

# ASD2 : Labo 2

## Ex 1 - FindMaxDistanceFrom

```
template<typename Graph>
int FindMaxDistanceFrom(const Graph& G,int v) {    BFS<Graph> bfs(G);
    int maximum = 0;
    //parcours le graph
    bfs.visit(v, [&](int v){
        int counter = 0;
        //tant que l'on est pas arrivé à v.
        while(bfs.parentOf(v)!= v){
            counter++;
            v = bfs.parentOf(v);
            if(maximum < counter){
                maximum = counter;
            }
        }
        counter = 0;
    });
    return maximum;
}
```

On parcourt tous les  
prédécesseurs de  
chaque sommet visités

➡ Avec un parcours en largeur, nous savons que le dernier sommet visité est le plus éloigné

# ASD2 : Labo 2

## Ex 1 - FindMaxDistanceFrom

```
template<typename Graph>
int FindMaxDistanceFrom(const Graph& G,int v) {
    BFS<Graph> bfs(G);
    int lastVisitedSommet;

    bfs.visit(v,&)(int w) { lastVisitedSommet = w; });

    int distance = 0;
    while(lastVisitedSommet != v) {
        ++distance;
        lastVisitedSommet = bfs.parentOf(lastVisitedSommet);
    }
    return distance;
}
```

On garde l'index du dernier sommet visité

On parcourt les prédécesseurs du dernier sommet visité, jusqu'à v, afin de calculer la distance

# ASD2 : Labo 2




## Ex 2 - Encapsulation

```
class Pixel {  
public :  
  
    unsigned x;  
    unsigned y;  
  
    Pixel(const int x, const int y) {  
        this->x = x;  
        this->y = y;  
    }  
  
    Pixel orientationNord() const {  
        return Pixel(x, y - 1);  
    }  
};
```

```
...  
if(pixelActuel.y > 0) {  
    Pixel temp = pixelActuel.orientationNord();  
    image.get_pixel(temp.x, temp.y, temp.rouge, temp.vert, temp.bleu);  
    if(pixelActuel == temp) {  
        neighbors.push_back(idx(temp.x, temp.y));  
    }  
}  
...
```

# ASD2 : Labo 2

## Ex 3 - Choix de la structure de données

Container STL	Structure utilisée	Complexité insertion	Complexité <i>name()</i>	Complexité <i>index()</i>
vector<string>	tableau	$O(1)$  réallocations	$O(1)$	$O(N)$
map<string,int>	arbre binaire	$O(\log(N))$	$O(N)$	$O(\log(N))$
 unordered_map<string,int>	table de hachage	$O(1)$  réallocations	$O(N)$	$O(1)$ idéalement
list<string>	liste	$O(1)$	$O(N)$	$O(N)$

Aucune structure de données n'est optimale pour *name()* ET pour *index()*

# ASD2 : Labo 2

## Ex 3 - Choix de la structure de données

- **Time-memory tradeoff**

La mémoire peut être utilisée pour réduire le temps d'exécution

- On va utiliser 2 structures de données

- Une favorisant la méthode *name()*

- *vector*

- Une favorisant la méthode *index()*

- *map* ou *unordered\_map*

# ASD2 : Labo 2

## Ex 3 - Choix de la structure de données

- Idéalement *vector* ne devrait être rempli que lorsque l'on connaîtra sa taille finale, sinon de nombreux redimensionnements auront lieu.
- Idem pour *unordered\_map*
- *map* peut être rempli en  $O(\log(N))$

# ASD2 : Labo 2

Ex 3 - Choix de la structure de données

## Solution

### 1. `map<string,int>`

- On va la remplir en lisant le fichier
- Attribuant les index dans l'ordre

### 2. `vector<string>`

- Une fois tous les symboles dans la *map*, on connaît leur nombre, on peut créer un *vecteur* à la bonne taille
- On parcourt **une seule fois** la *map* en  $O(N)$

# ASD2 : Labo 2

## Ex 3 - Choix de la structure de données

### Résultat

- Lecture du fichier  $O(N)$
  - Insertion dans la *map*  $O(N \log(N))$
  - Insertion dans le *vecteur*  $O(N)$
- }  $O(N \log(N))$
- `name()`  $O(1)$
  - `index()`  $O(\log(N))$