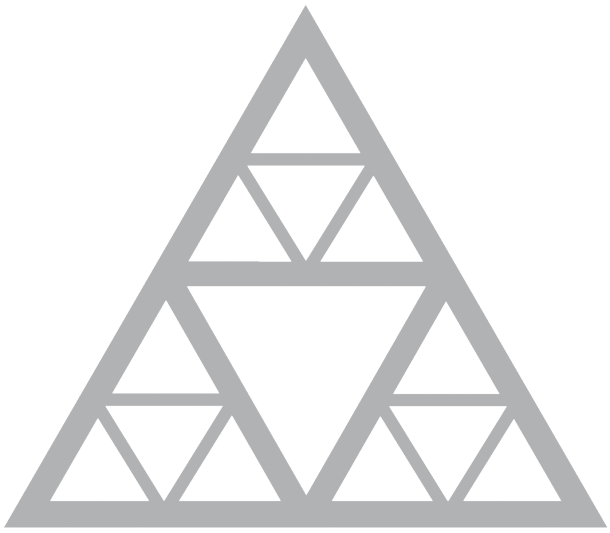
Algorithmique et Structures de Données



École des Ponts

ParisTech

Cours de l'École des Ponts ParisTech - 2022/2023

Pascal Monasse
IMAGINE - École des Ponts ParisTech
pascal.monasse@enpc.fr

Version électronique : http://imagine.enpc.fr/~monasse/Algo/algo.pdf

"Beware of bugs in the above code ; I have only proved it correct, not tried it." (Donald Knuth)

Table des matières

Chapitre 1

Structure de données

Les ordinateurs sont là pour nous affranchir des tâches fastidieuses ou répétitives. Mais les répétitions se retrouvent même dans nos programmes. Ainsi, nous sommes souvent amenés à rassembler une collection de données dans un tableau. Nous savons déjà utiliser les tableaux C++, mais notre usage en est primitif : ce ne sont pas des classes, donc nous devons tout faire à la main. En fait, il n'y a pas une structure canonique satisfaisant tout le monde. Suivant l'usage qu'on en fait, une manière de gérer la collection peut être préférable à une autre. Ce problème se rencontre si fréquemment que le C++ propose par défaut des classes. Pour la plus grande généricité possible, ces classes sont template *et font partie de la STL, que nous avons déjà rapidement mentionnée dans le cours de programmation. Ce chapitre n'a pas pour but de documenter les détails de ces classes, mais plutôt d'expliquer les forces et faiblesses de chacune et la façon dont elles ont pu être implémentées.*

—

1.1 Rappels sur les tableaux

Nous connaissons maintenant bien les tableaux, mais rappelons quelques caractéristiques importantes :

- les tableaux sont de taille fixe et ne peuvent être étendus après leur création ;
- les tableaux statiques doivent avoir une taille connue par le compilateur ;
- les tableaux dynamiques peuvent avoir une taille qui n'est connue qu'à l'exécution, mais il faut alors s'occuper soi-même de la mémoire (penser à désallouer) ;
- aucune facilité n'est offerte pour copier un tableau.

Ainsi, il faut presque tout faire soi-même ! Par exemple, pour insérer un élément il faut décaler tous les suivants tout en s'assurant que la place est suffisante. On ne peut pas non plus demander sa taille à un tableau... Même si c'est suffisant pour faire tout ce qu'on veut, ces limitations ne facilitent pas la tâche du programmeur et alourdissent le code.

L'idée dans ce cas est bien sûr de mettre toutes ces fonctionnalités dans une classe, dont les méthodes se chargent des traitements bas niveau (allocation mémoire, copie, etc.). De fait, c'est ce qui est proposé dans la Standard Template Library (STL), qui est partie intégrante du C++. En réalité, il y a même plusieurs classes pour cela ! Mais pourquoi donc ? En fait ces classes s'utilisent presque de la même façon (interface proche) mais leur implémentation est totalement différente. Suivant l'usage qu'on veut en faire, l'une sera plus efficace que les autres.

1.2 La complexité

Avant de regarder de plus près les différentes classes, il convient de préciser ce qu'on entend par complexité.

1.2.1 Mais qu'est-ce donc que la complexité ?

C'est le nombre d'opérations qu'il faut effectuer pour résoudre un problème et la taille de la mémoire nécessaire. On parle de complexité en temps dans le premier cas et de complexité en espace dans le second. On est souvent amené à trouver un équilibre entre les deux, quand on ne peut pas réduire l'une et l'autre simultanément. Dans ce cas le principe général est le suivant : "la mémoire est bon marché, le temps est précieux". Plus concrètement, ajouter de la mémoire à une machine est facile, par contre la puissance de calcul est limitée (et ajouter des processeurs n'est pas une option viable en général, car il faut réécrire le programme pour l'exploiter et beaucoup de problèmes ne sont pas ou difficilement parallélisables). Prenons un exemple simple : tracer l'histogramme d'une image. Voici une façon de faire :

```
int histo[256];
for(int i=0; i < 256; i++)
    histo[i] = 0;
for(int i=0; i < image.height(); i++)
    for(int j=0; j < image.width(); j++)
        ++ histo[image(j,i)];
for(int i=0; i < 256; i++)
    drawRect(i, 0, 1, histo[i]);
```

On voit qu'en plus de parcourir deux fois le tableau histo (ce qui ne dépend pas de la taille de l'image), on ne lit qu'une fois chaque pixel de l'image. Sa complexité est donc de l'ordre de N, le nombre de pixels. Considérons maintenant une variante :

```
for(int c=0; c < 256; c++) {
    int h=0;
    for(int i=0; i < image.height(); i++)
        for(int j=0; j < image.width(); j++)
            if(image(j,i) == c)
                ++h;
    drawRect(i, 0, 1, h);
}
```

La différence est qu'on a économisé en mémoire un tableau de 256 entiers, histo. Du coup, on lit chaque pixel de l'image 256 fois. On a une complexité en temps de $256N$, soit un facteur 256 par rapport à l'implémentation précédente. On préférera sans ambiguïté la première solution.

Nous considérerons ici surtout la complexité en temps, car la mêmoire n'est pas un facteur déterminant dans les problèmes que nous traiterons.

1.2.2 Comment la mesure-t-on ?

1.2.3 La notation O

La question de la complexité se pose quand on a des problèmes de grande dimension. Sur des jeux de données petits, tous les algorithmes se valent ou presque. On s'intéresse donc au cas où le nombre d'éléments devient grand, c'est-à-dire quand N tend vers l'infini. On utilise dans ce cas la notation commode $O(f(N))$ où f est une fonction croissante. Cela signifie que le nombre d'opérations est borné par $C.f(N)$ où C est une constante *qui ne dépend pas de* N. Bien sûr la constante C a un rôle, mais on s'intéresse avant tout à la fonction f puisque c'est le facteur déterminant. Les algorithmes d'histogramme de la section ci-dessus sont tous deux en $O(N)$ bien que leur constante soit très différente. Voici quelques exemples de complexité qu'on rencontre souvent par ordre de complexité croissante :

— $O(1)$ ($f(N) = 1$ pour tout N) signifie un algorithme qui s'effectue en temps constant indépendamment de la taille des données.

— $O(\log N)$ est un algorithme rapide, puisqu'il n'a pas besoin de lire toutes les données. Il n'est possible que si celles-ci ont déjà une certaine structure imposée. Exemple : recherche par dichotomie dans un tableau trié.

— $O(N)$ (f est la fonction identité) désigne un algorithme linéaire, proportionnel au nombre d'éléments.

— $O(N \log N)$ se rencontre fréquemment, et ces algorithmes sont considérés comme rapides. La fonction $\log$ augmentant très lentement, c'est légèrement plus que linéaire [1]. Ainsi la FFT [2], que nous verrons en TP, un algorithme célèbre de calcul de la transformée de Fourier, peut tout à fait s'appliquer à une image de 10 Mpixels, c'est-à-dire $N = 10^7$.

— $O(N^2)$ est un algorithme quadratique. Il peut rester acceptable pour des N pas trop grands.

— $O(2^N)$ est une complexité exponentielle ($f(N) = e^{N \log 2}$). C'est un algorithme qui n'est pratique que pour des problèmes de petite dimension.

1.2.4 P et NP

Nous n'entrerons pas dans des considérations d'informatique théorique, mais beaucoup ont entendu parler de la question $P = NP$? et du million de dollars offert par l'institut Clay pour sa résolution. Pour faire bref, la classe P désigne des problèmes pour lesquels il existe un algorithme polynomial (f est un polynôme) et NP est un problème pour lequel on ne connaît pas de tel algorithme (ce qui ne veut pas dire qu'il n'en existe pas !), typiquement un problème de combinatoire pour lequel on ne sait pas faire mieux que tester toutes les combinaisons. Les problèmes NP ne sont pas solubles exactement en temps raisonnable [3] lorsque N est grand. Mais cela ne veut pas dire qu'on ne peut pas trouver une solution approchée en temps polynômial.

1. Il s'agit souvent du logarithme en base 2, mais celui-ci ne différant du logarithme népérien que par un facteur constant, nous n'avons pas besoin de préciser dans la notation O
2. Fast Fourier Transform, notez le *Fast*
3. ou plus exactement s'ils le sont, on ne sait pas comment !

1.2.5 Pourquoi est-ce important ?

Un algorithme exponentiel est inutilisable pour de grandes données. Mais parmi les algorithmes polynômiaux, on privilégiera celui de complexité minimale, car il nous fera gagner du temps [4]. De nombreuses architectures ajustent leur fréquence en fonction de l'utilisation du CPU, donc consomment moins d'énergie quand le processeur a peu à faire, comme par exemple la technologie SpeedStep chez Intel. Cela a des conséquences sur la batterie de votre ordinateur portable, sur la quantité d'énergie consommée, sur le dégagement de chaleur du processeur, sur le bruit produit suivant que le ventilateur se met en marche... Utilisons donc des algorithmes efficaces, que le processeur puisse se remettre rapidement en mode basse fréquence, cela participe à la lutte contre le réchauffement climatique !

Attention, il faut quand même se garder de la tentation de vouloir optimiser trop tôt [5]. Il faut d'abord identifier la partie coûteuse en temps, et elle n'est pas toujours où l'on croit, et chercher à optimiser celle-ci. Car malheureusement les algorithmes efficaces en temps sont souvent plus complexes du point de vue de la programmation et le temps gagné en calcul est perdu en temps de programmation, traque des bugs et maintenance.

1.3 Le vecteur : un tableau encapsulé dans une classe

1.3.1 Usage

Il s'agit de la classe std :: vector (ou vector après using namespace std;) et il s'agit d'un tableau encapsulé tout simplement dans une classe, à une différence de taille : il se charge de gérer sa mémoire et l'ajuste dynamiquement à la demande. On l'obtient par un #include <vector>. Un vecteur de `double` se construit ainsi et ajouter les éléments se fait par la méthode push_back :

```
vector<double> vect;
vect.push_back(3.1416);
vect.push_back(2.718);
cout << vect.size() << " elements: "
     << vect[0] << " " << vect[1] << endl;
```

Remarquons que :

— Nous n'avons pas eu besoin de déclarer le nombre d'éléments que nous mettrons dans *vect* lors de sa construction. C'est ici le constructeur sans argument qui est appelé, la classe proposant aussi des constructeurs prenant un nombre initial d'éléments.

— Nous pouvons retrouver la taille du tableau à tout moment avec la méthode size.

— On accède aux éléments du vecteur comme pour un tableau, ce qui est possible grâce à la définition de l'opérateur [] pour la classe vector.

Il y a cependant des erreurs à éviter :

```
std::vector<double> vect;
vect[0] = 3.1416; // NON, vect[0] n'existe pas
```

4. et notre temps sur Terre est limité...
5. "Premature optimization is the root of all evil" (Donald Knuth)

```
vect.push_back(2.0); // OK, vect[0]==2
vect[0] = 3.1416; // C'est bon maintenant, vect[0] existe
```

L'opérateur = et le constructeur par recopie sont définis comme il faut, mais justement, il faut éviter de l'appeler inutilement :

```
void print(std::vector<double> vect) { // Constr. par copie
    for(int i=0; i < vect.size(); i++)
        std::cout << vect[i] << " ";
    std::cout << std::endl;
}
```

Le paramètre vect de la fonction print est passé par valeur, ce qui appelle donc un *constructeur par recopie*. Celui-ci alloue la mémoire nécessaire et recopie le paramètre passé en entrée de print, ce qui est inutile. Coût : $O(N)$. Il vaut bien mieux passer par référence, mais constante pour bien indiquer au compilateur qu'on ne souhaite pas modifier le tableau :

```
void print(const std::vector<double>& vect) { // Très bien
    ...
```

Bien sûr, comme il s'agit d'une classe, l'opérateur d'affectation est également défini et on peut avoir :

```
std::vector<double> vect2;
vect2 = vect; // OK, vector<double>::operator=
```

mais à nouveau, le coût est en $O(N)$.

1.3.2 Complexité

Intéressons-nous à la complexité des opérations élémentaires sur le vecteur. L'opération élémentaire est ici le nombre de lectures ou d'écritures dans une case du tableau. L'ajout d'un élément à la fin du tableau se fait en $O(1)$ de même que le retrait du dernier élément (pop_back). Par contre, l'ajout d'un élément en position i (insert) nécessite de décaler tous les éléments qui le suivent, c'est-à-dire $N-i$, donc jusqu'à $O(N)$. Le retrait d'un élément (erase) également. Par contre la lecture ou l'écriture dans une case quelconque du tableau se fait en $O(1)$. On voit que le vecteur est particulièrement efficace quand on ne fait qu'ajouter des éléments et que l'ordre n'a pas d'importance, [6] ce qui est le cas dans la plupart des cas.

1.3.3 Gestion mémoire

Nous avons passé sous silence la façon dont la classe gère sa mémoire. C'est pourtant crucial pour son efficacité. Ainsi, si la fonction push_back a besoin de réallouer de la mémoire et de recopier dans le nouveau tableau les éléments déjà existants, on passe du $O(1)$ annoncé à $O(N)$. Pour éviter donc de réallouer trop souvent, le principe énoncé plus haut est appliqué : "la mémoire est bon marché, on peut se permettre d'en abuser (un peu)". Cela se traduit par le choix suivant : chaque fois que la mémoire est pleine au moment où on veut ajouter un élément, on alloue une place mémoire double de la précédente. Ainsi si on veut ajouter N éléments, on a besoin de seulement $log_2 N$

6. mais il est toujours possible de trier le vecteur après l'avoir rempli, voir le chapitre suivant.

allocations mémoires. Par contre, si la mémoire est pleine et qu'on n'a besoin de n'ajouter qu'un élément, on aura gaché $N-1$ emplacements mémoire, mais c'est un choix assumé.

