sqrt[2]{3}b^2$
- Pyramide carré : Volume : $\frac{1}{3}s^2 \times h$ — Surface : $s^2 + 2sh$
- Cuboide : Volume : $l \times w \times h$ — Surface : $2lh + 2lw + 2wh$ (Cube : $6s^2$)

8.2.2 2D

- Polygone simple : Aire : $A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i)$
- Cercle : Aire : πr^2 — Périmètre : $2 \times \pi \times r$
- Losange : Aire : $\frac{D \times d}{2}$
- Trapèze : Aire : $\frac{(B+b) \times h}{2}$
- Parralélogramme : Aire : $B \times h$
- Ellipse : Aire : $a \times b \times \pi$
  Périmètre approximation :
  $p = \pi(a + b)\left(1 + \frac{1}{4}h + \frac{1}{64}h^2 + \frac{1}{256}h^3 + ...\right)$

8.2.3 Points entiers dans un polygone

Sur le contour :
Dans le polygone :
Théorème de Pick : $P = n_i + \frac{n_b}{2} - 1$

8.2.4 Théorème de la galerie d'art

Pour garder un polygone simple à n sommets, $\lfloor \frac{n}{3} \rfloor$ gardiens suffisent.

8.3 Approximations

8.3.1 Méthode de Newton

$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$ — $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$

8.3.2 Méthode de la sécante

Cette méthode est à appliquer quand le calcul de la dérivée est couteux $\frac{f(x_k)-f(x_{k-1})}{x_k - x_{k-1}}$

8.3.3 Plus forte pente — Méthode du gradient

Cette méthode peut être assez couteuse (ZigZag)
Algorithme du gradient — On se donne un point initial $x_0 \in \mathbb{E}$ et un seuil de tolérance $\epsilon \geq 0$. L'algorithme du gradient donne une suite d'itérés $x_1, x_2 ... \in \mathbb{E}$, jusqu'à ce qu'un test d'arrêt soit satisfait. Il passe de x_k à x_{k+1} par les étapes suivantes :

- 1. *Simulation* : Calcul de $\nabla f(x_k)$
- 2. *Test d'arrêt* : Si $\|\nabla f(x_k)\| \leq \epsilon$, arrêt
- 3. *Calcul du pas* : $\alpha_k > 0$ par un règle de recherche linéaire sur f en x_k le long de la direction $-\nabla f(x_k)$
- 4. *Nouvel itéré* : $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$

Note : plus le pas est petit, plus la convergence est lente, mais précise, et plus il est grand, plus la convergence est forte, mais imprécise : **il faut donc choisir un pas adapté**

8.4 Probabilités et Statistiques

8.4.1 Lois de probabilités

Discrètes :

- Poisson : $\mathbb{P}(X = k) = e^{-\lambda} \frac{\lambda^k}{k!}, \quad \mathbb{E}[X] = \lambda, \quad \mathbb{V}[X] = \lambda$
- Binomiale : $\mathbb{P}(X = k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad \mathbb{E}[X] = np, \quad \mathbb{V}[X] = np(1-p)$
- Géométrique : $\mathbb{P}(X = k) = (1-p)^{k-1} p, \quad \mathbb{E}[X] = \frac{1}{p}, \quad \mathbb{V}[X] = \frac{1-p}{p^2}$
- Uniforme : $\mathbb{P}(X = k) = \frac{1}{n}, \quad \mathbb{E}[X] = \frac{1}{n} \sum_{k=1}^{n} x_k$

8.4.2 Techniques statistiques

Théorème de la limite centrale :
Soit $X_1, X_2, ...$ une suite de variable aléatoires réelles définies sur le même espace de probabilités, i.i.d et suivant la même loi $\mathcal{L}$. De plus, l'espérance μ et l'écart-type σ de $\mathcal{L}$ existent et soient finis avec $\sigma \neq 0$.
Soit la somme $S_n = X_1 + X_2 + ... + X_n$
Alors l'espérance de S_n est $n\mu$ et l'écart-type est $\sigma\sqrt{n}$
Quand n est assez grand, la Loi Normale $\mathcal{N}(n\mu, n\sigma^2)$ est une bonne approximation de S_n
On pose $\overline{X_n} = \frac{S_n}{n}$ et $Z_n = \frac{S_n - n\mu}{\sigma\sqrt{n}} = \frac{\overline{X_n} - \mu}{\sigma/\sqrt{n}}$

