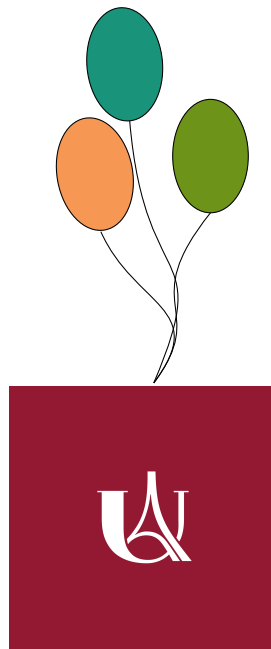# SWERC NoteBook

Équipe SaintGermainDesPrés

Mathilde BONIN, Eyal COHEN, Hugo DEMARET

Avril 2022

Ensemble d'algorithmes et techniques de programmation
Et quelques notions de Mathématiques

1 Configuration

1.1 C/C++

1.1.1 Header

```
1  #include <iostream>
2  #include <stdlib.h>
3  #include <stdio.h>
```

1.1.2 Vecteur

```
1   #include <vector>
2   //Declaration :
3   std::vector<int> name;
4   //Debut /Fin (element)
5   name.begin();
6   name.back();
7   //Debut / Fin
8   name.front();
9   name.end();
10  //Ajout /Deletion (fin)
11  name.push_back(val);
12  name.pop_back();
13  //Ajout /Suppression(index)
14  name.insert(index, val);
15  name.erase(index);
16  name.erase(index1,index2);
17  //Taille
18  name.size();
19  //Destruction
20  name.clear();
```

1.1.3 Deque

```
1  #include <deque>
2  //meme methodes que vector, plus :
3  name.pop_front();
4  name.push_front();
5  name.emplace_front();
6  name.emplace_back();
```

1.1.4 Stack

```
1  #include <stack>
2  std::stack<int> pile;
3  pile.push(val);
4  pile.pop();
5  pile.size();
6  pile.empty();
7  pile.top();
```

1.1.5 Tri

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

1.2 Python

1.2.1 Numpy

```python
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
```

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

2 Chaînes de caractères

2.1 Anagramme

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

```
1   def anagrams(w):
2       w = list(set(w)) #retire les doublons
3       d = {}
4       for i in range(len(w)):
5           s = ''.join(sorted(w[i]))
6           if s in d:
7               d[s].append(i)
8           else:
9               d[s] = [i]
10      answer = []
11      for s in d:
12          if len(d[s]) > 1:
13              answer.append([w[i] for i in d[s]])
14      return answer
```

2.2 Plus long palindrome d'une chaîne

$\mathcal{O}(n)$

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

3

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

3 Séquences

3.1 Distance de Levenshtein

$\mathcal{O}(nm)$

```
1  def levenshtein(x,y):
2      n = len(x)
3      m = len(y)
4      A = [[i+j for j in range(m+1)] for i in range(n+1)]
5      for i in range(n):
6          for j in range(m):
7              A[i+1][j+1] = min(A[i][j+1] + 1, A[i+1][j] + 1, A[i][j] + int(x[i] != y[j]))
8      return A[n][m]
```

3.2 Plus grand facteur commun

$\mathcal{O}(nm)$

```
1  def longest_common_subsequence(x,y):
2      n = len(x)
3      m = len(y)
4      A = [[0 for j in range(m+1)] for i in range(n+1)]
5      for i in range(n):
6          for j in range(m):
7              if x[i] == y[j]:
8                  A[i+1][j+1] = A[i][j] + 1
9              else:
10                 A[i+1][j+1] = max(A[i][j+1], A[i+1][j])
11     sol = []
12     i, j = n, m
13     while A[i][j] > 0:
14         if A[i][j] == A[i-1][j]:
15             i -= 1
16         elif A[i][j] == A[i][j-1]:
17             j -= 1
18         else:
19             i -= 1
20             j -= 1
21             sol.append(x[i])
22     return ''.join(sol[::-1])
```

3.3 Plus longue sous-séquence croissante

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

```
1  from bisect import bisect_left
2
3
4  def longest_increasing_subsequence(x):
5      n = len(x)
6      p = [None] * n
7      h = [None]
8      b = [float('-inf')] # - infinity
9      for i in range(n):
10         if x[i] > b[-1]:
11             p[i] = h[-1]
```

```python
            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