1.4 La liste chaînée

Le vecteur n'est efficace que pour les ajouts et retraits d'éléments en fin de tableau, pas au milieu (et encore moins au début). Par contre, l'accès à un élément d'indice donné s'y fait en $O(1)$. En fait, il y a possibilité d'inverser les choses : la liste chaînée permet d'ajouter et retirer un élément n'importe où en $O(1)$ mais ne permet l'accès à un élément quelconque qu'en $O(N)$. On sacrifie donc la facilité d'accès au profit de la modification. Pour cela, on stocke avec chaque élément les indices du suivant next et précédent prev dans le tableau.

```
struct chainon {
    int prev, next; // pour la liaison
    double val; // charge utile
};
```

Pour insérer l'élement d qu'on stocke à l'indice j juste après l'élément stocké à l'indice i, il suffit de faire :

```
t[j].val = d;
t[j].prev = i;
t[j].next = t[i].next;
t[i].next = j;
if(t[j].next != -1) t[t[j].next].prev = j;
```

Pour retirer l'élément stocké en i, il suffit de reconnecter ensemble les suivant et précédent :

```
if(t[i].prev != -1) t[t[i].prev].next = t[i].next;
if(t[i].next != -1) t[t[i].next].prev = t[i].prev;
```

L'indice -1 est un marqueur de début ou de fin de liste. Cette façon de faire laisse des "trous" dans le tableau t. supposant que l'indice du premier élément est $i0$, pour trouver l'élément $list[i]$ on doit faire

```
int ii = i0;
for(int j=0; j < i; j++)
    ii = t[ii].next;
double d = t[ii].val;
```

La boucle montre bien que la complexité est $O(N)$.

En réalité, les champs next et prev sont des pointeurs, c'est-à-dire des adresses en mémoire. Nous avons évité au maximum de parler de pointeurs dans ce cours, il suffit de savoir qu'un pointeur indique où trouver un élément en mémoire. La STL fournit une classe template std :: list qui implémente une liste. L'accès et l'ajout en tête et queue de list se font simplement avec des fonctions dédiées :

```
std::list<int> L;
L.push_back(1);
L.push_back(2);
```

```
L.push_front(3);
L.push_back(4);
L.pop_front();
L.pop_back();
cout << L.size() << "elements: ";
cout << L.front() << ' ' << L.back() << std::endl;
```

Ce petit code affiche “2 elements : 1 2”. L'énumération de tous les éléments ainsi que l'ajout/retrait à une autre position nous obligerait à parler d'itérateurs, qui est un concept assez similaire aux pointeurs, nous laissons cela pour la fin du chapitre.

Noter que la présentation concerne les listes doublement chaînées, où chaque élément connait son prédecesseur. Les listes simplement chaînées ne s'embarrassent pas ainsi et prennent donc moins de mémoire et sont aussi plus rapides en ajout/retrait. Par contre, on ne peut pas les parcourir à l'envers.

Bien entendu, tout comme avec les vecteurs, on prendra garde de ne pas faire des recopies inutiles de listes, notamment en les passant par valeur à une fonction.

1.5 La pile, Last In First Out (LIFO)

La pile est une structure permettant de mémoriser des données dans laquelle celles-ci s'empilent, de telle sorte que celle rangée en dernier est celle qui sera extraite en premier. On voit que la pile peut être considérée comme un cas particulier du vecteur. Le haut de la pile est le dernier élément, celui d'indice size()−1, qu'on peut aussi consulter avec la méthode back(). L'ajout d'un élément se fait donc par push_back, et le retrait du haut de la pile par pop_back, opérations en $O(1)$. Les autres méthodes de std :: vector ne sont pas utilisées pour une pile. Pour retrouver le vocabulaire spécifique et standard de la pile, la STL propose quand même une classe std :: stack :

```
std::stack<double> s;
s.push(3.1416);
s.push(2.718);
cout << s.top() << std::endl; // Affiche 2.718
s.pop();
cout << s.top() << std::endl; // Affiche 3.1416
s.pop();
assert(s.size()==0); // Ou bien assert(s.empty());
```

Pour raison d'efficacité[7], il y a une légère différence avec l'interface de pile que nous avons déjà rencontrée : la méthode `pop` *ne renvoie pas* l'élément en haut de la pile, en fait elle ne renvoie rien (`void`). Cet élément est donc perdu après l'appel à `pop`, mais on pouvait toujours le récupérer en appelant `top` juste avant.

Un structure de pile est utilisée par exemple en interne pour les appels de fonction : les variables locales sont empilées avant tout appel de fonction, puis la sortie de fonction appelée dépile ces variables qui retrouvent donc leur valeur.

7. En effet cela obligerait à créer une variable locale dans la méthode pour y recopier la valeur en haut de pile et retourner cette valeur à l'utilisateur, donc une deuxième recopie. Cela n'est guère troublant pour un type de base, mais devient plus gênant pour un type complexe, par exemple une image.

1.6 La file, First In First Out (FIFO)

Contrairement à la pile, la file correspond à la file d'attente habituelle : premier arrivé, premier servi! Nous pouvons stocker les éléments dans un vecteur, mais si le push_back est en $O(1)$, le "pop_front" (cette méthode n'existe pas en fait) serait en $O(N)$. En d'autres termes, le push de la file serait en $O(1)$ et le pop en $O(N)$. En fait, ce n'est pas la bonne solution. Celle-ci correspond à la liste chaînée que nous avons vue : list ::push_back correspond à queue::push et list ::pop_front à queue::pop. [8]

```
std::queue<int> Q;
Q.push(1);
Q.push(2);
Q.push(3);
cout << Q.size() << " elements: ";
while(!Q.empty()) {
    cout << Q.front() << ' ';
    Q.pop();
}
cout << endl;
```

Ce programme affiche "3 elements : 1 2 3 ".

La file est par exemple utilisée par `Imagine++` pour transmettre les événements à l'utilisateur. Les événements sont les actions du clavier (touche enfoncée, relachée) et de la souris (déplacement du pointeur, clic) entre autres. Un programme `Imagine++` a en fait deux parties, deux threads [9]. Le thread utilisateur correspond à vos instructions et le thread graphique tourne en coulisses et se charge de tout ce qui est affichage et événements. Ainsi, même quand vous faites de longs calculs, le rafraichissement de la fenêtre fonctionne par exemple quand vous la redimensionnez, utilisez les barres de défilement, etc [10]. Les événements, captés par le thread graphique, qui sont à gèrer par le thread utilisateur sont mis dans une file d'attente. L'appel à la fonction getEvent est l'équivalent d'un pop dans cette file. Une fonction de plus haut niveau comme getMouse appelle getEvent de façon répétitive jusqu'à tomber sur le type bouton enfoncé.

8. Comme la pile et la file peuvent semblent des cas particuliers du vecteur et de la liste chaînée respectivement, le lecteur peut être sceptique sur l'utilité d'avoir des classes dédiées pour cela. Néanmoins, cela indique une information importante sur la façon dont ces structures sont utilisées et facilite donc la compréhension du programme.

9. Les threads sont comme différents programmes qui tournent en parallèle, mais qui partagent une partie de la mémoire en commun

10. Un programme graphique ne comportant qu'un thread se programme de manière complètement différente : il faut interrompre les longs calculs par une consultation fréquente des événements en attente, comme l'instruction de rafraichissement, pour pouvoir y répondre promptement et ne pas laisser l'utilisateur dans l'angoisse de l'application qui ne répond pas.

FIGURE 1.1 – Un *vecteur* d'individus (haut-gauche), une *liste chaînée* (haut-droite) avec les bras droit pour *next* et gauche pour *previous*, une *pile* de livres (on ne peut dépiler que depuis le haut sans tout casser), et une *file* d'attente.

1.7 Récapitulatif des complexités

	vecteur	pile	file	liste
push_back	$O(1)$	$O(1)$[1]	$O(1)$[1]	$O(1)$
pop_back	$O(1)$	$O(1)$[2]	n/a	$O(1)$
push_front	$O(N)$[3]	n/a	n/a	$O(1)$
pop_front	$O(N)$[3]	n/a	$O(1)$[2]	$O(1)$
tab[i]	$O(1)$	n/a[4]	n/a	$O(N)$
insert	$O(N)$	n/a	n/a[4]	$O(1)$
erase	$O(N)$	n/a	n/a[4]	$O(1)$

[1] push_back s'appelle en fait push pour pile et file
[2] pop_back/pop_front s'appellent en fait pop pour pile et file
[3] Ces méthodes ne font pas partie de l'interface du vecteur mais la fonctionnalité équivalente peut être obtenue par insert ou erase.
[4] Ces méthodes ne font pas partie de l'interface de la file ou de la pile, mais peuvent être ajoutées avec complexité $O(1)$ dans l'implémentation que nous avons vue.

1.8 Les itérateurs

En réalité, pour indiquer un emplacement dans le tableau, comme par exemple pour l'insertion ou le retrait d'un élément, les classes de la STL utilisent des itérateurs. Ceux-ci permettent d'énumérer les éléments ou de désigner un emplacement.

Ils sont assez similaires à des pointeurs. L'intérêt de les utiliser c'est qu'il n'est pas nécessaire de définir tous les algorithmes sur toutes les classes. Il suffit de faire une fois l'algorithme et d'accéder aux éléments du tableau uniquement par l'intermédiaire des itérateurs, et non par un indice. Les itérateurs servent donc surtout de lien entre les structures et les algorithmes qu'on peut leur appliquer. Voici un exemple d'utilisation de ces itérateurs :

```
std::vector<double>::const_iterator it=vect.begin();
for(; it != vect.end(); ++it)
    std::cout << *it << " ";
std::cout << std::endl;
```

La méthode begin renvoie un itérateur sur le premier élément du tableau et end pointe une case plus loin que le dernier élément (donc attention de ne pas essayer d'accéder à l'élément pointé par cet itérateur). L'opérateur ++ fait passer à l'élément suivant et l'opérateur * retourne l'élément pointé par l'itérateur (ici un double).

Si on veut modifier les éléments, il faut utiliser iterator au lieu de const_iterator, mais cela n'est possible que si vect n'est pas const.

1.9 Autres structures

D'autres structures très classiques car souvent utiles ont été passées sous silence dans ce chapitre. Elles ne sont pas toutes proposées par défaut dans la STL, mais certaines pourraient l'être dans le futur. Citons entre autres :

— Les ensembles (std :: set), qui sont semblables à ceux des maths dans le sens qu'une même valeur ne peut pas être présente plusieurs fois. Leur fonctionnement efficace suppose qu'il existe une relation d'ordre total sur le type des éléments.

— Les fonctions (std :: map), qui associent à une *clé* (l'ensemble des clés doit être totalement ordonné) une valeur, et permettent de retrouver efficacement la valeur associée à une clé. La STL en propose une implémentation avec std :: map.

— La file de priorité (std :: priority_queue), qui maintient un ensemble ordonné mais autorise l'insertion ou le retrait efficaces d'éléments. Nous en reparlerons dans ce cours.

— La table de hachage (std :: unordered_map), qui permet une recherche efficace pourvu qu'on ait une méthode pour grouper les objets en un petit nombre de classes ordonnées, typiquement une fonction associant un nombre entier à chaque élément.

— Les graphes, formés de sommets et d'arêtes (paires de sommets). Beaucoup de problèmes se formulent facilement avec des graphes.

— Les arbres, qui sont des graphes particuliers (connexes et sans cycle).

De nombreux problèmes en particulier se modélisent très bien dans des graphes : réseau de tranport, circuit électrique, planification d'emploi du temps, etc. L'optimisation dans les graphes est un sujet à part entière en algorithmique, la recherche opérationnelle.

1.10 TP

Le TP A.1 calcule des zones d'influence de pixels, appelé diagramme de Voronoi, pour des distances d_1 et $\max$. Cela peut se faire efficacement avec une file d'attente, sans jamais avoir à calculer explicitement des distances, et avec une complexité indépendante du nombre n de points influents, alors que l'algorithme naïf calculerait $n \times N$ distances entre pixels.

Chapitre 2

Algorithmes de tri

Maintenant que nous savons ranger intelligemment nos données (voir le chapitre précédent), il faut souvent les trier. Ici, nous allons voir trois algorithmes de tris, et la conclusion n'est plus "ça dépend ce qu'on veut", mais nous avons un gagnant en pratique, QuickSort. C'est celui-ci qui est implémenté dans la STL.

—

2.1 Complexité minimale

Un algorithme de tri général et fonctionnant quel que soit la liste de valeurs en entrée doit retrouver la permutation permettant d'énumerer les éléments dans l'ordre. Il y a $N!$ permutations possibles. L'opération élémentaire est la comparaison de deux éléments. Ayant comparé deux éléments, on se retrouve alors au mieux avec deux tableaux de $N/2$ éléments. Nous verrons que c'est ce qui se passe en général avec QuickSort, et que la complexité résultante est $O(N \log N)$.

En fait, cela ne signifie pas qu'il n'est pas possible de faire mieux, mais uniquement dans le cas où des contraintes sur les valeurs en entrée sont connues. Par exemple, si les nombres en entrée prennent des valeurs parmi un petit nombre de valeurs possibles, on peut très bien faire un histogramme en $O(N)$. En supposant que les valeurs possibles sont les entiers entre 0 et 255 :

```
int histo[256];
for(int i=0; i < 256; i++)
    histo[i] = 0;
for(int i=0; i < N; i++)
    ++ histo[tab[i]];
int k=0;
for(int i=0; i < 256; i++)
    for(int j=0; j < histo[i]; j++)
        tab[k++] = i;
```

2.2 Algorithmes quadratiques

Il est extrêment simple d'imaginer un algorithme quadratique de tri. Par exemple, on peut chercher la valeur minimale v_0 dans le tableau en $O(N)$ (et on compte son

nombre n_0 d'occurences). Puis on cherche la valeur minimale v_1 parmi les éléments qui restent et ainsi de suite. On parcourt donc N fois le tableau et la complexité est donc en $O(N^2)$. Notons la ressemblance avec le tri par histogramme proposé ci-dessus. La seule différence est que les valeurs v_0, v_1, etc. ne sont pas connues à l'avance et doivent donc être trouvées.