8.5 Analyse

8.5.1 Sommes

$$\sum_{k=0}^{n} a^k = \frac{1-a^{n+1}}{1-a}, a \neq 1 \tag{1}$$

$$\sum_{k=0}^{n} ka^k = \frac{a}{(1-a)^2}[1-(n+1)a^n + na^{n+1}], a \neq 1 \tag{2}$$

$$\sum_{k=0}^{n} k^2a^k = \frac{a}{(1-a)^3}[(1+a)-(n+1)^2a^n + (2n^2+2n-1)a^{n+1} - n^2a^{n+2}], a \neq 1 \tag{3}$$

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2} \tag{4}$$

$$\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \tag{5}$$

$$\sum_{k=0}^{n} k^3 = \left(\frac{n(n+1)}{2}\right)^2 \tag{6}$$

$$\sum_{k=0}^{n} k^3 = \frac{n}{30}(n+1)(2n+1)(3n^2+3n-1) \tag{7}$$

8.5.2 Dérivée et primitives usuelles

Fonction	**Domaine de dérivabilité**	**Dérivée**
$\ln(x)$	$\mathbb{R}^{+*}$	$\frac{1}{x}$
e^x	$\mathbb{R}$	e^x
$\frac{1}{x}$	$\mathbb{R}^*$	$-\frac{1}{x^2}$
$\sqrt{x}$	$\mathbb{R}^{+*}$	$\frac{1}{2\sqrt{x}}$
$x^\alpha, \alpha \in \mathbb{R}$	$\mathbb{R}^{+*}$	$\alpha x^{\alpha-1}$
$\cos(x)$	$\mathbb{R}$	$-\sin(x)$
$\sin(x)$	$\mathbb{R}$	$\cos(x)$
$\tan(x)$	$-]-\frac{\pi}{2}+k\pi;\frac{\pi}{2}+k\pi[, k \in \mathbb{Z}$	$1+\tan^2(x) = \frac{1}{\cos^2(x)}$
$\arccos(x)$	$]-1;1[$	$\frac{-1}{\sqrt{1-x^2}}$
$\arcsin(x)$	$]-1;1[$	$\frac{1}{\sqrt{1-x^2}}$
$\arctan(x)$	$\mathbb{R}$	$\frac{1}{1+x^2}$
$\cosh(x)$	$\mathbb{R}$	$\sinh(x)$
$\sinh(x)$	$\mathbb{R}$	$\cosh(x)$
$\tanh(x)$	$\mathbb{R}$	$\frac{1}{\cosh^2(x)} = 1-\tanh^2(x)$