```python
#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):
```

```
46          # store the start position with the hash value
47          if hash_t in pos:
48              pos[hash_t].append(j)
49          else:
50              pos[hash_t] = [j]
51          if j < len(t) - k:
52              hash_t = roll_hash(hash_t, ord(t[j]), ord(t[j + k]), last_pos)
53      hash_s = 0
54      # Now check for matching factors in s
55      for i in range(k):       # preprocessing
56          hash_s = (DOMAIN * hash_s + ord(s[i])) % PRIME
57      for i in range(len(s) - k + 1):
58          if hash_s in pos:     # is this signature in s?
59              for j in pos[hash_s]:
60                  if matches(s, t, i, j, k):
61                      return (i, j)
62          if i < len(s) - k:
63              hash_s = roll_hash(hash_s, ord(s[i]), ord(s[i + k]), last_pos)
64      return None
```

4   Parcours de graphes

4.1   DFS - Depth First Search

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

```
1  #version iterative pour eviter la recursion limit de python
2  def dfs_iterative(graph,start,seen):
3      seen[start] = True
4      to_visit = [start]
5      while to_visit:
6          node = to_visit.pop()
7          for neighbour in graph[node]:
8              if not seen[neighbour]:
9                  seen[neighbour] = True
10                 to_visit.append(neighbour)
```

4.2   BFS - Breadth First Search

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

```
1  from collections import deque
2  def bfs(graph, start=0):
3      to_visit = deque()
4      dist = [float('inf')] * len(graph)
5      prec = [none] * len(graph)
6      dist[start] = 0
7      to_visit.appendleft(start)
8      while to_visit: #evalue a faux si vide
9          node = to_visit.pop()
10         for neighbour in graph[node]:
11             if dist[neighbour] == float('inf'):
12                 dist[neighbour] = dist[node] + 1
13                 prec[neighbour] = node
14                 to_visit.appendleft(neighbour)
15     return dist, prec
```

4.3   Topological Sort

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

```
1  def topological_order(graph):
2      V = range(len(graph))
3      indeg = [0 for _ in V]
```

```
4      for node in V:          # compute indegree
5          for neighbor in graph[node]:
6              indeg[neighbor] += 1
7      Q = [node for node in V if indeg[node] == 0]
8      order = []
9      while Q:
10         node = Q.pop()         # node without incoming arrows
11         order.append(node)
12         for neighbor in graph[node]:
13             indeg[neighbor] -= 1
14             if indeg[neighbor] == 0:
15                 Q.append(neighbor)
16     return order
```

4.4   Composantes connexes

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

```
1   #connex : 4-connex
2   connex = [(i,j+1),(i,j-1),(i+1,j),(i-1,j)]
3   #connex : 8-connex
4   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)]
5   def dfs_grid(grid, i, j, mark, free):
6       grid[i][j] = mark
7       height = len(grid)
8       width = len(grid[0])
9       for ni, nj in connex:
10          if 0 <= ni < height and 0 <= nj < width:
11              if grid[ni][nj] == free:
12                  dfs_grid(grid, ni, nj, mark, free)
13  def nb_connected_components(grid, free='#'):
14      nb_components = 0
15      height = len(grid)
16      width = len(grid[0])
17      for i in range(height):
18          for j in range(width):
19              if grid[i][j] == free:
20                  nb_components += 1
21                  dfs_grid(grid, i, j, str(nb_components), free)
22      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

```
1   def cut_nodes_edges(graph):
2       n = len(graph)
3       time = 0
4       num = [None] * n
5       low = [n] * n
6       parent = [None] * n      # parent[v] = None if root else parent of v
7       critical_children = [0] * n # cc[u] = #{children v | low[v] >= num[u]}
8       times_seen = [-1] * n
9       for start in range(n):
10          if times_seen[start] == -1:           # init DFS path
11              times_seen[start] = 0
12              to_visit = [start]
13              while to_visit:
14                  node = to_visit[-1]
15                  if times_seen[node] == 0:      # start processing
16                      num[node] = time
17                      time += 1
18                      low[node] = float('inf')
19                  children = graph[node]
20                  if times_seen[node] == len(children): # end processing
```

```
21                          to_visit.pop()
22                          up = parent[node]        # propagate low to parent
23                          if up is not None:
24                              low[up] = min(low[up], low[node])
25                              if low[node] >= num[up]:
26                                  critical_children[up] += 1
27                      else:
28                          child = children[times_seen[node]] # next arrow
29                          times_seen[node] += 1
30                          if times_seen[child] == -1: # not visited yet
31                              parent[child] = node    # link arrow
32                              times_seen[child] = 0
33                              to_visit.append(child) # (below) back arrow
34                          elif num[child] < num[node] and parent[node] != child:
35                              low[node] = min(low[node], num[child])
36      cut_edges = []
37      cut_nodes = []                              # extract solution
38      for node in range(n):
39          if parent[node] is None:                # characteristics
40              if critical_children[node] >= 2:
41                  cut_nodes.append(node)
42          else:                                   # internal nodes
43              if critical_children[node] >= 1:
44                  cut_nodes.append(node)
45              if low[node] >= num[node]:
46                  cut_edges.append((parent[node], node))
47      return cut_nodes, cut_edges
```