En fait, un algorithme encore plus simple à programmer et bien connu est le tri à bulles. Chaque valeur représente le poids d'une bulle et on voit les bulles les plus légères remonter (ou plutôt aller vers le début du tableau). On parcourt le tableau et on permute un élément avec son suivant s'ils ne sont pas dans l'ordre. On itère N fois cette boucle et en fin de compte le tableau est trié :

```
for(int i=0; i < N; i++)
    for(int j=0; j+1 < N; j++)
        if(tab[j] > tab[j+1])
            swap(tab[j],tab[j+1]);
```

Cet algorithme fait clairement $N(N-1)$ soit $O(N^2)$ comparaisons. En fait on peut faire mieux en observant que dans la boucle d'indice j, dès qu'on arrive sur l'élément le plus grand, on le fait descendre jusqu'au bout du tableau. Après un premier parcours, l'élément le plus grand est donc en bout et à sa position finale. Il est alors inutile de comparer tab[N−2] et tab[N−1], et ainsi de suite. Nous pouvons donc remplacer la boucle en j par

```
for(int j=0; j+i+1 < N; j++)
    ...
```

On fait ainsi $(N-1)+(N-2)+...+1$ comparaisons, soit $N(N-1)/2$. Avec cette modification modeste, nous avons divisé par 2 la complexité (ce n'est pas à négliger !), mais cela reste du $O(N^2)$.

De nombreux autres algorithmes simples, tels que le tri par insertion ou le tri par sélection, sont simplement des variantes et restent en $O(N^2)$.

2.3 QuickSort

Nous allons étudier l'algorithme QuickSort, qui en pratique a une complexité optimale. Son principe est le suivant :

1. Nous sélectionnons un élément, appelé pivot, et cherchons sa place définitive dans le tableau final.
2. Ceci sépare le tableau en deux parties, la gauche et la droite du pivot, et nous itèrons sur chacun de ces sous-tableaux.

L'étape cruciale est de trouver la position du pivot, ce qui peut se faire en N opérations de la façon suivante :

— Nous maintienons un compteur i partant du début du tableau et l'incrémentons tant que $t[i]$ est inférieur au pivot.

— Nous avons un deuxième compteur j qui part de la fin et que nous décrémentons tant que $t[j]$ est supérieur au pivot.

— Si $i < j$, nous permutons $t[i]$ et $t[j]$ et continuons la progression des indices.

— Lorsque $i == j$, la place du pivot est i.

Quand on cherche à le programmer, il faut faire bien attention de ne pas risquer de dépasser du tableau et cela demande du soin. L'élément pivot est un élément du tableau, souvent le premier ou le dernier.

Le QuickSort fonctionne sur le principe du "divide and conquer" (diviser pour régner). On montrera au chapitre 3 que la complexité normale de l'algorithme est $O(N \log N)$, donc optimale pour un tableau général. L'important est de diviser équitablement le tableau, c'est-à-dire que $i = N/2$. Examinons maintenant ce qui se passe pour une violation extrême de cette hypothèse, où $i = 0$ (première position) ou $i = N - 1$ (dernière position). La relation de récurrence sur la complexité (en nombre de comparaisons) est alors

$$C_N = (N - 1) + C_{N-1}.$$

Si cela se reproduit à chaque étape, le résultat devient

$$C_N = (N - 1) + (N - 2) + \cdots + 1 = O(N^2).$$

Il s'agit du cas le pire, qui se produit quand le tableau est déjà trié ou bien en ordre inverse. Nous voyons que ce cas n'est pas meilleur qu'un tri à bulles. Malheureusement, ce cas le pire n'est pas si rare en pratique. Deux techniques courantes sont utilisées pour éviter ce cas défavorable :

1. Choisir comme pivot un élément tiré au hasard dans le tableau plutôt que le premier ou dernier.
2. Choisir comme pivot l'élément du milieu (médian) entre le permier, le dernier et l'élément à la moitié du tableau.

Ce choix de pivot se fait en temps constant, donc a un coût négligeable dans le cas "normal" où il serait inutile. Cela n'est vrai que lorsqu'on peut accéder à un élément quelconque en $O(1)$, ce qui est valable pour un tableau ordinaire ou un vecteur, mais faux pour une liste, comme nous l'avons vu au chapitre précédant.

Il est possible de se demander ce qu'il en est entre ces deux extrêmes : pivot au milieu ou pivot à une extrêmité. En fait il est possible de montrer que la complexité moyenne est $O(N \log N)$, même si le cas le pire reste $O(N^2)$. QuickSort est implémenté par la fonction std :: sort de la STL (utiliser #include <algorithm>).

Nous allons voir qu'en fait il existe un algorithme de tri, HeapSort, qui reste optimal dans le pire cas $O(N \log N)$.

2.4 HeapSort

Le principe de HeapSort est juste d'utiliser une file de priorité, appelée *heap* en anglais : les éléments sont ajoutés dans la file puis retirés les uns après les autres. La file de priorité, structure que nous étudierons au chapitre 4, a les caractéristiques suivantes :

— Accès à l'élément le plus prioritaire en $O(1)$;

— Ajout d'un élément en $O(\log N)$;

— Retrait d'un élément en $O(\log N)$.

Faisons durer le suspense sur le miracle d'une telle efficacité (si vous n'y tenez plus, allez voir au chapitre 4 directement), contentons-nous d'utiliser cette structure pour trier :

```
double HeapSort(std::vector<double>& v) {
    std::priority_queue<double> f;
    for(int i=0; i < v.size(); i++)
        f.push(v[i]);
    for(int j=v.size()-1; j >= 0; j--) {
        v[j] = f.top();
        f.pop();
    }
}
```

Mettre les N éléments dans la file nécessite $O(N \log N)$ comparaisons et les retirer également $O(N \log N)$. A noter qu'il s'agit du cas moyen aussi bien que du pire cas. Cet algorithme est donc de complexité asymptotique optimale dans tous les cas.

2.5 Utilisation : recherche par dichotomie, plus proche voisin

Trier des données est utile en soi pour leur présentation ordonnée, mais également pour faire de la recherche rapide. Alors que la recherche d'un élément dans un tableau devrait se faire en $O(N)$, si le tableau est pré-traité (trié en l'occurence), une dichotomie permet de le faire en $O(\log n)$:

```
int dichotomie(const std::vector<double>& V, double val) {
    int a=0, b=V.size()-1;
    while(a < b) {
        int c = (a+b)/2;
        if(V[c]==val)
            return c;
        if(V[c]<val)
            a = c+1;
        else
            b = c-1;
    }
    return (V[a]==val)? a: -1;
}
```

Cette fonction suppose que le tableau a préalablement été trié par ordre croissant. Elle retourne -1 si la valeur n'est pas trouvée. Cette procédure a l'air très simple, mais ne met pas en valeur quelques subtilités :

— Remplacer l'affectation de a en a=c peut conduire à une boucle infinie. Par contre, remplacer celle de b en b=c ne pose pas de problème.

— Tester $V[b]$ au lieu de $V[a]$ dans le return peut conduire à un résultat faux, et même à un crash du programme...

— Si V est vide, le programme pourrait également planter. Pourquoi ? Proposez une mesure rectificative.

Un exercice formateur est de réfléchir aux points ci-dessus.

Il est clair que si N est le nombre d'éléments de V, on doit faire de l'ordre de $O(\log N)$ comparaisons de type double.

Notons enfin que cette procédure peut être facilement modifiée pour trouver l'indice du plus proche élément de val dans V. Ce problème du plus proche voisin se retrouve assez fréquemment, également en dimension supérieure. Malheureusement, on ne peut plus alors compter sur une relation d'ordre. Nous avons vu un cas particulier dans le TP précédent en dimension 2 et pour le plus proche voisin en norme maximum, $\|.\|_\infty$. Pour la norme euclidienne par exemple et en dimension supérieure, on ne peut pas procèder ainsi, et on a recours souvent à un *k-d tree*, un arbre partitionant un espace à k dimensions. Ce n'est malheureusement efficace que si k est petit, sinon on tombe sous le coup de la redoutable *curse of dimensionality*. Nous n'entrerons pas dans l'analyse de ces cas, qui requièrent des structures avancées.

2.6 Conclusion

QuickSort et HeapSort sont tous deux en $O(N \log N)$ dans le cas moyen. Cependant QuickSort est en $O(N^2)$ au pire. Mais HeapSort nécessite de la mémoire supplémentaire pour la file de priorité et le cas le pire de QuickSort peut être rendu très improbable par sélection du pivot comme expliqué plus haut.

2.7 TP

Le TP A.2 illustre le tri par quelques méthodes simples, et compare celles-ci avec le fameux QuickSort.

Chapitre 3

Diviser pour régner

Un des grands principes à la base de certains algorithmes célèbres et réputés pour leur efficacité est emprunté à une maxime politique de Machiavel : "diviser pour régner" (divide and conquer). On divise le problème général à résoudre en plusieurs sous-problèmes similaires de plus petite taille, dont on assemble les solutions en un temps typiquement linéaire. Évidemment, cela nécessite une bonne compréhension du problème et une bonne dose d'astuce pour repérer la façon dont le problème peut se décomposer ainsi.

Nous avons déjà parlé du QuickSort, dont la clé est l'idée qu'on peut trouver en temps linéaire la position qu'aura un élément quelconque, appelé pivot, dans le tableau trié final et s'arranger en même temps pour que les valeurs inférieures soient toutes à gauche et celles supérieures à droite. Nous avons évoqué sa complexité usuelle optimale en $O(N \log N)$, que nous allons justifier. Puis nous parlerons d'un algorithme extrêmement célèbre calculant la transformée de Fourier discrète, qui lui aussi a une telle complexité asymptotique. Cet algorithme, la FFT, a de très nombreux usages ; il est hors de question de les énumérer tous. Nous détaillerons juste son utilisation pour la résolution de l'équation de Poisson, ce qui nous permettra d'utiliser l'éditeur de Poisson pour les images en TP.

3.1 Complexité de QuickSort

Le positionnement du pivot, vu dans le chapitre précédent, nécessite un parcours du tableau, donc $N-1$ comparaisons. Nous parlons ici des comparaisons entre éléments du tableau t, pas d'indices i et j, qui ne sont pas de même nature : ceux-ci sont des entiers, les éléments du tableau peuvent être d'un type complexe avec une relation d'ordre coûteuse à calculer. Ce sont ces dernières qu'il faut chercher à minimiser. Si nous notons C_N la complexité et que nous nous retrouvons avec un pivot en position i, nous avons la relation de récurrence :

$$C_N = (N-1) + C_i + C_{N-i-1}$$

L'indice i dépend des données du tableau, mais nous pouvons supposer qu'en moyenne il s'agit du milieu du tableau. Nous écrivons donc

$$C_N = N + 2C_{N/2}.$$

En utilisant la relation de récurrence pour $C_{N/2}$, nous obtenons

$$C_N = N + 2(N/2 + 2C_{N/4}) = 2N + 4C_{N/4}.$$

En allant un cran plus loin, il vient

$$C_N = 3N + 8C_{N/8}.$$

Le processus s'arrête lorsque l'indice de C à droite de la complexité devient 1, et alors

$$C_N = (\log N)N + NC_1.$$

Le résultat est donc donné par le terme dominant, $C_N = O(N \log N)$. Pour le vérifier rigoureusement, il faut en fait montrer par récurrence que pour $N = 2^n$ une puissance de 2, la suite $C_N = nN$ satisfait la relation de récurrence

$$C_{2N} = 2N + 2C_N,$$

ce qui n'est pas difficile : puisque $2N = 2^{n+1}$, on a

$$C_{2N} = (n+1)2N = 2N + 2nN = 2N + 2C_N.$$

3.2 Transformée de Fourier

La transformée de Fourier rapide, Fast Fourier Transform (FFT), est un algorithme rapide de calcul de la transformée de Fourier discrète (DFT) dû à Cooley et Tukey et datant de 1965. Son importance est centrale en traitement du signal et ses applications extrêmement variées. [1]

3.2.1 DFT

La transformée de Fourier discrète transforme un tableau f de N nombres réels ou complexes en un tableau $DFT(f)$ de même taille par l'opération suivante :

$$DFT(f)[k] = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} f[j] e^{-\frac{2i\pi}{N} jk}.$$

Cette transformation est réversible en appliquant l'opérateur inverse

$$IDFT(f)[k] = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} f[j] e^{+\frac{2i\pi}{N} jk},$$

qui ne diffère que par un changement de signe dans l'exponentielle. Cette symétrie, a priori étonnante, vient du fait que les vecteurs

$$e_k = \frac{1}{\sqrt{N}} \begin{pmatrix} e^{\frac{2i\pi}{N} 0 \cdot k} & e^{\frac{2i\pi}{N} 1 \cdot k} & \dots & e^{\frac{2i\pi}{N} (N-1) \cdot k} \end{pmatrix}$$

1. Outre la résolution de certaines équations aux dérivées partielles, rendue facile par le fait que la fonction exponentielle est un vecteur propre de l'opérateur de dérivation (on parle de méthode spectrale), comme nous allons le voir dans le TP, mentionnons aussi le fait que la convolution se transforme en multiplication des transformées de Fourier, et surtout le lien qu'elle fait entre les échantillons discrets d'un signal et sa version continue, formulé dans le fameux théorème de Shannon.

pour $k = 0, \dots, N-1$ forment une famille orthonormale (donc base) de $\mathbb{C}^N$ pour la forme hermitienne

$$\langle f, g \rangle = \sum_{j=0}^{N-1} f[j]\overline{g[j]}.$$

Notons que

$$DFT(f)[k] = \sum_{j=0}^{N-1} f[j]\overline{e_k[j]}, \qquad IDFT(f)[k] = \sum_{j=0}^{N-1} f[j]e_k[j].$$

En d'autres termes on a

$$DFT(f)[k] = \langle f, e_k \rangle \quad f = \sum_k DFT(f)[k]e_k.$$

Cette dernière égalité, décomposition de f sur cette base orthononormée, implique que

$$f[j] = \sum_k DFT(f)[k]e_k[j] = \sum_k DFT(f)[k]e_j[k] = IDFT(DFT(f))[j],$$

(la deuxième égalité provenant de la symétrie $e_k[j] = e_j[k]$) et donc que $IDFT \circ DFT = Id$.[2]

On peut aussi raisonner de la façon suivante :

$$\begin{aligned} DFT : \quad & \mathbb{C}^N \to \mathbb{C}^N \\ & f = \begin{pmatrix} f[0] & \dots & f[N-1] \end{pmatrix} \to DFT(f) = \begin{pmatrix} \langle f, e_0 \rangle & \dots & \langle f, e_{N-1} \rangle \end{pmatrix} \end{aligned}$$

La DFT fait passer un signal f exprimé dans la base canonique à sa décomposition dans la base $(e_0, \cdots, e_{N-1})$. Puisque ces bases sont orthonormales, la matrice de passage est unitaire, $P^{-1} = \bar{P}^\top$. Son terme général est $P_{jk} = e^{\frac{2i\pi}{N} j \cdot k}$. On a donc

$$DFT(f) = P^{-1}f = \bar{P}^\top f = \bar{P}f,$$

puisque P est symétrique. Ceci explique le signe négatif dans les exponentielles de la formule de la DFT. Appliquer la DFT, c'est multiplier le signal par $\bar{P}$, tandis qu'appliquer IDFT, c'est multiplier le signal par P.

