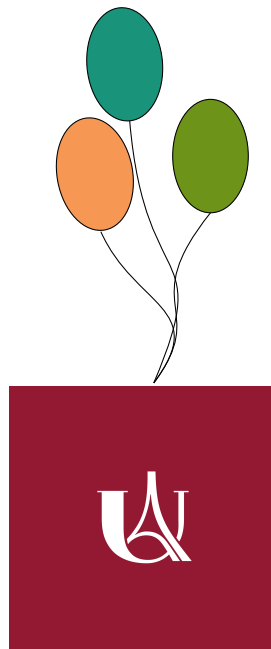# SWERC NoteBook

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

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

1.1.2 Vecteur

```cpp
#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

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

1.1.4 Stack

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

2 Chaînes de caractères

3 Séquences

3.1 Anagramme

```python
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
```

3.2 Distance de Levenshtein

```python
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]
```

3.3 Plus grand facteur commun

```python
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])
```

3.4 Rabin-Karp — Recherche d'un pattern

```python
1  #a mettre en C++ !
2  PRIME = 72057594037927931  # < 2^{56}
3  DOMAIN = 128
4  def roll_hash(old_val, out_digit, in_digit, last_pos):
5      """roll_hash """
6      val = (old_val - out_digit * last_pos + DOMAIN * PRIME) % PRIME
7      val = (val * DOMAIN) % PRIME
8      return (val + in_digit) % PRIME
```

```
 9   def matches(s, t, i, j, k):
10       for d in range(k):
11           if s[i + d] != t[j + d]:
12               return False
13       return True
14   def rabin_karp_matching(s, t):
15       hash_s = 0
16       hash_t = 0
17       len_s = len(s)
18       len_t = len(t)
19       last_pos = pow(DOMAIN, len_t - 1) % PRIME
20       if len_s < len_t:          # substring too long
21           return -1
22       for i in range(len_t):       # preprocessing
23           hash_s = (DOMAIN * hash_s + ord(s[i])) % PRIME
24           hash_t = (DOMAIN * hash_t + ord(t[i])) % PRIME
25       for i in range(len_s - len_t + 1):
26           if hash_s == hash_t:           # hashes match
27               # check character by character
28               if matches(s, t, i, 0, len_t):
29                   return i
30           if i < len_s - len_t:
31               # shift window and calculate new hash on s
32               hash_s = roll_hash(hash_s, ord(s[i]), ord(s[i + len_t]),
33                             last_pos)
34       return -1                        #no match
35   def rabin_karp_factor(s, t, k):
36       last_pos = pow(DOMAIN, k - 1) % PRIME
37       pos = {}
38       assert k > 0
39       if len(s) < k or len(t) < k:
40           return None
41       hash_t = 0
42       # First calculate hash values of factors of t
43       for j in range(k):
44           hash_t = (DOMAIN * hash_t + ord(t[j])) % PRIME
45       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

```
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

```
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

```
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

```
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

```
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

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

```
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 Postier Chinois

4.9 Chemin eulérien

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

```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
```

```
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

```
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

```
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
```

5   Points et polygones

5.1   Points

5.1.1   Points

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

5.1.2   Cross-product

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

5.1.3   Direction

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

5.2   Enveloppe convexe

Complexité : $\mathcal{O}(n \log(n))$

```
1  def andrew(S):
2      S.sort()
3      top = []
4      bot = []
5      for p in S:
6          while len(top) >= 2 and not left_turn(p,top[-1],top[-2]):
7              top.pop()
8          top.append(p)
9          while len(bot) >= 2 and not left_turn(bot[-2],bot[-1],p):
10             bot.pop()
11         bot.append(p)
12     return bot[:-1] + top[:0:-1]
```

5.3   Aire d'un polygone

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

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

5.4 Polygone simple

```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:
        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
```

5.5 Paire de points les plus proches

6 Ensembles

6.1 Rendu de monnaie
Problème NP-Complet.

```python
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.

```python
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]:
```

```
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

6.4   Points les plus proches

```
1  def dist(p, q):
2      return hypot(p[0] - q[0], p[1] - q[1]) # Euclidean dist.
3  def cell(point, size):
4      """ returns the grid cell coordinates containing the given point.
5      size is the side length of a grid cell
6      beware: in other languages negative coordinates need special care
7      in C++ for example int(-1.5) == -1 and not -2 as we need
8      hence we need floor(x / pas) in C++ using #include <cmath>
9      """
10     x, y = point                      # size = grid cell side length
11     return (int(x // size), int(y // size))
12 def improve(S, d):
13     G = {}                            # maps grid cell to its point
14     for p in S:                       # for every point
15         a, b = cell(p, d / 2)         # determine its grid cell
16         for a1 in range(a - 2, a + 3):
17             for b1 in range(b - 2, b + 3):
18                 if (a1, b1) in G:     # compare with points
19                     q = G[a1, b1]     # in surrounding cells
20                     pq = dist(p, q)
21                     if pq < d:        # improvement found
22                         return pq, p, q
23         G[a, b] = p
24     return None
25 def closest_points(S):
26     shuffle(S)
27     assert len(S) >= 2
28     p = S[0]                # start with distance between
29     q = S[1]                # first two points
30     d = dist(p, q)
31     while d > 0:            # distance 0 cannot be improved
32         r = improve(S, d)
33         if r:              # distance improved
34             d, p, q = r
35         else:              # r is None: could not improve
36             break
37     return p, q
```

6.5   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]
```

7 Calculs

7.1 PGCD

```python
def pgcd(a,b):
    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

```python
def eratosthene(n):
    """O(n loglog 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):
    """Prime numbers by the sieve of Gries-Misra
    Computes both the list of all prime numbers less than n,
    and a table mapping every integer 2 < x < n to its smallest prime factor
    :param n: positive integer
    :returns: list of prime numbers, and list of prime factors
    :complexity: O(n)
    """
    primes = []
    factor = [0] * n
    for x in range(2, n):
        if not factor[x]:    # no factor found
```

```
23              factor[x] = x      # meaning x is prime
24              primes.append(x)
25          for p in primes:       # loop over primes found so far
26              if p > factor[x] or p * x >= n:
27                  break
28              factor[p * x] = p # p is the smallest factor of p * x
29      return primes, factor
```

8 Mathématiques

8.1 Mesure

Distance définition :

8.1.1 Distance de Manhattan

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

8.2 Géométrie

8.2.1 3D

- 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 : $B \times h$

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

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

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 $Z_n = \frac{S_n - n\mu}{\sigma\sqrt{n}} = \frac{\overline{X_n} - \mu}{\sigma/\sqrt{n}}$

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.