4.6 Composantes fortement connexe

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

4.6.1 Kosaraju

```
1   def kosaraju_dfs(graph,nodes,order,sccp):
2       times_seen = [-1] * len(graph)
3       for start in nodes:
4           if times_seen[start] == -1:
5               to_visit = [start]
6               times_seen[start] = 0
7               sccp.append([start])
8               while to_visit:
9                   node = to_visit[-1]
10                  children = graph[node]
11                  if times_seen[node] == len(children):
12                      to_visit.pop()
13                      order.append(node)
14                  else:
15                      child = children[times_seen[node]]
16                      times_seen[node] += 1
17                      if times_seen[child] == -1:
18                          times_seen[child] = 0
19                          to_visit.append(child)
20                          sccp[-1].append(child)
21  def reverse(graph):
22      rev_graph = [[] for node in graph]
23      for node in range(len(graph)):
24          for neighbour in graph[node]:
25              rev_graph[neighbour].append(node)
26      return rev_graph
27  def kosaraju(graph):
28      n = len(graph)
29      order = []
30      sccp = []
```

```
31    kosaraju_dfs(graph, range(n), order, [])
32    kosaraju_dfs(reverse(graph),order[::-1], [], sccp)
33    return sccp[::-1]
```

4.7   2-SAT

Complexité linéaire

```
1  def vertex(lit):
2      if lit > 0:
3          return 2 * (lit - 1)
4      else:
5          return 2 * (-lit -1) +1
6  def two_sat(formula):
7      n = max(abs(clause[p]) for p in (0,1) for clause in formula)
8      graph = [[] for node in range(2*n)]
9      for x,y in formula:
10         graph[vertex(-x)].append(vertex(y))
11         graph[vertex(-y)].append(vertex(x))
12     sccp = kosaraju(graph)
13     comp_id = [None] * (2*n)
14     affectations = [None] * (2*n)
15     for component in sccp:
16         rep = min(component)
17         for vtx in component:
18             comp_id[vtx] = rep
19             if affectations[vtx] == None:
20                 affectations[vtx] = True
21                 affectations[vtx ^ 1] = False
22     for i in range(n):
23         if comp_id[2*i] == comp_id[2*i+1]:
24             return None
25     return affectations[::2]
```

4.8   Cycle le plus court

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

```
1  def path(tree, v):
2      P = []
3      while not P or P[-1] != v:
4          P.append(v)
5          v = tree[v]
6      return P
7  def shortest_cycle(graph):
8      best_cycle = float('inf')
9      best_u = None
10     best_v = None
11     best_tree = None
12     V = list(range(len(graph)))
13     for root in V:
14         tree, cycle_len, u, v = bfs(graph, root, best_cycle // 2)
15         if cycle_len < best_cycle:
16             best_cycle = cycle_len
17             best_u = u
18             best_v = v
19             best_tree = tree
20     if best_cycle == float('inf'):
21         return None                # no cycle found
22     Pu = path(best_tree, best_u)   # combine path to make a cycle
23     Pv = path(best_tree, best_v)
24     cycle = Pu[::-1] + Pv # last vertex equals first vertex
25     return cycle[1:]      # remove duplicate vertex
26 def powergraph(graph, k):
27     V = range(len(graph))
```

```
28      # create weight matrix for paths of length 1
29      M = [[float('inf') for v in V] for u in V]
30      for u in V:
31          for v in graph[u]:
32              M[u][v] = M[v][u] = 1
33          M[u][u] = 0
34      floyd_warshall(M)
35      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