Fonction	Intervalle d'intégration	Primitive
$(x-a)^n, n \in \mathbb{N}, a \in \mathbb{R}$	$\mathbb{R}$	$\frac{1}{n+1}(x-a)^{n+1}$
$\frac{1}{x-a}, a \in \mathbb{R}$	$]-\infty; a[\cup]a; +\infty[$	$\ln(\lvert x-a\rvert)$
$\frac{1}{(x-a)^n}, a \in \mathbb{R}, n \geq 2$	$]-\infty; a[\cup]a; +\infty[$	$-\frac{1}{(n-1)(x-a)^{n-1}}$
$\cos(ax), a \in \mathbb{R}^*$	$\mathbb{R}$	$\frac{1}{a}\sin(ax)$
$\sin(ax), a \in \mathbb{R}^*$	$\mathbb{R}$	$-\frac{1}{a}\cos(ax)$
$\tan(x)$	$-]-\frac{\pi}{2}+k\pi; \frac{\pi}{2}+k\pi[, k \in \mathbb{Z}$	$-\ln(\lvert\cos(x)\rvert)$
$\ln(x)$	$\mathbb{R}^{+*}$	$x\ln(x) - x$
$e^{ax}, a \in \mathbb{R}^*$	$\mathbb{R}$	$\frac{1}{a}e^{ax}$
$(x-a)^\alpha, a \in \mathbb{R}, \alpha \in \mathbb{R} \setminus \{-1\}$	$]a; +\infty[$	$\frac{1}{\alpha+1}(x-a)^{\alpha+1}$
$a^x, a > 0$	$\mathbb{R}$	$\frac{1}{\ln(a)}a^x$
$\frac{1}{x^2+1}$	$\mathbb{R}$	$\arctan(x)$
$\sqrt{x-a}, a \in \mathbb{R}$	$]a; +\infty[$	$\frac{3}{2}(x-a)^{\frac{3}{2}}$
$\frac{1}{\sqrt{x-a}}, a \in \mathbb{R}$	$]a; +\infty[$	$2\sqrt{x-a}$
$\frac{1}{1-x^2}$	$]-1; 1[$	$\arcsin(x)$
$\cosh(x)$	$\mathbb{R}$	$\sinh(x)$
$\sinh(x)$	$\mathbb{R}$	$\cosh(x)$
$\tanh(x)$	$\mathbb{R}$	$\ln(\cosh(x))$

Opération	Dérivée
$f+g$	$f'+g'$
$f \cdot g$	$f' \cdot g + f \cdot g'$
$\frac{f}{g}$	$\frac{f' \cdot g - f \cdot g'}{g^2}$
$g \circ f$	$f' \times g' \circ f$
$\frac{1}{u}$	$-\frac{u'}{u^2}$
u^n	$nu'u^{n-1}$
$\sqrt{u}$	$\frac{u'}{2\sqrt{u}}$
e^u	$u'e^u$
$\ln(u)$	$\frac{u'}{u}$
$\sin(u)$	$u'\cos(u)$
$\cos(u)$	$-u'\sin(u)$

8.6 Trigonométrie

Soient $a, b, x \in \mathbb{R}$:

$$\cos^2(x) + \sin^2(x) = 1 \tag{8}$$

$$\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b) \tag{9}$$

$$\sin(a+b) = \sin(a)\sin(b) + \cos(a)\cos(b) \tag{10}$$

$$\cos(2x) = 2\cos^2(x) - 1 = 1 - 2\sin^2(x) \tag{11}$$

$$\cos^2(x) = \frac{1+\cos(2x)}{2} \tag{12}$$

$$\sin(2x) = 2\sin(x)\cos(x) \tag{13}$$

$$\sin^2(x) = \frac{1-\cos(2x)}{2} \tag{14}$$

9 Techniques de programmation

9.1 Programmation dynamique

Résoudre le problème en le divisant en sous-problèmes, résoudre les sous-problèmes, stocker les résultats intermédiaires ("mémoisation")

9.2 Diviser pour régner

Diviser un problème en sous-problèmes; Résoudre les sous-problèmes; Combiner : calculer la solution grâce aux solutions des sous-problèmes.

9.3 Floyd's Hare and Tortoise

L'objectif de cette méthode est de détecter des cycles. L'idée est de parcourir la liste chaînée avec deux pointeurs : un lent (tortoise) et un deux fois plus rapide (hare). Si les deux pointeurs s'intersectent, il y a un cycle.

10 Expressions régulières

Utilité	Python & C++
Tous les caractères	.
Début	^
Fin	$
Bordure du mot	\b
non(Bordure)	\B
Opérateur "ou"	|
Un des caractères inclus	[abc]
Sauf les caractères inclus	[^abc]
Un des caractères dans l'intervalle	[a-z]
Un des caractères dans les intervalles	[a-zA-Z]
≥ 0	*
≥ 1	+
≤ 1	?
n	$\{n\}$
$\geq n$	$\{n,\}$
$\geq n$ ET $\leq m$	$\{n,m\}$