Il est à noter que si $1 \leq k \leq N-1$, on a $e_{N-k} = \overline{e_k}$, et donc que pour un signal $f \in \mathbb{R}^N$, on a $DFT(f)[N-k] = \overline{DFT(f)[k]}$; en particulier, prenant $k = N/2$, on trouve que $DFT(f)[N/2] \in \mathbb{R}$; puisque $e_0 \in \mathbb{R}^N$, on a aussi $DFT(f)[0] \in \mathbb{R}$.

Ceci étant établi, on voit que le calcul de $DFT(f)[k]$ implique N multiplications et $N-1$ additions, donc $O(N)$ opérations. Donc le calcul de $DFT(f)$ prend $O(N^2)$ opérations. C'est là qu'intervient le miracle de la FFT, qui réduit ce coût à $O(N \log N)$.

3.2.2 FFT

Pour $N \geq 2$, on sépare la somme dans la DFT en indices pairs et impairs :

$$\sqrt{N}DFT(f)[k] = \sum_{j=0}^{N/2-1} f[2j]e^{-\frac{2i\pi}{N}(2j)k} + \sum_{j=0}^{N/2-1} f[2j+1]e^{-\frac{2i\pi}{N}(2j+1)k},$$

2. On voit souvent comme définition de la DFT la même somme non normalisée par $\sqrt{N}$, et la base n'est plus qu'orthogonale, non ortho*normale*. Il faut alors diviser par N dans IDFT pour retrouver le signal initial.

dont on déduit facilement

$$\sqrt{N}DFT(f)[k] = \sum_{j=0}^{N/2-1} f[2j]e^{-\frac{2i\pi}{N/2}jk} + e^{-\frac{2i\pi}{N}k} \sum_{j=0}^{N/2-1} f[2j+1]e^{-\frac{2i\pi}{N/2}jk}.$$

Sous cette forme, on reconnait

$$\sqrt{N}DFT(f)[k] = \sqrt{N/2}\left(DFT(f(\texttt{0:2:N-2}))[k] + e^{-\frac{2i\pi}{N}k}DFT(f(\texttt{1:2:N-1}))[k]\right), \tag{3.1}$$

expression dans laquelle on a utilisé la notation Matlab [3]

$$f(\texttt{deb:pas:fin}) = \left(f[\texttt{deb}] \quad f[\texttt{deb+pas}] \quad \ldots \quad f[\texttt{fin-pas}] \quad f[\texttt{fin}]\right).$$

On a donc réduit le problème à

1. Calculer la DFT des indices pairs de f.
2. Calculer la DFT des indices impairs de f.
3. Combiner en $O(N)$ les deux suivant la formule (3.1).

Les deux DFTs auxiliaires étant de longueur $N/2$, on voit que cette relation de récurrence suit parfaitement celle qu'on a rencontrée pour QuickSort, et donc on a bien une complexité en $O(N \log N)$ pour la FFT. La récursion s'arrête pour $N = 1$ avec $DFT(f)[0] = f[0]$.

3.2.3 Implémentation

Dans la formule de récurrence (3.1), il y a un petit raccourci subtil : les deux sous-DFT sont de longueur $N/2$, donc ne sont réellement définies que pour $0 \leq k < N/2$, or le membre de gauche fait varier k jusqu'à $N-1$. En fait, avec $\exp(-\frac{2i\pi}{N/2}j(k+N/2)) = \exp(-\frac{2i\pi}{N/2}jk)$ et $\exp(-\frac{2i\pi}{N}(k+N/2)) = -\exp(-\frac{2i\pi}{N}k)$, on peut écrire l'étape de mise à jour 3 ci-dessus de la façon suivante :

— Pour résoudre la difficulté d'écrire les résulats dans le même tableau, copier f dans un tableau temporaire `buffer`. On a dans les indices pairs et impairs les résultats des sous-DFT (non normalisées).

— Boucle de $k = 0$ à $N/2 - 1$:

$$f[k] \leftarrow \texttt{buffer}[2*k] + e^{-\frac{2i\pi}{N}k}\,\texttt{buffer}[2*k+1]$$
$$f[k+N/2] \leftarrow \texttt{buffer}[2*k] - e^{-\frac{2i\pi}{N}k}\,\texttt{buffer}[2*k+1].$$

Le facteur $t_k = e^{-\frac{2i\pi}{N}k}$, souvent appelé *twiddle*, peut se calculer plus rapidement qu'en faisant appel aux fonctions trigonométriques par la relation de récurrence de suite géométrique $t_{k+1} = r\,t_k$ (avec $t_0 = 1$), de raison $r = e^{-\frac{2i\pi}{N}}$ pré-calculée.

Pour éviter d'allouer/libérer le tableau `buffer` à chaque appel, on peut ne le faire qu'une fois, d'une taille suffisante N, et le passer dans les appels récursifs, même si les sous-DFT n'utilisent qu'une partie du tableau.

3. Syntaxe impossible en `C++`, car ce symbole `:` est déjà réservé pour un autre usage.

3.2.4 Dérivation

Si on définit la fonction $e_j(x) = \frac{1}{\sqrt{N}} \exp(\frac{2i\pi}{N} jx)$, on voit que cette fonction périodique de période N coïncide avec les composantes de nos vecteurs de base, $e_j(k) = e_j[k]$ pour $k = 0, 1, \ldots, N-1$. On peut donc à partir du tableau f définir une fonction périodique également notée f par la formule :

$$f(x) = \sum_j DFT(f)[j] e_j(x),$$

et cette fonction périodique coïncide avec le tableau f aux indices entiers (fonction interpolante). On peut calculer sa dérivée :

$$f'(x) = \sum_j DFT(f)[j] e_j'(x) = \sum_j \frac{2i\pi j}{N} DFT(f)[j] e_j(x).$$

Par identification avec la formule de reconstruction, on peut donc écrire

$$DFT(f')[j] = \frac{2i\pi j}{N} DFT(f)[j].$$

Cette propriété remarquable est très utile pour résoudre certaines EDP.

Il faut corriger un peu cette formule pour les fonctions réelles. Notons que d'après la formule ci-dessus, on a $e_{j-N}(k) = e_j(k)$ pour k entier. Les fonctions e_j et e_{j-N} coïncident donc aux valeurs entières,[4] mais sont bien sûr des fonctions différentes. Cela a une influence sur la dérivée. La bonne interprétation est de préférer les fréquences négatives $j - N$ plutôt que j pour $N/2 < j < N$ (pensez à $\cos(x) = (e^{ix} + e^{-ix})/2$ qui a donc bien une fréquence négative).[5] Pour $j = N/2$, cela peut aussi bien être la fréquence $-N/2$. Pour ne pas avoir à choisir, on préfère mettre cette valeur à 0.[6] On amende donc la formule de dérivation par la suivante :

$$DFT(f')[j] = \begin{cases} \frac{2i\pi}{N} j\, DFT(f)[j] & \text{pour } 0 \leq j < N/2 \\ 0 & \text{pour } j = N/2 \\ \frac{2i\pi}{N} (j-N)\, DFT(f)[j] & \text{pour } N/2 < j < N \end{cases}$$

3.3 TP

Le TP A.3 sera en trois parties. La première consistera en l'implémentation de la FFT et nous servira de fondation pour les deux autres parties. Celles-ci utilisent l'équation de Poisson pour obtenir des effets intéressants sur les images. L'idée est de modifier le gradient en certains points dans l'image avant de reconstruire une image dont le gradient est ainsi prescrit. Typiquement, on cherche à trouver l'image u vérifiant l'équation suivante :

$$\nabla u = \begin{pmatrix} V_x \\ V_y \end{pmatrix}$$

4. On dit que e_j est un alias de e_{j-N}

5. Cette opération est tellement courante que Matlab a une fonction `fftshift` qui déplace les fréquences au-delà de $N/2$

6. Une autre bonne raison est que la fréquence $N/2$ d'une fonction réelle est toujours réelle, tout comme la fréquence 0 : $DFT(f)[0] = \sum_j f[j] \in \mathbb{R}$ et $DFT(f)[N/2] = \sum_j (-1)^j f[j] \in \mathbb{R}$, contrainte que ne respecterait pas la multiplication par $i\pi$ de ce dernier.

FIGURE 3.1 – Exemples d'utilisation de l'éditeur de Poisson : réhaussement de contraste et copier/coller subtil.

avec des conditions aux bords. En prenant la divergence de chaque membre, on se retrouve avec l'équation de Poisson $\Delta u = f$:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y}.$$

Comme on a vu que la dérivation se traduit par une multiplication des coefficients de la DFT, il suffit de diviser pour intégrer et on peut retrouver u.

La première application consiste en le réhaussement du contraste des zones sombres de l'image. On regarde tous les pixels où le gradient est inférieur à un seuil (typiquement 50 sur une échelle de 0 à 255) et on multiplie le gradient en ces points par un facteur α, par exemple $\alpha = 3$, avant de reconstruire l'image.

La deuxième application est l'éditeur de Poisson. L'objectif est de prendre un morceau d'une image pour l'insérer dans une autre. Un simple copier/coller est forcément évident et ne produit pas une image naturelle. Mais si au lieu de cela on copie/colle le gradient, on obtient des résultats bien plus harmonieux.

Chapitre 4

File de priorité

Nous avons parlé de HeapSort, le concurrent de QuickSort qui se comporte toujours avec une complexité asymptotique optimale, mais nous avons passé sous silence la structure permettant une telle efficacité. En réalité, cette structure, la file de priorité, a une utilité bien plus générale que lors de la tâche de tri, et nous lui consacrons ce court chapitre.

—

4.1 Introduction

Nous avons vu la file FIFO au chapitre 1. En fait, nous allons modifier cette file en considérant que tous les éléments ne sont pas égaux mais que chacun a une priorité. L'élément de plus grande priorité sera en tête, c'est-à-dire le premier à sortir.

Étant donné sa grande utilité, la STL fournit déjà une file de priorité, la classe template std :: priority_queue. C'est bien entendu celle-ci qu'on utilisera en pratique, mais il est intéressant en soi de comprendre comment elle fonctionne. Nous allons donc voir comment on pourrait implémenter une telle structure.

Une solution simple qui vient à l'esprit est de maintenir à tout moment les éléments de la file triée par ordre de priorité. En fait, ce n'est pas pratique :

1. Si on peut accéder à un élément arbitraire de la file en $O(1)$, on peut trouver l'emplacement d'un élément à ajouter par dichotomie en $O(\log N)$ comparaisons, mais alors l'insertion se fait en $O(N)$ copies d'éléments dans le tableau.
2. Si on peut insérer l'élément en $O(1)$, comme pour une liste, il faut alors $O(N)$ comparaisons pour trouver l'emplacement où l'insérer.

En réalité, il suffit de s'assurer qu'à tout moment nous pouvons accéder facilement à l'élément de plus grande priorité, sans qu'il soit nécessaire de tout trier. Une structure d'arbre binaire vérifiant le principe suivant est bien adaptée : tout parent a une priorité au moins aussi grande que ses deux enfants. Il est alors facile de voir que la racine est l'élément de plus grande priorité. Voyons d'abord comment insérer un élément.

4.2 Insertion dans la file de priorité (push)

Nous commençons par ajouter le nouvel élément comme feuille de l'arbre à profondeur minimale, puis nous réparons l'arbre : s'il est plus grand que son parent, nous permutons les deux, comparons au nouveau parent et ainsi de suite jusqu'à arriver à

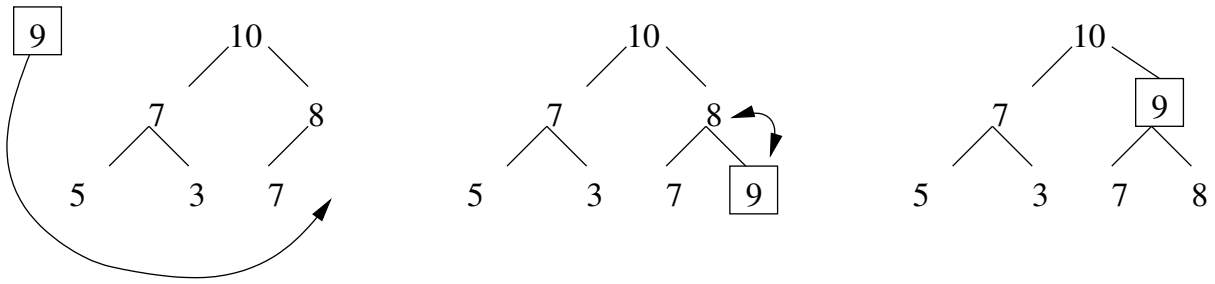


FIGURE 4.1 – Fonction push dans la file de priorité

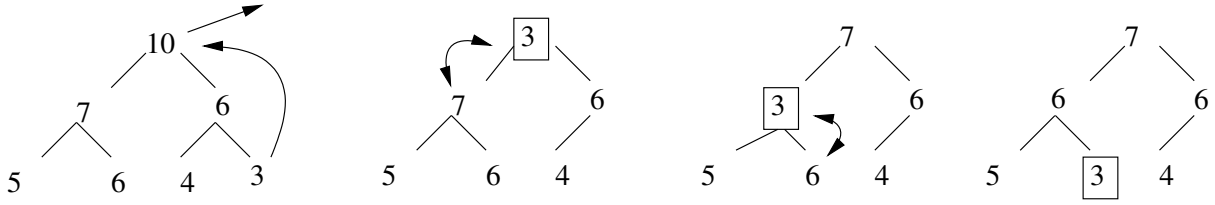


FIGURE 4.2 – Fonction pop dans la file de priorité

la racine ou à un parent qui a plus grande priorité (Fig. 4.1). L'arbre étant équilibré, il a une profondeur $\log_2 N$ et donc $O(\log N)$ comparaisons sont nécessaires.

4.3 Retrait de la file de priorité (pop)

L'élément en tête est la racine de l'arbre. Nous retirons l'élément la dernière feuille de l'arbre pour la mettre à la racine et réparons l'arbre : si la racine a plus faible priorité que le plus grand des (un ou deux) enfants, nous permutons les deux et comparons aux deux nouveaux enfants et ainsi de suite (Fig. 4.2). Le processus s'arrête lorsque nous atteignons une feuille ou bien lorsque l'élément a plus grande priorité que ses enfants. L'arbre étant équilibré, nous descendons l'élément au plus jusqu'à la profondeur maximale, soit $O(\log N)$.

4.4 Stockage de la file de priorité

Pour stocker cet arbre binaire équilibré, il est très efficace d'utiliser un simple tableau. Pour une fois, nous considérerons que le premier élément du tableau est d'indice 1 et non 0. Le C++ n'autorise pas cela, mais plutôt que de jouer sur les indices, il est plus facile d'ignorer simplement l'indice 0 du tableau. La racine est donc en 1, ses enfants en 2 et 3, les enfants de 2 en 4 et 5 et ceux de 3 en 6 et 7, etc. Les enfants de l'élément d'indice i sont en $2i$ et $2i+1$. Son parent en indice $i/2$ (quotient de la division euclidienne).