```
1  def eulerian_tour_directed(graph):
2      P = []
3      Q = [0]
4      R = []
5      next = [0] * len(graph)
6      while Q:
7          node = Q.pop()
8          P.append(node)
9          while next[node] < len(graph[node]):
10             neighbour = graph[node][next[node]]
11             next[node] += 1
12             R.append(neighbour)
13             node = neighbour
14         while R:
15             Q.append(R.pop())
16     return P
```

4.9.2 Non Dirigé

```
1  def eulerian_tour_undirected(graph):
2      P = []
3      Q = [0]
4      R = []
5      next = [0] * len(graph)
6      seen = [set() for _ in graph]
7      while Q:
8          node = Q.pop()
9          P.append(node)
10         while next[node] < len(graph[node]):
11             neighbour = graph[node][next[node]]
12             next[node] += 1
13             if neighbour not in seen[node]:
14                 seen[neighbour].add(node)
15                 R.append(neighbour)
16                 node = neighbour
17         while R:
18             Q.append(R.pop())
19     return P
```

4.10 Chemin le plus court

4.10.1 Poids positif ou nul - Dijkstra

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

```
1  from heapq import heappop, heappush
2  def dijkstra(graph, weight, source=0, target=None):
3      """single source shortest paths by Dijkstra
```

```
 4          :complexity: O(|V| + |E|log|V|)"""
 5      n = len(graph)
 6      assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
 7      prec = [None] * n
 8      black = [False] * n
 9      dist = [float('inf')] * n
10      dist[source] = 0
11      heap = [(0, source)]
12      while heap:
13          dist_node, node = heappop(heap)    # Closest node from source
14          if not black[node]:
15              black[node] = True
16              if node == target:
17                  break
18              for neighbor in graph[node]:
19                  dist_neighbor = dist_node + weight[node][neighbor]
20                  if dist_neighbor < dist[neighbor]:
21                      dist[neighbor] = dist_neighbor
22                      prec[neighbor] = node
23                      heappush(heap, (dist_neighbor, neighbor))
24      return dist, prec
25  def dijkstra_update_heap(graph, weight, source=0, target=None):
26      """single source shortest paths by Dijkstra
27         :complexity: O(|V| + |E|log|V|)"""
28      n = len(graph)
29      assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
30      prec = [None] * n
31      dist = [float('inf')] * n
32      dist[source] = 0
33      heap = OurHeap([(dist[node], node) for node in range(n)])
34      while heap:
35          dist_node, node = heap.pop()    # Closest node from source
36          if node == target:
37              break
38          for neighbor in graph[node]:
39              old = dist[neighbor]
40              new = dist_node + weight[node][neighbor]
41              if new < old:
42                  dist[neighbor] = new
43                  prec[neighbor] = node
44                  heap.update((old, neighbor), (new, neighbor))
45      return dist, prec
```

4.10.2 Poids arbitraire - Bellman-Ford

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

```
 1
 2  def bellman_ford2(graph, weight, source):
 3      """     :complexity: O(|V|*|E|)"""
 4      n = len(graph)
 5      dist = [float('inf')] * n
 6      prec = [None] * n
 7      dist[source] = 0
 8
 9      def relax():
10          for nb_iterations in range(n-1):
11              for node in range(n):
12                  for neighbor in graph[node]:
13                      alt = dist[node] + weight[node][neighbor]
14                      if alt < dist[neighbor]:
15                          dist[neighbor] = alt
16                          prec[neighbor] = node
17      relax()
18      intermediate = dist[:] # is fixpoint in absence of neg cycles
```

```
19        relax()
20        for node in range(n):
21            if dist[node] < intermediate[node]:
22                dist[node] = float('-inf')
23        return dist, prec, min(dist) == float('-inf')
```

4.10.3 Floyd-Warshall

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

