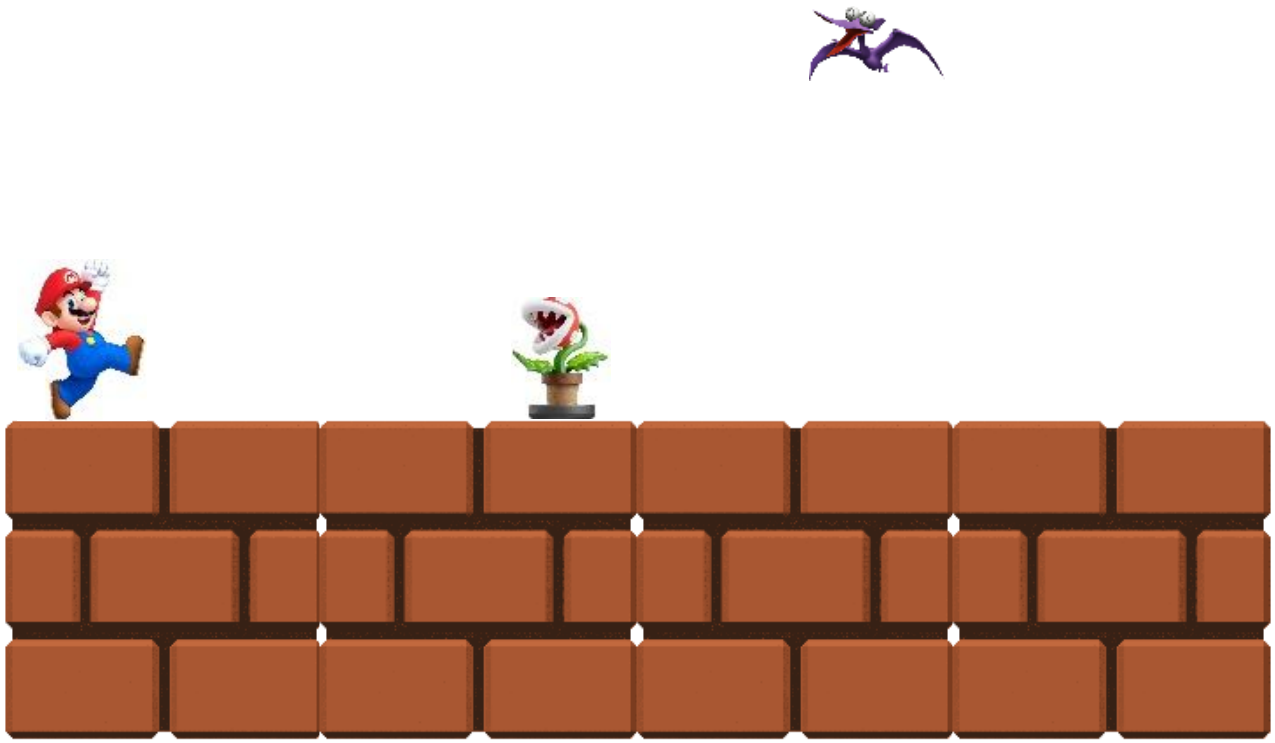
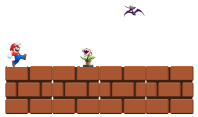
EPFC