4.5 Application : algorithme de Dijkstra

Ce célèbre algorithme, dû à Esdger Dijkstra,[1] calcule les plus courts chemins dans un graphe : chaque arête, reliant deux nœuds, a une distance associée, et le but est

1. (1930-2002), informaticien néerlandais, prix Turing 1972 (équivalent du prix Nobel)

de calculer le chemin le plus court entre un point de départ (un nœud) et un ou des points d'arrivée (éventuellement tous les autres nœuds). Pensez par exemple au plan du métro, les nœuds sont les stations et les arêtes ont chacune un temps de parcours. On se restreint au cas particulier où le coût d'entrée dans un nœud est le même pour toutes les arêtes, ce qui simplifie l'algorithme. Il procède ainsi :

1. Initialiser toutes les distances D_p à l'infini.
2. Mettre les nœuds de départ dans une file de priorité F avec priorité 0 et fixer leur distance $D_p = 0$.
3. p=F.pop(), et pour tous les voisins q de p (reliés par une arête de poids $w_q > 0$) :
   (a) Si $D_q > D_p + w_q$, mettre D_q à cette valeur ; de plus, si q est dans la file, changer (augmenter) sa priorité à $-D_q$, sinon F.push(q,$-D_q$).
4. Retourner en 3 si F n'est pas vide.

Notez qu'on transforme une file de priorité orientée max en une file de priorité orientée min en prenant l'opposé de chaque priorité.

Quand on regarde l'étape 3a, on se rend compte d'un petit problème : comment savoir si q est dans la file pour changer sa priorité ? La file se prête mal à une recherche efficace.[2] Il faut donc suivre toutes les opérations qui ont lieu dans la file, ce qui est lourd et fastidieux du point de vue programmation. Il y a une variante plus intelligente, où on accepte d'avoir le même nœud plusieurs fois dans la file avec des priorités différentes. Pour ne pas le retraiter une deuxième fois avec une priorité inférieure à la première, une simple vérification avec la distance actuellement affectée est suffisante. Ainsi l'algorithme modifié devient le suivant.

1. Initialiser toutes les distances D_p à l'infini.
2. Mettre les nœuds de départ dans une file de priorité F avec priorité 0.
3. (p,d)=F.pop() avec d la priorité. Si $-d \geq D_p$, ne rien faire de plus et passer à l'itération suivante. Sinon :
   (a) $D_p \leftarrow -d$.
   (b) Pour tous les voisins q de p (reliés par une arête de poids $w_q > 0$) : si $D_q > D_p + w_q$, ajouter dans la file F.push(q,$-D(q)$).
4. Retourner en 3 si F n'est pas vide.

Il est important d'affecter D_p *après* sortie de la file et non avant, de manière à pouvoir continuer la propagation, sinon on pourrait croire que le nœud a déjà été propagé à une distance inférieure.

2. Une structure adaptée pour ça est un arbre binaire de recherche, non abordé dans ce cours.

Annexe A

Travaux Pratiques

A.1 Diagramme de Voronoi

Dans ce TP, nous allons calculer les zones d'influence de points du plan $\{p_i\}_{i=1,\dots,5}$ (figure A.1). On définit une zone d'influence d'un point p_i comme l'ensemble des points plus proches de p_i que de tout autre p_j, ceci pour une distance donnée.

Si la distance choisie est la distance euclidenne d_2, il s'agit du diagramme de Voronoi classique, très utilisé en géométrie algorithmique, délimité par des portions de médiatrices des paires de points initiaux ; l'algorithme pour son calcul est bien différent de celui que nous allons détailler, qui concerne les distances d_1 et d_∞ :

$$d_1(p,q) = |x_p - x_q| + |y_p - y_q| \qquad d_\infty(p,q) = \max(|x_p - x_q|, |y_p - y_q|).$$

La particularité de ces distances est que si on connait l'ensemble P des pixels à distance $\leq d$ de p_i, les pixels à distance $d+1$ sont les 4-voisins (8-voisins pour d_∞) des pixels de P qui ne sont pas dans P.

L'algorithme est alors le suivant :

1. Mettre les points de départ p_i dans une file d'attente vide, chacun avec sa couleur, marquer ces pixels comme explorés.
2. Retirer l'élément p en tête de file.
3. Mettre les voisins q de p dans la file, avec la même couleur que p, et les marquer comme explorés.
4. Si la file n'est pas vide, retourner à l'étape 2.

Notez qu'on ne calcule jamais des distances, et que la complexité est indépendante du nombre de points p_i.

Question (répondre en commentaire dans le main.cpp) : Pourquoi un algorithme récursif ne fonctionnerait pas dans ce cas ?

A.1.1 File utilisant une liste

1. *Pour commencer, étudiez le projet :*
   Téléchargez le fichier `Voronoi_Initial.zip` sur la page du cours, le décompresser et lancez votre IDE. Vérifier que le programme compile et se lance, même s'il ne fait rien d'intéressant.
2. *Initialisez le tableau t*
   Il indique si le pixel a été exploré ou non.

FIGURE A.1 – Zones d'influence de 5 points pour la distance d_∞ (gauche) et d_1 (droite).

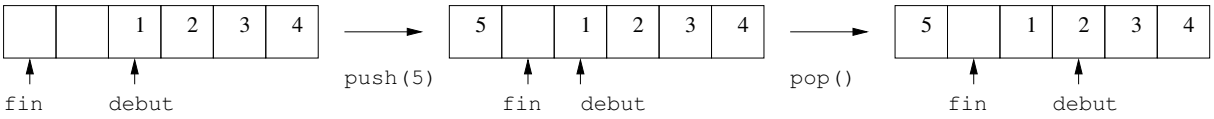


FIGURE A.2 – Les fonctions `push` et `pop` d'une file implémentée dans un tableau circulaire. Après le `pop`, on se contente d'avancer l'indice de début.

3. *Faites choisir à l'utilisateur les points de départ*
   Chacun prend une couleur aléatoire, l'utilisateur termine la sélection par un clic droit.
4. *La file d'attente*
   Vous avez l'interface de `fileListe`, qui s'appuie sur `std::list<point>`. Compléter son implémentation.
5. *Diagramme de Voronoi*
   Programmer le calcul des zones d'influence en suivant l'algorithme indiqué.

A.1.2 File utilisant un tableau circulaire

Nous allons nous compliquer les choses pour ne plus dépendre de `std::list`, mais nous allons utiliser un tableau circulaire. C'est un tableau classique d'une certaine taille, alloué dynamiquement, mais nous maintenons deux indices indiquant la tête de liste et la case suivant la fin de liste. Ces indices sont circulaires, c'est-à-dire qu'on revient à 0 quand on arrive à la fin du tableau. Le principe suit le schéma de la figure A.2.

La pile est vide lorsque debut==fin. Lors d'un `push`, si on se retrouve avec le cas précédent, c'est que le tableau est trop petit. On crée alors un tableau deux fois plus grand dans lequel on copie les éléments.

6. *Changez de file*
   Vous voyez un typedef dans le code, qui indique un alias pour un nom de type. Sélectionnez le type `fileTableau` maintenant. Comme sa partie publique est identique à celle de `fileListe`, vous ne devez rien changer d'autre dans le

`main.cpp`. Le code devrait continuer à compiler, mais bien sûr le calcul du Voronoi ne fonctionne plus.

7. *Constructeur et destructeur de fileTableau*
   Implémentez-les en allouant un seul élément pour le tableau dans le constructeur.
8. *pop et empty*
   Ces méthodes sont simples, quelques lignes suffisent.
9. *push*
   Celle-ci est plus délicate car on peut avoir à réallouer. Dans un premier temps, modifiez le constructeur pour allouer un grand nombre d'éléments et ne pas risquer de tomber dans ce cas. Vérifiez que le calcul du Voronoi fonctionne toujours.
10. *Tableau dynamique*
    Remettez le constructeur avec un seul élément alloué, et implémentez la réallocation dans le `push` quand elle est nécessaire.
    Si vous n'arrivez pas à faire cette question, remettez-vous à la situation de la question précédente pour refaire fonctionner le programme.

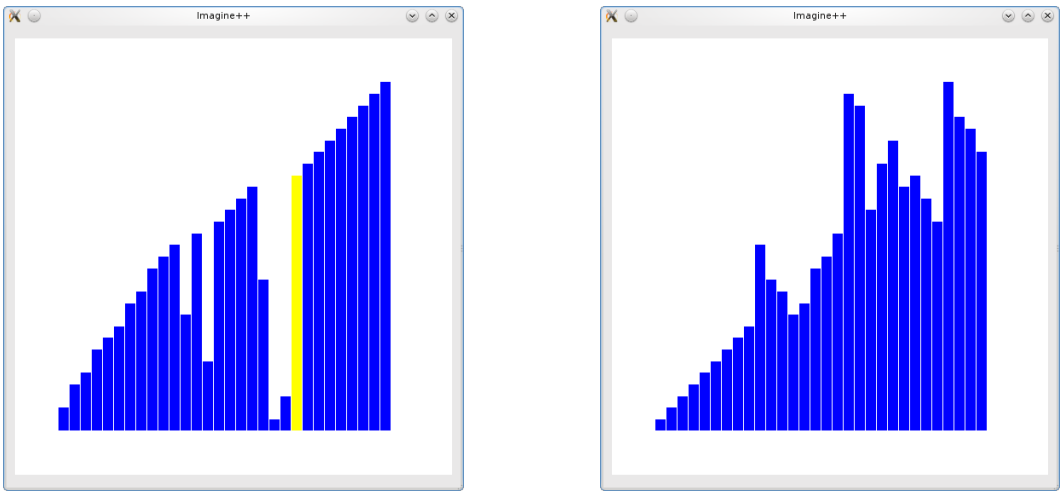


FIGURE A.3 – Deux tris en cours d'exécution : tri à bulle et Quicksort...

A.2 Les tris

Dans ce TP, nous allons programmer quelques algorithmes de tri d'éléments dans un tableau. Pour que cela soit interactif, et que l'on puisse voir comment se passent chacun des tris, une interface graphique a été programmée, qui affiche le tableau et permet de visualiser les opérations qu'on réalise dessus (figure A.3).

A.2.1 Mélanger un tableau

1. *Pour commencer, étudier le projet :*
   Télécharger le fichier `Tris_Initial.zip` sur la page habituelle, le décompresser et lancer votre IDE. Le projet est séparé entre le fichier `main.cpp`, dans lequel on programmera les algorithmes de tri, et le couple (`tools.cpp`, `tools.h`), qui gère l'interface graphique et quelques fonctions utiles à la comparaison des différents tris.
   Ouvrir `tools.h` pour découvrir les fonctions de cette interface graphique, puis `main.cpp` pour voir comment elles sont utilisées (les lignes commentées sont là pour montrer comment vous utiliserez les fonctions `melange_tableau` et `tri_selection` que vous allez programmer).
   Exécuter le projet. Pour l'instant, il ne fait qu'initialiser un tableau et l'afficher.
2. *Accès à un tableau, échange de deux valeurs*
   Pour que les opérations que vous effectuerez sur les tableaux soient affichées automatiquement, il faudra que vous utilisiez uniquement les fonctions `valeur` et `echange` déclarées dans `tools.h` (ne pas accéder directement au tableau avec `T[i]`). Entraînez-vous à les utiliser dans la fonction main(), en accédant à une valeur du tableau `T0`, et en permutant deux de ses éléments.
3. *Sécuriser les accès tableau*
   Pour s'assûrer qu'on ne sort pas des bornes du tableau, ajouter des appels à la fonction assert dans `valeur` et `echange`. Constater ce qui se passe à l'exécution lorsqu'on appelle par exemple

```
valeur(T0, taille, taille);
```

4. *Mélanger un tableau*
   Une fonction déclarée dans `tools.h` n'existe pas encore dans `tools.cpp` : la fonction

   ```
   void melange_tableau (double T[], int taille)
   ```

   qui, comme son nom l'indique, mélange un tableau. Ajoutez-la (dans `tools.cpp` bien sûr)
   Idée : le mélange le plus rapide (et le plus efficace) consiste à parcourir le tableau une fois, et à permuter chacun des éléments avec un autre choisi au hasard (utiliser la fonction int random(int a) définie dans `tools.cpp` pour tirer un entier entre `0` et `a-1`).

A.2.2 Tris quadratiques

Les trois algorithmes de tri qui suivent trient un tableau en temps $0(n^2)$, c'est à dire que le temps pour trier un tableau est proportionnel au carré de la taille de ce tableau.
Le temps du TP étant court, ne faites qu'un des algorithmes ci-dessous avant de passer au QuickSort, plus délicat ; vous programmerez les autres à la maison.

5. *Tri sélection*
   C'est le tri le plus naïf. Il consiste à parcourir le tableau une première fois pour trouver le plus petit élément, mettre cet élément au début (par une permutation), parcourir une seconde fois le tableau (de la 2ème à la dernière case) pour trouver le second plus petit élément, le placer en 2ème position, et ainsi de suite...
   Programmer la fonction void tri_selection (double T[], int taille). Vous pouvez alors décommenter les lignes commentées dans main() et exécuter.
6. *Tri insertion*
   En général, c'est à peu près l'algorithme qu'utilise un être humain pour trier un paquet de cartes, des fiches...
   Il consiste à ajouter un à un les éléments non triés au bon endroit parmi ceux qui sont déjà triés : si nécessaire on échange les deux premiers éléments du tableau pour les mettre dans l'ordre, puis (si nécessaire) on déplace le 3ème élément vers la gauche, par des échanges de proche en proche, jusqu'à ce qu'il soit à la bonne position par rapport aux deux premiers, puis le 4ème, et ainsi de suite...
   Programmer void tri_insertion (double T[], int taille) et regarder comment ça se passe.
7. *Tri à bulle*
   Le tri à bulle consiste à parcourir n fois le tableau, et à chaque fois qu'on est sur un élément, on l'échange avec son voisin de droite si ce dernier est plus petit que lui.
   Programmer `tri_bulle(double T[], int taille)` et regarder comment ça se passe. Constater qu'on n'est pas obligé de parcourir tout le tableau à chaque fois et améliorer la fonction.

A.2.3 Quicksort

L'algorithme :
Le tri Quicksort adopte la stratégie *"diviser pour régner"* qui consiste à réduire le

problème du tri d'un tableau de taille n aux tris de deux tableaux de taille $\frac{n}{2}$:
on choisit un élément dans le tableau (on le prend en général au hasard, mais par commodité on prendra ici le premier élément du tableau) qu'on appelle **pivot**. On sépare ensuite les autres éléments entre ceux inférieurs au pivot et ceux supérieurs au pivot. Il n'y a plus alors qu'à trier ces deux moitiés.