```
1  def floyd_warshall(weight):
2      """O(|V|^3)"""
3      for k, Wk in enumerate(weight):
4          for _, Wu in enumerate(weight):
5              for v, Wuv in enumerate(Wu):
6                  alt = Wu[k] + Wk[v]
7                  if alt < Wuv:
8                      Wu[v] = alt
9      for v, Wv in enumerate(weight):
10         if Wv[v] < 0:     # negative cycle found
11             return True
12     return False
```

4.10.4 A* - Chemin le moins coûteux

```
1  Input : graph G(V,E) avec node start et node end
2  Output : Chemin le moins couteux de start a end
3  ALGORITHME :
4  Initialisation :
5      to_visit = {start}
6      visited = {}
7      g(start) = 0
8      h_cost(start) = heuristic_function(start, end)
9      f(start) = g(start) + h(start)
10 While to_visit not empty:
11     m = noeud on top of to_visit with least f
12     if m == end :
13         return
14     remove m from to_visit
15     add m to visited
16     for each n in child(m):
17         if n in visited:
18             continue
19         cost = g(m) + distance(m,n)
20         if n in to_visit and cost<g(n):
21             remove n from to_visit if better path
22         if n in visited and cost<g(n):
23             remove n from visited
24         if n not in to_visit and n not in visited:
25             add n to to_visit
26             g(n) = cost
27             h(n) = heuristic_function(n,end)
28             f(n) = g(n) + h(n)
```

4.11 Couplages

4.11.1 Bipartite Matching

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

```
1  def augment(u, bigraph, visit, match):
2      for v in bigraph[u]:
3          if not visit[v]:
4              visit[v] = True
```

```python
 5          if match[v] is None or augment(match[v], bigraph,                    visit, match):
 6              match[v] = u      # found an augmenting path
 7              return True
 8      return False
 9  def max_bipartite_matching(bigraph):
10      n = len(bigraph)            # same domain for U and V
11      match = [None] * n
12      for u in range(n):
13          augment(u, bigraph, [False] * n, match)
14      return match
```

4.11.2 Biparti avec poids - Kuhn-Munkres

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

```python
 1  def kuhn_munkres(G, TOLERANCE=1e-6):
 2      nU = len(G)
 3      U = range(nU)
 4      nV = len(G[0])
 5      V = range(nV)
 6      assert nU <= nV
 7      mu = [None] * nU             # empty matching
 8      mv = [None] * nV
 9      lu = [max(row) for row in G] # trivial labels
10      lv = [0] * nV
11      for root in U:               # build an alternate tree
12          au = [False] * nU        # au, av mark nodes...
13          au[root] = True          # ... covered by the tree
14          Av = [None] * nV         # Av[v] successor of v in the tree
15          # for every vertex u, slack[u] := (val, v) such that
16          # val is the smallest slack on the constraints (*)
17          # with fixed u and v being the corresponding vertex
18          slack = [(lu[root] + lv[v] - G[root][v], root) for v in V]
19          while True:
20              (delta, u), v = min((slack[v], v) for v in V if Av[v] is None)
21              assert au[u]
22              if delta > TOLERANCE: # tree is full
23                  for u0 in U:      # improve labels
24                      if au[u0]:
25                          lu[u0] -= delta
26                  for v0 in V:
27                      if Av[v0] is not None:
28                          lv[v0] += delta
29                      else:
30                          (val, arg) = slack[v0]
31                          slack[v0] = (val - delta, arg)
32              assert abs(lu[u] + lv[v] - G[u][v]) <= TOLERANCE # equality
33              Av[v] = u             # add (u, v) to A
34              if mv[v] is None:
35                  break            # alternating path found
36              u1 = mv[v]
37              assert not au[u1]
38              au[u1] = True        # add (u1, v) to A
39              for v1 in V:
40                  if Av[v1] is None: # update margins
41                      alt = (lu[u1] + lv[v1] - G[u1][v1], u1)
42                      if slack[v1] > alt:
43                          slack[v1] = alt
44          while v is not None:     # ... alternating path found
45              u = Av[v]            # along path to root
46              prec = mu[u]
47              mv[v] = u            # augment matching
48              mu[u] = v
49              v = prec
50      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

```python
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

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

5.1.2 Cross-product

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

5.1.3 Direction