Le parcours de Super Mario

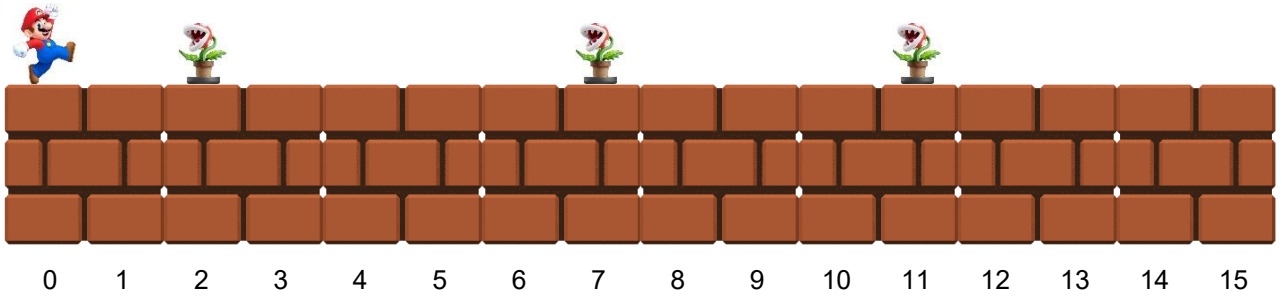


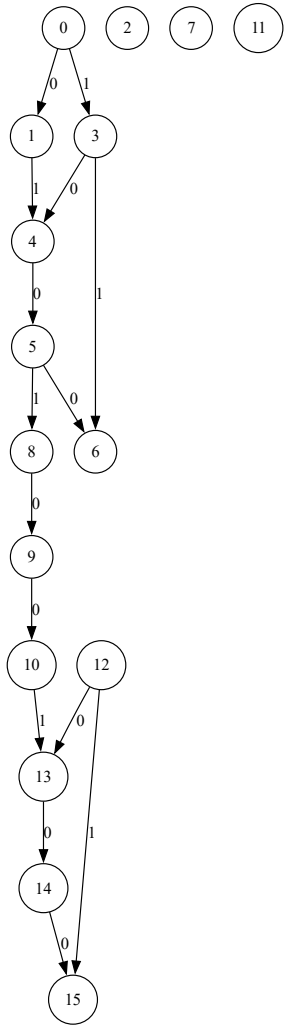
Le problème :

Combien de sauts, au minimum, devra effectuer SuperMario pour arriver vivant à l'autre bout de son univers sachant qu'il ne peut rencontrer de fleur carnivore en marchant ni de ptérodactyle en sautant

Le graphe des mouvements possibles

LG = 3





Idée de base

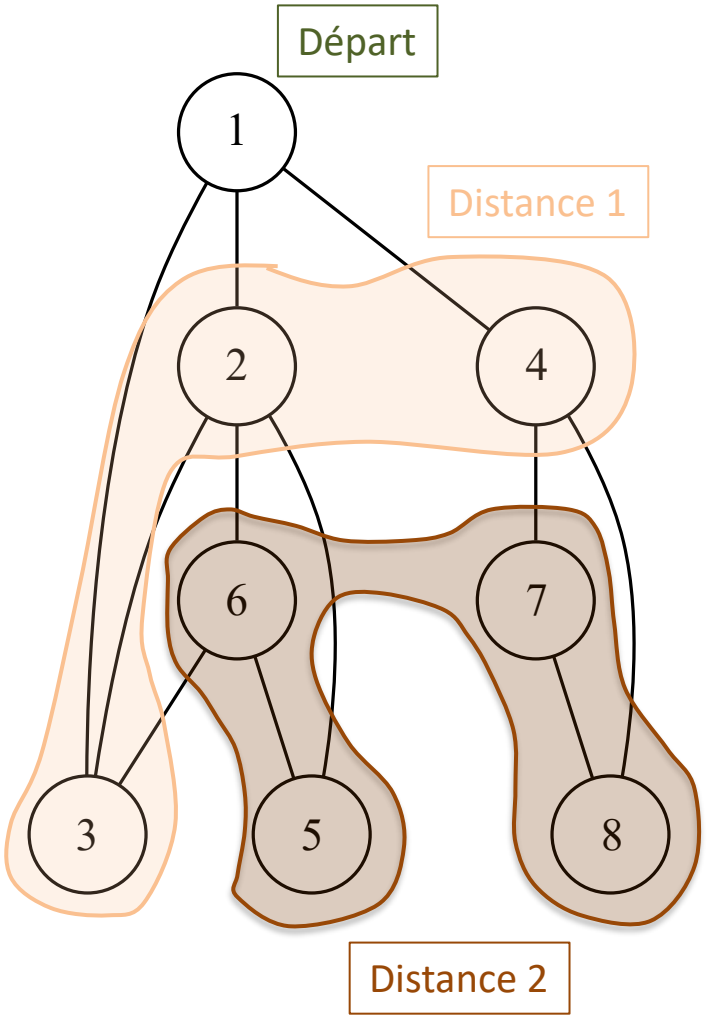
- Les arcs du graphe indiquent le coût (la « longueur ») du déplacement
- On cherche le plus court chemin!
- En général, algorithme de Dijkstra (qui utilise une file de priorité):

$$T(n) \in O(a + n \times \log n)$$

n: nb de nœuds, a: nb d'arcs

Vu au cours: Parcours en largeur

Si tous les arcs avaient le même coût, alors parcours en largeur
(utilise une simple file)