8. *Pivot*
   - Créer une fonction int pivot(double T[], int taille) qui prend comme pivot le premier élément du tableau, échange les éléments du tableau de manière à ce qu'on aie d'abord les éléments inférieurs au pivot, puis le pivot, puis les éléments supérieurs au pivot, et qui renvoie la nouvelle position du pivot.
   - Ne pas oublier d'exécuter la fonction pour vérifier qu'elle marche bien !
   - Idée :
     - on utilise deux index qui parcourent le tableau, le premier à partir du 2ème élément (le 1er étant le pivot) et avançant vers la droite, le second à partir du dernier élément et avançant vers la gauche
     - le premier index s'arrête sur le premier élément supérieur au pivot qu'il rencontre, et le second index, sur le premier élément inférieur au pivot qu'il rencontre
     - on échange alors les deux éléments, et les deux index continuent d'avancer, et ainsi de suite
     - quand les deux index se rencontrent, à gauche de l'intersection tous les éléments sont inférieurs au pivot, et à droite ils sont supérieurs
     - on échange le pivot (en 1ère position) avec le dernier des éléments inférieurs pour obtenir ce qu'on désire

     Attention, des différences apparemment anodines dans la façon de programmer cette fonction peuvent la faire échouer dans des cas particuliers : réfléchissez au cas où le pivot est la valeur minimale ou la valeur maximale du tableau, si le tableau est vide…

9. *Fonctions récursives*
   Le principe même de la statégie *diviser pour régner* implique que la fonction qui effectue le tri quicksort va s'appeler elle-même pour trier une sous-partie du tableau. En pratique, on va utiliser deux arguments `debut` et `fin` qui indiquent qu'on ne réalise le tri qu'entre les indices `debut` et `fin`.
   Changer la fonction `pivot` en lui ajoutant ces deux arguments, et en ne la faisant effectivement travailler que entre ces deux indices (par exemple, le pivot est initialement à l'indice `debut`)

10. *Quicksort*
    On peut maintenant écrire une fonction
    void quicksort_recursif(double T[], int taille , int debut, int fin),
    qui contient l'algorithme, ainsi que la fonction
    void quicksort(double T[], int taille),
    qui ne fait qu'appeler cette dernière, mais qui sera celle qu'on utilisera dans main() (car plus simple).

A.2.4 Gros tableaux

Maintenant qu'on a vu graphiquement comment marchent ces algorithmes, il est intéressant de les faire fonctionner et de les comparer sur des tableaux de grande taille.

11. *Tableaux de taille variable*
Si on veut tester nos algorithmes sur des tableaux de grande taille, il faut utiliser l'allocation dynamique. Dans la partie délimitée par #ifdef BIG,[1] remplacer toutes les lignes de type double t[taille]; par double *t = new double[taille], et à la fin des fonctions où se trouvent ces déclarations, ajouter la ligne delete[] t;.[2]
D'autre part il faut désactiver l'affichage :

```
init_tools(512, false);
```

12. *Nombre de lectures et d'écriture*
Pour comparer plus précisément deux algorithmes, on peut compter le nombre de lectures du tableau et le nombre d'écritures dans le tableau.
Créer deux variables globales[3] servant de compteurs dans `tools.cpp` et modifier les fonctions `init_tri`, `valeur`, `echange` et `fin_tri` pour initialiser, compter et afficher le nombre moyen de lectures et d'écritures par élément du tableau. **Au fait, en combien d'opérations s'effectue en moyenne Quicksort ?**

13. *Temps de calcul*
Il est intéressant également d'avoir les temps de calcul exacts. Pour cela, on peut enregistrer dans une nouvelle variable globale `timer0` le temps qu'il est avant le tri :
timer0 = double(clock())/CLOCKS_PER_SEC;
et la retrancher au temps qu'il est après le tri (modifier les fonctions `init_tri` et `fin_tri` pour faire ce calcul et l'afficher).

14. *Mode Release*
On peut changer le mode de compilation en le passant de *Debug* à *Release*. Vérifier que l'exécution des algorithmes est effectivement bien plus rapide en mode *Release* !

Pour travailler sur votre TP, il est normal de mettre en commentaire la partie du code correspondant à une question résolue pour ne pas perdre de temps lors du travail sur les questions en cours de traitement. Néanmoins, une fois le TP terminé et avant de le remettre sur educnet, enlevez les parties de code en commentaire, *recompilez* et *vérifiez* que tout fonctionne depuis le début !

1. L'instruction #ifdef fait de la compilation conditionnelle : selon que `BIG` est défini ou non, il compile une partie du code et ignore l'autre. Pour définir BIG, on aurait pu le faire par un #define BIG, mais on peut aussi le faire au niveau du CMakeLists.txt (solution adoptée), ce qui permet de compiler deux fois le même code, une fois sans `BIG` (programme `Tris`), une fois avec (programme `TrisBig`). Donc pour travailler avec les grands tableaux, mettez bien comme programme à lancer `TrisBig`.
2. en entourant la désallocation par #ifdef BIG et #endif pour que cela n'affecte que le programme `TrisBig`.
3. Comme vous le savez, ce n'est pas bien d'utiliser des variables globales, mais c'est ici bien commode. Pour atténuer cette faute, on les met ici dans un namespace global, ce qui permet au moins de les identifier facilement comme telles, comme dans global :: fenetre.

A.3 Éditeur de Poisson

Ce long TP nous occupera plusieurs séances. La première partie sera consacrée à l'implémentation de la FFT et servira pour la résolution de l'équation de Poisson qui surviendra dans des tâches d'édition d'images.

Utilisation des nombres complexes

Le C++ a déjà une classe de complexes, paramétrée (template) par le type codant les parties réelle et imaginaire. Voici un exemple d'utilisation :

```
#include <complex>
#include <iostream>
int main() {
  std::complex<float> i(0, 1); // Constructeur
  std::cout << i*i << std::endl; // complexe*complexe: (-1,0)
  std::complex<float> j = std::polar<float>(1,2*M_PI/3); // Polaire
  std::cout << i+j << std::endl; // complexe+complexe: (-0.5,1.86603)
  float f=2;
  std::cout << f*j << std::endl; // float*complexe: (-1,1.73205)
  f = j.real()+j.imag();
  j = f; // Conversion automatique de float en complexe.
  f = j; // Erreur, on ne peut pas convertir un complexe en float
  return 0;
}
```

A.3.1 Partie 1 : FFT

1. *Pour commencer, étudier le projet :*
   Télécharger le fichier `Poisson_Initial.zip` sur la page habituelle, le décompresser. Le `main` se trouve dans test_fft .cpp et n'aura pas à être modifié. Notez qu'on manipule des complexes dont les membres sont stockés en float, le type std :: complex<float>. On utilise les méthodes real () et imag() pour lire ou écrire les parties réelle et imaginaire. Bien entendu, tous les opérateurs standard sont définis pour les complexes, comme l'addition, soustraction, multiplication, division...
2. *DFT*
   La fonction `dft` calcule le coefficient d'indice k de la DFT du tableau f suivant la formule vue en cours :

$$DFT(f)[k] = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} f[j] e^{s \frac{2i\pi}{N} jk}.$$

   Le réel s vaut -1 pour la DFT et $+1$ pour la DFT inverse. Implémenter cette fonction. On pourra utiliser la fonction suivante définie par le header `complex` :

   ```
   std::complex<float> std::polar(float r, float theta)
   ```

   qui cacule le nombre complexe $r\, e^{i\theta}$.
3. *FFT*
   Les fonctions `fft` et `ifft` font toutes deux appel à fft_main en changeant le s en

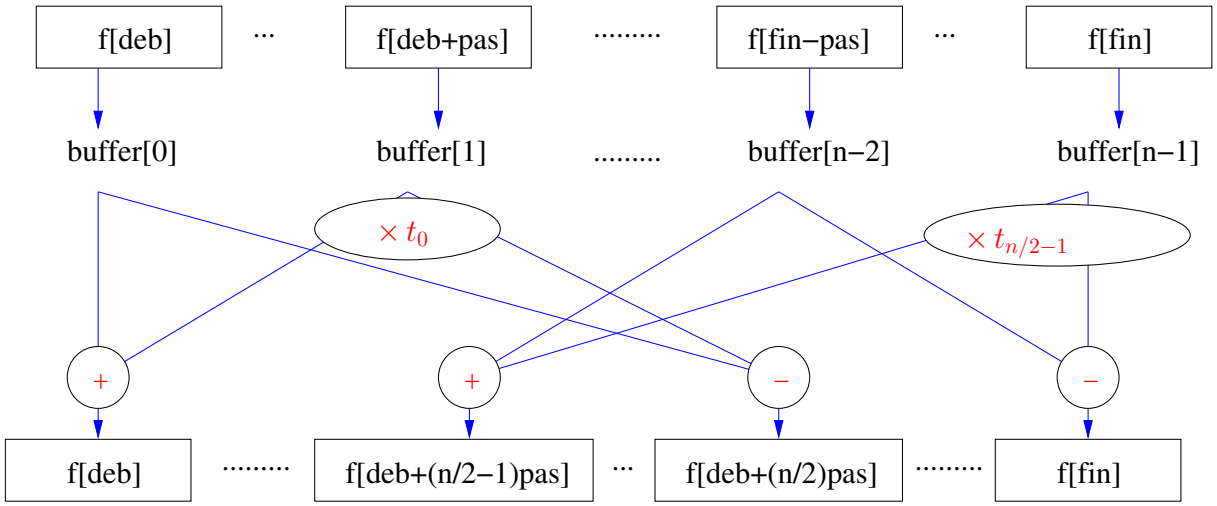


FIGURE A.4 – Etape de fusion après les deux sous-FFT : on recopie $f(\texttt{deb:pas:fin})$ dans $\texttt{buffer}(\texttt{0:1:n-1})$ pour ne pas s'emmêler dans la mise à jour. Les coefficients de `buffer` d'indice impair $2k+1$ sont multipliés par le twiddle $t_k = e^{s\frac{2i\pi}{n}k}$ avant addition ou soustraction au coefficient d'indice pair $2k$ précédent.

± 1. Elles allouent le buffer vu en cours pour le passer à cette fonction et éviter que celle-ci ne perde tu temps à réallouer pour chaque appel récursif. Implémenter ces deux fonctions, ne pas oublier de normaliser avec la fonction à cet effet à la fin.

4. *Cœur de la FFT*
   La fonction fft_main calcule la FFT sans normalisation du signal $f(\texttt{deb:pas:fin})$, dont elle écrit les résultats dans le même tableau. Il y a donc $n = (\texttt{fin-deb})/\texttt{pas}+1$ nombres.
   La fonction procède ainsi :
   (a) fft_main des indices pairs, donc commençant à `deb`.
   (b) fft_main des indices impairs, donc commençant à `deb+pas`.
   (c) Recopie des n éléments de f ainsi modifiée dans `buffer` (voir figure A.4) et mise à jour suivant l'équation ($0 \leq k < n/2$) :

$$f[\texttt{deb} + \texttt{k}*\texttt{pas}] \leftarrow \texttt{buffer}[2*\texttt{k}] + e^{s\frac{2i\pi}{n}k}\,\texttt{buffer}[2*\texttt{k}+1]$$
$$f[\texttt{deb} + \texttt{(k+n/2)}*\texttt{pas}] \leftarrow \texttt{buffer}[2*\texttt{k}] - e^{s\frac{2i\pi}{n}k}\,\texttt{buffer}[2*\texttt{k}+1].$$

5. *FFT 2D*
   La FFT 2-dimensionnelle se fait de façon séparable : faire d'abord la FFT sur les lignes, de longueur w, puis sur les colonnes, de longueur h. Appeler donc fft_main dans une boucle sur les lignes puis sur les colonnes. Normaliser à la fin en divisant par $\sqrt{w\,h}$.

Voici l'affichage du programme de test (en rouge les DFT, en bleu les signaux reconstruits après DFT inverse). Notez que les reconstructions ne sont pas parfaites, il reste un peu de partie imaginaire, de l'ordre de 10^{-6}, due à l'imprécision des nombres à virgule flottante.

```
(16,0) (15,0) (14,0) (13,0) (12,0) (11,0) (10,0) (9,0) (8,0) (7,0) (6,0) (5,0) (4,0) (3,0) (2,0) (1,0)
(34,0) (2,-10.0547) (2,-4.82843) (2,-2.99321) (2,-2) (2,-1.33635) (2.00001,-0.828429) (2,-0.397822) (2,3.09547e-06)
```

```
(2,0.397817) (2,0.828426) (2.00001,1.33637) (2,2.00001) (1.99999,2.99321) (2,4.82844) (1.99999,10.0547)
- 1D -
(34,0) (2,-10.0547) (2,-4.82843) (2,-2.99321) (2,-2) (2,-1.33636) (2,-0.828427) (2,-0.397824) (2,0) (2,0.397825)
(2,0.828427) (2,1.33636) (2,2) (2,2.99321) (2,4.82843) (2,10.0547)
(16,2.38419e-07) (15,-1.02917e-06) (14,2.38419e-07) (13,-6.99362e-07) (12,4.76837e-07) (11,1.62921e-07) (10,7.15256e-07)
(9,1.56561e-06) (8,-2.38419e-07) (7,4.01339e-07) (6,-2.38419e-07) (5,2.54312e-07) (4,-4.76837e-07) (3,1.62921e-07)
(2,-7.15256e-07) (1,-8.18572e-07)
- 2D -
(34,0) (2,-4.82843) (2,-2) (2,-0.828427) (2,0) (2,0.828427) (2,2) (2,4.82843) (16,0) (0,0) (0,0) (0,0) (0,0) (0,0) (0,0) (0,0)
(16,0) (15,1.03316e-07) (14,-5.96046e-08) (13,1.58933e-08) (12,0) (11,-1.58933e-08) (10,5.96046e-08) (9,-1.03316e-07) (8,0)
(7,1.03316e-07) (6,-5.96046e-08) (5,1.58933e-08) (4,0) (3,-1.58933e-08) (2,5.96046e-08) (1,-1.03316e-07)
```

A.3.2 Intermède : équation de Poisson

Nous allons utiliser la FFT pour appliquer des effets aux images. L'idée est de ne pas modifier directement l'image, mais de modifier son *gradient* en certains points, ce qui donne des effets bien plus subtils et naturels. Notons u l'image de dimension $w \times h$, et $V_x = \partial u/\partial x$ et $V_y = \partial u/\partial y$ ses dérivées partielles. On suppose celles-ci modifiées suivant différentes modalités détaillées ci-dessous, et on essaie de reconstruire u dont le gradient est prescrit ainsi. En dérivant encore une fois ces équations on se retrouve avec l'équation de Poisson, si courante en physique :

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \frac{\partial V_x}{\partial x} + \frac{\partial V_y}{\partial y}.$$

Rappelons la formule de dérivation du cours, transposée dans le cas 2D :

$$\hat{u}_x[k,l] = \begin{cases} \frac{2i\pi}{w} k\, \hat{u}[k,l] & \text{pour } 0 \le k < w/2 \\ 0 & \text{pour } k = w/2 \\ \frac{2i\pi}{w}(k-w)\, \hat{u}[k,l] & \text{pour } w/2 < k < w \end{cases} \tag{A.1}$$

et de façon analogue pour u_y. Si on passe en Fourier l'équation de Poisson, on va trouver :

$$\left(\left(\frac{2i\pi}{w}k\right)^2 + \left(\frac{2i\pi}{h}l\right)^2\right)\hat{u}[k,l] = \left(\frac{2i\pi}{w}k\right)\hat{V}_x[k,l] + \left(\frac{2i\pi}{h}l\right)\hat{V}_y[k,l], \tag{A.2}$$

avec k et l remplacés par $k-w$ et $l-h$ dans les facteurs multiplicatifs si $k > w/2$ ou $l > h/2$ comme ci-dessus. Notons que comme on peut s'y attendre, ces formules laissent $\hat{u}[0,0]$ indéterminé (constante d'intégration). Pour avoir une image affichable, on devra donc ramener les intensités de façon raisonnable dans la plage $[0, 255]$.