```python
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 : $\mathcal{O}(n \log(n))$

```python
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

```python
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.

```python
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

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

```python
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:
```

```
12        x, y = polygon[i]
13        rank = y_to_rank[y]
14        right_x = max(polygon[i - 1][0], polygon[(i + 1) % n][0])
15        left = x < right_x
16        below_y = min(polygon[i - 1][1], polygon[(i + 1) % n][1])
17        high = y > below_y
18        if left:                  # y does not need to be in S yet
19            if S[rank]:
20                return False      # two horizontal segments intersect
21            S[rank] = -1          # add y to S
22        else:
23            S[rank] = 0           # remove y from S
24        if high:
25            lo = y_to_rank[below_y] # check S between [lo + 1, rank - 1]
26            if (below_y != last_y or last_y == y or
27                    rank - lo >= 2 and S.range_min(lo + 1, rank)):
28                return False      # horiz. & vert. segments intersect
29        last_y = y                # remember for next iteration
30    return True
```

5.6   Rectangle avec des points

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

```
1  def rectangles_from_points(S):
2      """
3      :param S: list of points, as coordinate pairs
4      :returns: the number of rectangles
5      """
6      answ = 0
7      pairs = {}
8      for j, _ in enumerate(S):
9          for i in range(j):     # loop over point pairs (p,q)
10             px, py = S[i]
11             qx, qy = S[j]
12             center = (px + qx, py + qy)
13             dist = (px - qx) ** 2 + (py - qy) ** 2
14             signature = (center, dist)
15             if signature in pairs:
16                 answ += len(pairs[signature])
17                 pairs[signature].append((i, j))
18             else:
19                 pairs[signature] = [(i, j)]
20     return answ
```

5.7   Plus grand rectangle dans un histogramme

Complexité linéaire

```
1  def rectangles_from_histogram(H):
2      """Largest Rectangular Area in a Histogram
3      :param H: histogram table
4      :returns: area, left, height, right, rect. is [0, height] * [left, right)
5      """
6      best = (float('-inf'), 0, 0, 0)
7      S = []
8      H2 = H + [float('-inf')] # extra element to empty the queue
9      for right, _ in enumerate(H2):
10         x = H2[right]
11         left = right
12         while len(S) > 0 and S[-1][1] >= x:
13             left, height = S.pop()
14             # first element is area of candidate
15             rect = (height * (right - left), left, height, right)
```

```
16              if rect > best:
17                  best = rect
18          S.append((left, x))
19      return best
```

5.8 Rectangle sur une grille

$\mathcal{O}(n)$

```
1  def rectangles_from_grid(P, black=1):
2      """Largest area rectangle in a binary matrix
3      :param P: matrix
4      :param black: search for rectangles filled with value black
5      :returns: area, left, top, right, bottom of optimal rectangle
6              consisting of all (i,j) with
7              left <= j < right and top <= i <= bottom
8      """
9      rows = len(P)
10     cols = len(P[0])
11     t = [0] * cols
12     best = None
13     for i in range(rows):
14         for j in range(cols):
15             if P[i][j] == black:
16                 t[j] += 1
17             else:
18                 t[j] = 0
19         (area, left, height, right) = rectangles_from_histogram(t)
20         alt = (area, left, i, right, i-height)
21         if best is None or alt > best:
22             best = alt
23     return best
```

6 Ensembles

6.1 Rendu de monnaie

Problème NP-Complet.

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

6.2 Sac à dos

Problème NP-Complet.

```
1  def knapsack(p, v, cmax):
2      n = len(p)
3      Opt = [[0] * (cmax + 1) for _ in range(n+1)]
4      Sel = [[False] * (cmax + 1) for _ in range(n+1)]
5      #cas de base
6      for cap in range(p[0], cmax +1):
7          Opt[0][cap] = v[0]
8          Sel[0][cap] = True
9      # cas d'induction
10     for i in range(1,n):
11         for cap in range(cmax+1):
12             if cap >= p[i] and Opt[i-1][cap - p[i]] + v[i] > Opt[i-1][cap]:
13                 Opt[i][cap] = Opt[i-1][cap-p[i]] + v[i]
14                 Sel[i][cap] = True
```