Rappel: Parcours en largeur

```
private static void parcours_largeur(int depart) {
  visite(depart);
  Queue<Integer> queue = new LinkedList<>();
  queue.add(depart);
  while(!queue.isEmpty()) {
    int n = queue.remove();   // Enlève au début
    for(int v : voisins(n))
      if(!deja_visite(v)) {
        visite(v);
        queue.add(v);         // Ajoute à la fin
      }
  }
}
```

$$T(n) \in O(n + a)$$

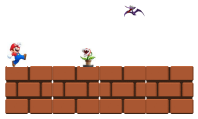
Parcours en largeur 0-1

- Dans notre cas, les arcs du graphe coûtent soit 0 (pour un simple pas), soit 1 (pour un saut)
- L'algorithme de Dijkstra peut être optimisé

  (Algorithms for Competitive Programming: 0-1 BFS)
- Il faut assurer, dans un parcours en largeur, que les nœuds atteints à coût 1 soient toujours après ceux à coût 0 dans la file

Parcours en largeur 0-1

- Utilise une *Deque* (double ended queue)
- Les voisins de distance 0 sont placés en début
- Les voisins de distance 1 sont placés en fin



EPFC

Utilisation de la Deque

```
Deque<Integer> dq = new LinkedList<>();
//...

while(!dq.isEmpty()) {
  int i = dq.removeFirst();                  // Enlève au début
  //...
  for(int v : voisins(i))
    if(v == i + 1 && /* ... */) {            // Un simple pas...
      //...
      dq.addFirst(v);                        // Ajoute au début
    }
    else if(v == i + LG && /* ... */) {      // Un saut...
      //...
      dq.addLast(v);                         // Ajoute à la fin
    }
}
```

Initialisation du nombre de sauts

LG étant > 1, aucun coût ne pourra atteindre DIM.
C'est un bon initialisateur

```
// Le nombre de sauts nécessaires
int[] nbSauts = new int[DIM];
Arrays.fill(nbSauts, DIM);    // Valeur max
nbSauts[0] = 0;
```

Coût: le nombre de sauts

- Si un nœud est atteint par un pas, son coût est le coût du nœud précédent
- S'il est atteint par un saut, son coût est 1 de plus que celui du précédent
- On prend le minimum

Coût: le nombre de sauts

```
int i = dq.removeFirst();
if (i == DIM - 1)
  return nbSauts[i];

//...
for(int v : voisins(i))
  if(v == i + 1 && nbSauts[v] > nbSauts[i]) {
    nbSauts[v] = nbSauts[i];        // Un pas ne coute rien
    dq.addFirst(v);
  }
  else if(v == i + LG &&  nbSauts[v] > nbSauts[i] + 1) {
    nbSauts[v] = nbSauts[i] + 1;   // Un saut coute 1
    dq.addLast(v);
  }
```

Les voisins…

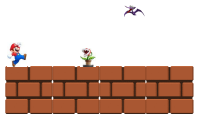
```
final Set<Integer> POS_FLEURS;
//...

private List<Integer> voisins(int i) {
  List<Integer> result = new ArrayList<>();

  if (i + 1 < DIM && !POS_FLEURS.contains(i + 1))
    result.add(i + 1);

  int j = i + LG;
  if (j < DIM && !pteroPresentIn(i, j)
              && !POS_FLEURS.contains(j))
    result.add(j);

  return result;
}
```



Les voisins…

```
final Set<Integer>  POS_PTEROS;
//...
private boolean pteroPresentIn(int d, int f) {
// Il y a un ptéro sur [d, f] ?
  for(int i = d; i <= f; ++i)
    if(POS_PTEROS.contains(i))
      return true;
  return false;
}
```

Félicitations à tous les participants