A.3.3 Partie 2 : Réhaussement du contraste

1. Regardez le rôle des fonctions qui vous sont données dans les fichiers `poisson.h` et `poisson.cpp`. Il n'est pas nécessaire de comprendre comment elles fonctionnent, mais savoir qu'elles existent et leur rôle.
   La fonction `affichable` transforme une image float en image byte en appliquant une transformation affine aux intensités de manière à ce que les valeurs

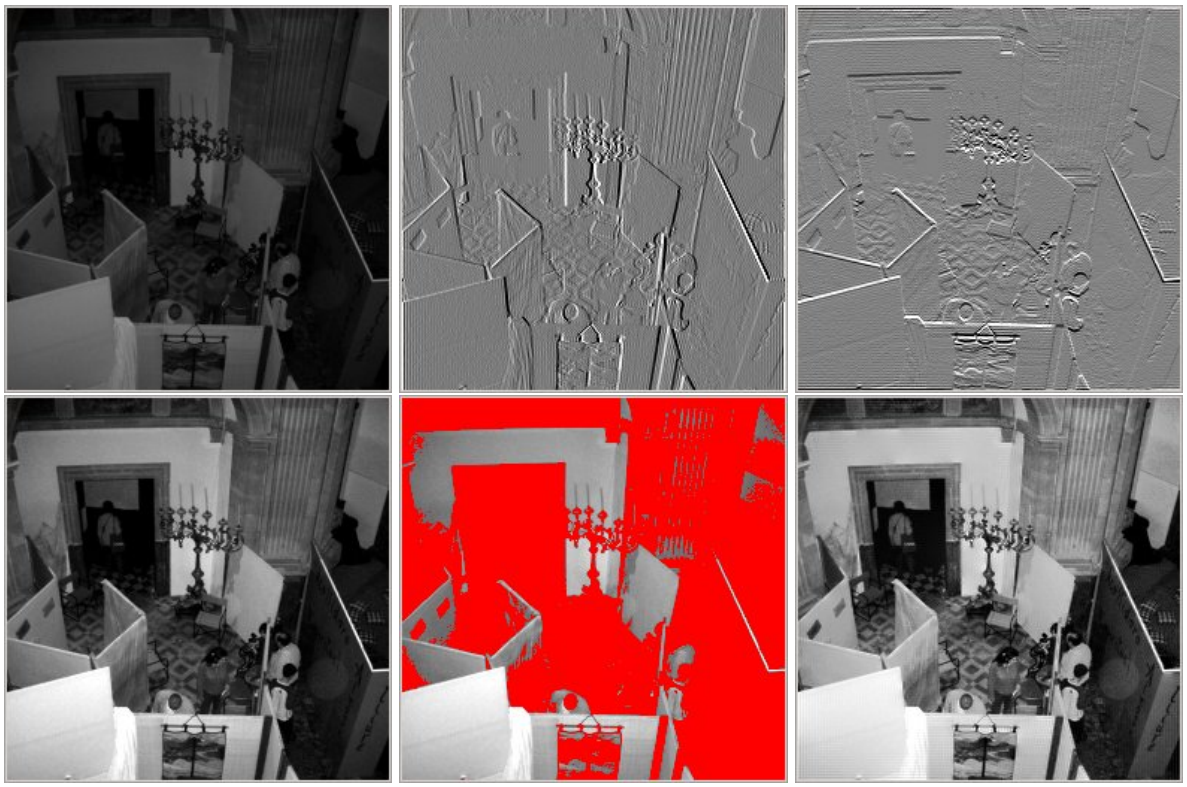


FIGURE A.5 – Image originale, dérivées en x et y, zones sombres en rouge, image avec ajustement affine du contraste, et résultat final (notez les détails qui n'étaient pas visibles sur le panneau à droite de l'image).

FIGURE A.6 – Exemples de réhaussement de contraste par Poisson.

les plus petites et les plus grandes (en ignorant les 0,5% de valeurs extrêmes) deviennent 0 et 255. C'est donc déjà un ajustement de contraste.
Notez qu'il est possible de passer directement une Image<byte> à une fonction attendant en paramètre une Image<float>, et l'un de ces deux types à une fonction attendant une Image<complex>.

2. *Gradient par différences finies*
Programmer la fonction `gradient` de `poisson.cpp` qui met dans `Vx` et `Vy` les images de dérivées partielles. Elles sont calculées par différences finies grâce à la fonction d'Imagine++ :

```
FVector<float,2> gradient(const Image<float>& I, Coords<2> pos);
```

(consulter la documentation d'Imagine++). En fait, nous utiliserons les dérivées calculées en Fourier pour la suite (plus précises), cette fonction ne servira donc qu'à vérifier.

3. *Gradient en Fourier*
Programmer les fonctions `Fourier_dx` et `dx`. La première fait la FFT, applique (A.1) et reste dans le domaine de Fourier. La deuxième appelle la première, repasse en domaine spatial et ignore les parties imaginaires résiduelles (fonction `realImage`). Attention, même si l'image `F` n'est pas passée en référence, la classe utilise une technique avancée pour ne pas faire de copie en profondeur. Pour être sûr de ne pas modifier l'image passée en paramètre, utiliser :

```
F = F.clone();
```

dans la fonction `dx` avant toute chose. Idem pour la dérivée suivant y.

4. *Visualisation du gradient*
Dans une fenêtre séparée, afficher successivement les dérivées calculées en différences finies et par Fourier séparées par un clic.

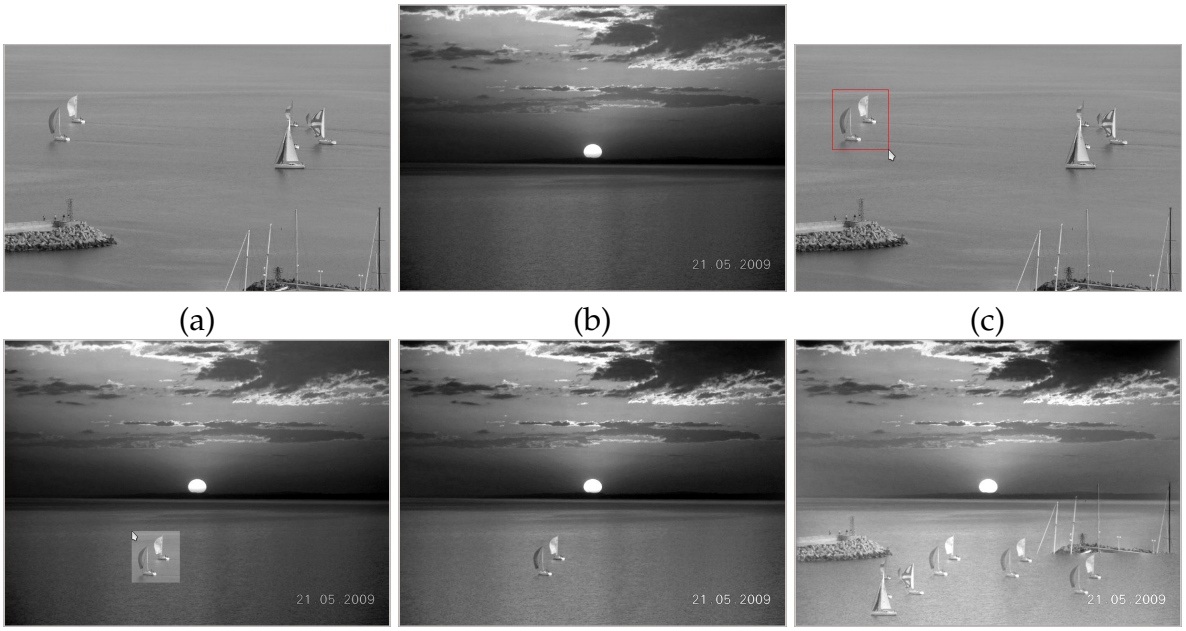
5. *Masque*
On va réhausser le contraste dans les zones sombres de l'image. Pour cela, compléter la fonction `masque` qui multiplie les dérivées par l'entier `FACTEUR` là où l'itensité de l'image est en-dessous de `OBSCUR`. Montrer en rouge ces pixels.

6. *Poisson*
Écrire la fonction `poisson`, qui utilise (A.2). Attention de mettre le coefficient $(0,0)$ à 0 (la formule vous ferait faire une division par 0). Pour le membre de droite de (A.2), on peut utiliser les fonctions `Fourier_dx` et `Fourier_dy`.

7. *Visualisation et mise en forme finale*
Afficher l'image avec zones sombres réhaussées. Notre FFT ne fonctionne que pour des dimensions puissances de 2. Modifier les fonctions `dx`, `dy` et `poisson` pour qu'elles agrandissent les images qui ne vérifient pas cette conditon (cf fonctions `puis2` et `agrandis`). À la fin, la méthode `getSubImage` sera utilisée. La fonction `agrandis` se charge d'appeler la fonction `clone` si nécessaire, vous pouvez donc supprimer votre propre appel à cette dernière.

A.3.4 Partie 3 : clonage

L'idée est de prendre des gradients d'une image et de les transférer dans une autre image, puis de reconstruire l'image (figure A.7).



(a) (b) (c)

(d) (e) (f)

FIGURE A.7 – Image de formes à extraire (a) et image de fond (b). On choisit dans (c) un rectangle à copier et le collage direct des intensités donne (d). Après clonage de Poisson, on obtient (e), où le collage est bien plus difficile à détecter. En choisissant des formes supplémentaires, on peut obtenir (f).

1. *Interface utilisateur*
   Utiliser les fonctions `selectionRect` et `cloneRect` pour extraire des rectangles de la première image et les placer dans la deuxième.
2. *Maximum du gradient*
   Compléter la fonction `maxGradient` qui met dans l'image 2 le gradient de plus grande norme entre les deux images. Ceci se fait dans le rectangle de coins (x_1, y_1) et (x_2, y_2) qu'on place en (x_c, y_c) dans l'image 2.
3. *Clonage*
   Utiliser la fonction `poisson` pour reconstruire l'image éditée. Arrangez votre programme pour qu'on puisse cloner plusieurs zones de l'image 1 dans l'image 2, jusqu'à un clic droit qui termine le programme.

N'hésitez pas à mettre vos œuvres dans l'archive que vous remettrez en testant avec vos propres images. Cela peut se faire sans recompiler le programme en utilisant les arguments passés au programme. Dans QtCreator, "Projets", "Run", "Arguments". Si vous ne mettez pas un chemin absolu (juste le nom de fichier image), mettez à jour le dossier de lancement.

A.4 Fast Marching

Ce long TP nous occupera plusieurs séances. La première partie sera consacrée à l'implémentation de la file de priorité et nous l'utiliserons pour la résolution de l'équation eïkonale, ce qui nous servira à calculer des cartes de distances sur des surfaces non homogènes.

A.4.1 Partie 1 : File de priorité

Pour stocker l'arbre binaire équilibré qui nous permet d'implémenter la file de priorité, il est très efficace d'utiliser un simple tableau. Pour une fois, nous considérerons que le premier élément du tableau est d'indice 1 et non 0. Le C++ n'autorise pas cela, mais plutôt que de jouer sur les indices, il est plus facile d'ignorer simplement l'indice 0 du tableau. La racine est donc en 1, ses enfants en 2 et 3, les enfants de 2 en 4 et 5 et ceux de 3 en 6 et 7, etc. Les enfants de l'élément d'indice i sont en $2i$ et $2i+1$. Son parent en indice $i/2$ (quotient de la division euclidienne). La façon d'effectuer les opérations `push` et `pop` est rappelée figure A.8.

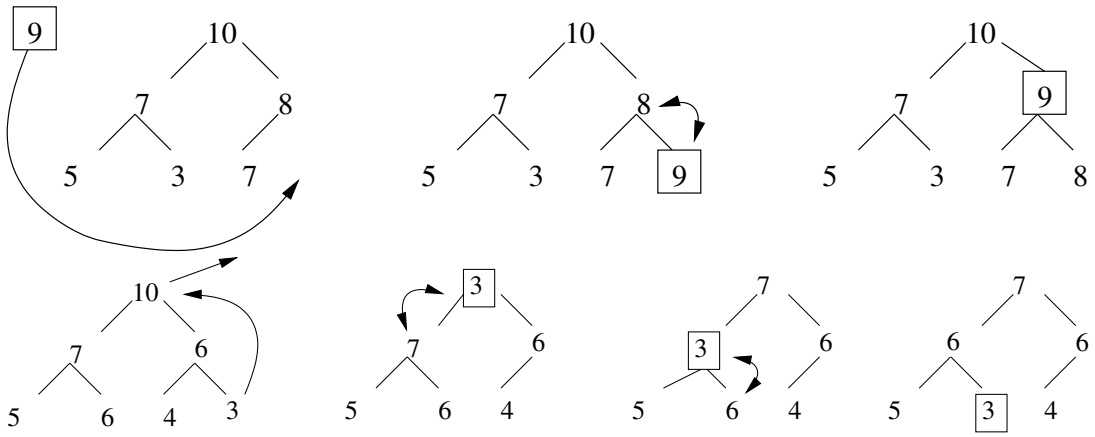


FIGURE A.8 – Procédures `push` (haut) et `pop` (bas) de la file de priorité codée avec un arbre binaire.