```
15          else:
16              Opt[i][cap] = Opt[i-1][cap]
17              Sel[i][cap] = False
18      cap = cmax
19      sol = []
20      for i in range(n-1, -1, -1):
21          if Sel[i][cap]:
22              sol.append(i)
23              cap -= p[i]
24      return (Opt[n-1][cmax], sol)
```

6.3   k-somme

$\mathcal{O}(nR)$

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

6.4   Valeurs les plus proches

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

6.5   Union d'intervalles

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

```
1 def intervals_union(S):
2     E = [(low, -1) for (low, high) in S]
3     E += [(high, +1) for (low, high) in S]
4     nb_open = 0
5     last = None
6     retval = []
7     for x, _dir in sorted(E):
8         if _dir == -1:
9             if nb_open == 0:
10                last = x
11            nb_open += 1
12        else:
13            nb_open -= 1
14            if nb_open == 0:
15                retval.append((last, x))
16    return retval
```

7   Calculs

7.1   PGCD

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

7.2 Coefficients de Bézout

```python
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

```python
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

```python
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 : $\mathcal{O}(n)$

```python
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* : $x_{k+1} = x_k - \alpha_k \nabla f(x_k)$

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

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

Tableaux des dérivées et primitives et quelques formules en prime

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

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

Fonction	Intervalle d'intégration	Primitive
$(x-a)^n, n \in \mathbb{N}, a \in \mathbb{R}$	$\mathbb{R}$	$\dfrac{1}{n+1}(x-a)^{n+1}$
$\dfrac{1}{x-a}, a \in \mathbb{R}$	$]-\infty;a[$ OU $]a;+\infty[$	$\ln(|x-a|)$
$\dfrac{1}{(x-a)^n}, a \in \mathbb{R}, n \geq 2$	$]-\infty;a[$ OU $]a;+\infty[$	$-\dfrac{1}{(n-1)(x-a)^{n-1}}$
$\cos(ax), a \in \mathbb{R}\backslash\{0\}$	$\mathbb{R}$	$\dfrac{1}{a}\sin(ax)$
$\sin(ax), a \in \mathbb{R}\backslash\{0\}$	$\mathbb{R}$	$-\dfrac{1}{a}\cos(ax)$
$\tan(x)$	$\left]k\pi-\dfrac{\pi}{2}; k\pi+\dfrac{\pi}{2}\right[, k \in \mathbb{Z}$	$-\ln(|\cos(x)|)$
$\ln(x)$	$\mathbb{R}^{+,*}$	$x\ln(x)-x$
$e^{ax}, a \in \mathbb{R}\backslash\{0\}$	$\mathbb{R}$	$\dfrac{1}{a}e^{ax}$
$(x-a)^\alpha, a \in \mathbb{R}, \alpha \in \mathbb{R}\backslash\{-1\}$	$]a;+\infty[$	$\dfrac{1}{\alpha+1}(x-a)^{\alpha+1}$
$a^x, a > 0$	$\mathbb{R}$	$\dfrac{1}{\ln(a)}a^x$
$\dfrac{1}{x^2+1}$	$\mathbb{R}$	$\arctan(x)$
$\sqrt{x-a}, a \in \mathbb{R}$	$]a;+\infty[$	$\dfrac{2}{3}(x-a)^{3/2}$
$\dfrac{1}{\sqrt{x-a}}, a \in \mathbb{R}$	$]a;+\infty[$	$2\sqrt{x-a}$
$\dfrac{1}{\sqrt{1-x^2}}$	$]-1;1[$	$\arcsin(x)$

Quelques formules de trigonométrie vraiment utiles. a, b et x sont des réels (quelconques) :

$$\cos^2(x) + \sin^2(x) = 1, \quad \cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b), \quad \sin(a+b) = \sin(a)\cos(b) + \cos(a)\sin(b),$$

$$\cos(2x) = 2\cos^2(x) - 1 = 1 - 2\sin^2(x), \quad \cos^2(x) = \frac{1+\cos(2x)}{2},$$

$$\sin(2x) = 2\sin(x)\cos(x), \quad \sin^2(x) = \frac{1-\cos(2x)}{2}.$$

1

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 $\leq m$	$\{n, m\}$