1. *Pour commencer, étudier le projet :*
   Télécharger le fichier `FastMarching_Initial.zip` sur la page habituelle, le décompresser.
2. *Mise en place :*
   Écrire le constructeur et la méthode `empty`.
3. *Implémentation*
   Implémenter les fonctions `push` et `pop`, suivant le principe de la figure A.8. La priorité est donnée par le champ `dist` de `PointDist`. Attention, seule la comparaison $<$ est codée, donc votre code ne doit pas utiliser les opérateurs analogues, comme $>, <=, >=$, à moins que vous ne les définissiez aussi. Pour intervertir deux variables a et b, utilisez std :: swap(a,b).
4. Test
   Dans `test_priorite.cpp`, mélanger $n = 20$ objets `PointDist` de différentes priorités $0, 1, \ldots, n-1$, les ajouter à une file de priorité, puis dépiler jusqu'au épuisement en affichant les priorités. Vous devriez trouver $n-1, n-2, \ldots, 0$.

A.4.2 Intermède : équation eikonale

Algorithme de Dijkstra avec poids sur les nœuds

On utilisera un graphe dont les nœuds sont les pixels d'une image et les arêtes les relient à leurs 4 voisins (moins de 4 sur les bords). Lorsque les poids sont affectés aux nœuds et non aux arêtes, l'algorithme de Dijkstra ne s'applique pas. Mais on peut s'y ramener par un graphe orienté avec poids différents pour les arêtes réciproques : $w_{(p,q)} = w_q$, et donc $w_{(q,p)} = w_p$. Comme toutes les arêtes arrivant en un même point ont le même poids, on ne retrouve pas dans la situation où un point a déjà une distance attribuée et on trouve plus tard un chemin plus court. Ainsi, il suffit de se souvenir quels pixels ont déjà été visités. On peut tester si $D_p = \infty$, ou bien en garder trace explicitement avec une table E. Cela donne :

1. Initialiser toutes les distances D_p à l'infini, et un état E_p à false (non visité).
2. Mettre les points de départ p dans une file de priorité F avec priorité 0, fixer leur distance $D_p = 0$ et leur état E_p =true.
3. p=F.pop(), et pour tous les voisins q de p *non visités* (E_q ==false) :
   (a) $D_q = w_q + \min_{p' \sim q} D_{p'}$ (notez que p participe dans ce minimum).
   (b) $E_q \leftarrow$true (point visité)
   (c) F.push(q,$-D_q$).
4. retourner en 3 si F n'est pas vide.

Cet algorithme, bien qu'exact, ne fonctionne pas pour une métrique non adaptée à un graphe : par exemple, si on place les nœuds sur les points de $\mathbb{Z}^2$, les points $(0,0)$ et $(1,1)$ sont calculés à distance 2 car les plus courts chemins passent par $(1,0)$ ou $(0,1)$. Or pour la distance euclidienne, ils sont à distance $\sqrt{2}$ et le plus court chemin ne passe *pas* par des points de la grille.

Équation eikonale

Il s'agit de l'équation aux dérivées partielles ($D : \mathbb{R}^2 \to \mathbb{R}^+$) :

$$|\nabla D| = W, \quad D(S) = \{0\}$$

avec $S \subset \mathbb{R}^2$ et $W > 0$. S est l'ensemble des points de départ, à distance $D = 0$, et W représente le coût d'entrée en un point. On peut mettre éventuellement une valeur $+\infty$ pour empêcher l'accès à un point.

Par exemple, si on fixe $W = 1$ partout, la solution est la fonction $D(x) = \mathrm{dist}(x, S)$.

Fast marching

Il s'agit d'une solution approchée sur une grille de l'équation eikonale faisant intervenir une modification simple de l'algorithme de Dijkstra. Pour cela, il faut utiliser un bon schéma numérique pour le gradient :

$$(|\nabla D|)^2_{i,j} = \max(D_{i,j} - D_{i,j+1}, D_{i,j} - D_{i,j-1}, 0)^2 + \max(D_{i,j} - D_{i+1,j}, D_{i,j} - D_{i-1,j}, 0)^2.$$

Si on cherche à mettre à jour $D_{i,j}$ connaissant D aux quatre voisins de (i,j), on obtient :

$$\begin{cases} D_{i,j} = \min(d_1, d_2) + W_{i,j} & \text{si } d_1 = +\infty \text{ ou } d_2 = +\infty \\ (D_{i,j} - d_1)^2 + (D_{i,j} - d_2)^2 = W_{i,j}^2 & \text{sinon,} \end{cases} \tag{A.3}$$

avec $d_1 = \min(D_{i,j+1}, D_{i,j-1})$ et $d_2 = \min(D_{i+1,j}, D_{i-1,j})$. Le deuxième cas de l'alternative donne une équation polynômiale de degré 2 dont la solution est :

$$D_{i,j} = \frac{d_1 + d_2 + \sqrt{2W_{i,j}^2 - (d_1 - d_2)^2}}{2}, \tag{A.4}$$

l'autre racine conduisant à une valeur inférieure à d_1 ou d_2. Si le discriminant (sous la racine carrée) est négatif, on applique la formule du premier cas. Le fast marching remplace donc simplement l'étape 3a de l'algorithme de Dijkstra par (A.4) dans le cas où cette formule s'applique (discriminant positif).

Dans les programmes interactifs ci-dessous, guidez l'utilisateur sur ce qui est attendu de lui par des std :: cout.

A.4.3 Partie 2 : carte de distance

1. Complètez la fonction `fastMarching`. Elle fait appel à `calcDistance` appliquant (A.3), qui elle-même appelle `minVoisins`, également à complèter.
2. Dans `distancePoint.cpp`, faites entrer à l'utilisateur des points de départ, puis afficher la carte de distance à ces points. Utilisez la fonction `affiche` pour montrer l'image des distances en couleurs (bleu=0, rouge=maximum).

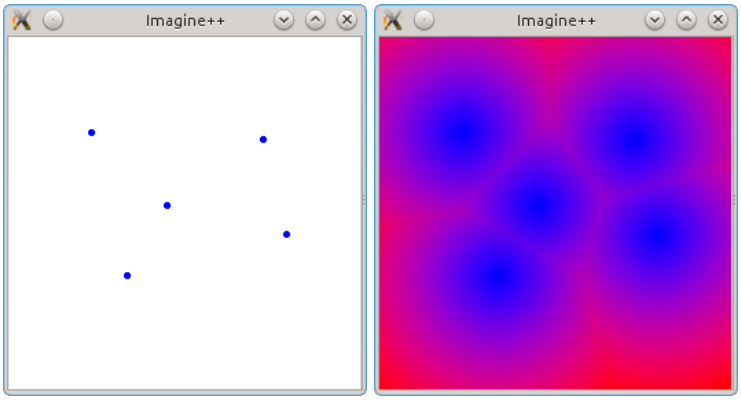


FIGURE A.9 – Points de départ et carte de distance à ces points dans le programme `distancePoint`

A.4.4 Partie 3 : ombres

3. Dans le programme `ombres.cpp`, faites dessiner à l'utilisateur des rectangles représentant des objets opaques (utilisez la fonction `selectionRect`). Remplir dans l'image I les pixels du rectangle en rouge.
4. Faites cliquer à l'utilisateur une position de source lumineuse, assurez-vous que ce ne soit pas un point rouge.
5. Remplissez la carte de coût `Wavec` avec des valeurs infinies là où l'image I est rouge. Un `float` peut prendre la valeur infinie, définie par la constante `INF` dans `fastMarching.h`. Les calculs impliquant des valeurs infinies se comportent comme on peut s'y attendre ; les opérations mal définies, comme $\infty - \infty$ et $0 * \infty$ donnent un NaN ("not a number").

6. L'idée est de calculer les ombres en calculant la carte des distances `Davec` comportant les obstacles et `Dsans` en l'absence d'obstacles. Là où `Davec` excède `Dsans` (avec une tolérance `TOL`), marquer une ombre en bleu.
7. Affichez les cartes de distance avec/sans obstacles dans des fenêtres séparées.

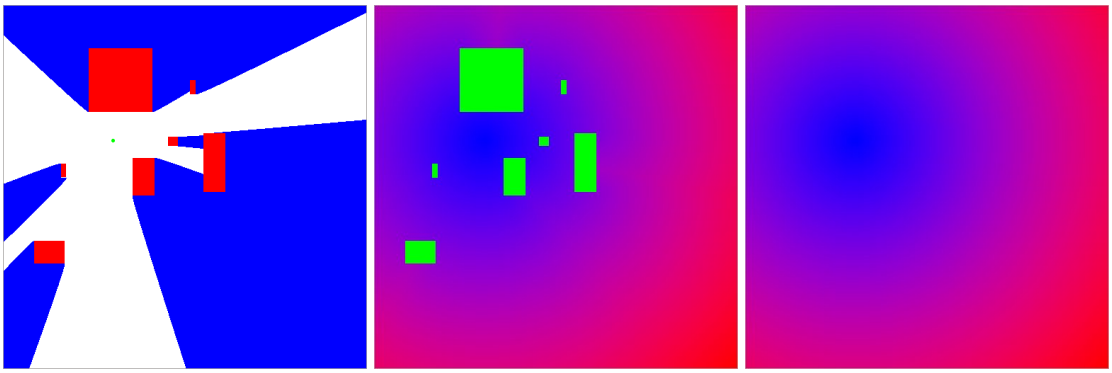


FIGURE A.10 – Ombres portées depuis la source lumineuse placée au point vert, cartes de distance avec et sans obstacles, dont la comparaison a permis le calcul des ombres.

A.4.5 Partie 4 : Géodésique

On va chercher un plus court chemin en suivant au mieux la couleur du pixel de départ.

8. Le point de départ p étant désigné par l'utilisateur, la carte de coût se calcule suivant la formule
$$W(q) = \epsilon + \|I(q) - I(p)\|_1/3$$
avec la constante $\epsilon = 10$ (règle la rectitude de la géodésique). Cette fonction croît avec la différence de couleur, donc encourage à rester à la couleur du pixel de départ.
9. Pour éviter des risques de sortie de l'image, mettre les $W(q)$ à $+\infty$ quand q est au bord de l'image.
10. La fonction `geodesique` retourne un vecteur donnant le plus court chemin entre les points p_1 et p_2 : on part de p_2 et on descend dans la direction du gradient :
$$q_{i+1} = q_i - \frac{\tau}{|\nabla D|}\nabla D.$$
On arrête dès que $|q_i - p_1| < 1$, et on ajoute p_1. Cependant, on restreindra la géodésique à un nombre maximum de points. Au cas où on n'arrive pas à proximité de p_1 au bout de ce nombre de points, on affiche un message d'avertissement à l'utilisateur.
11. Faites cliquer à l'utilisateur des points de départ et d'arrivée sur la carte des coûts, et tracez le chemin géodésique.

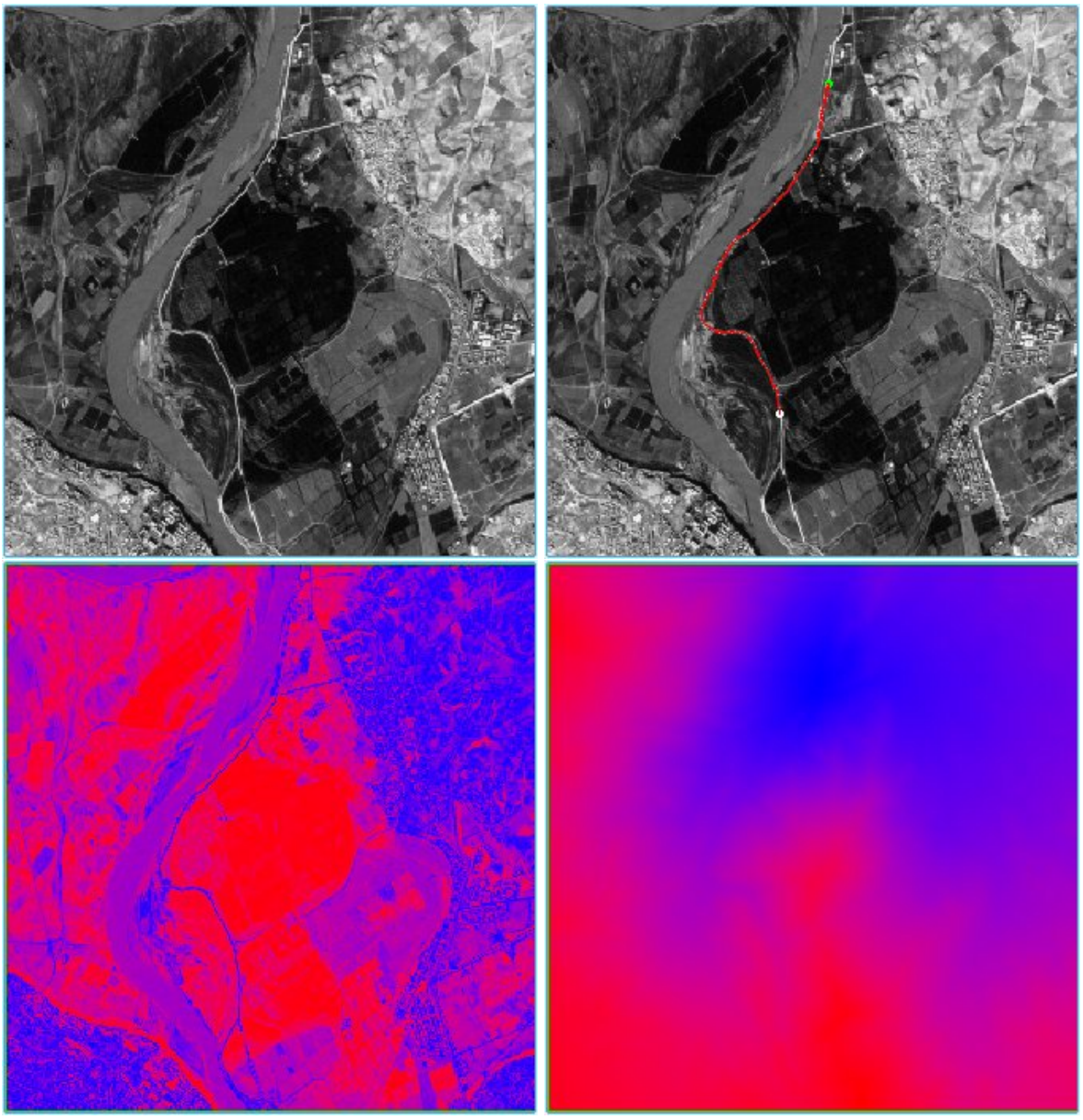


FIGURE A.11 – Tracé de géodésique : image initiale, géodésique vers le point blanc, image de coût par rapport au point de départ (en vert), image de distance à ce point.