Séquence 16 - Techniques algorithmiques - Bilan

Contents

Vocabulaire à connaître: algorithme exhaustif, *backtracing*, paradigme glouton, paradigme diviser pour régner, mémoïsation, approche top-down, approche bottom-up, valeur optimale, solution optimale, programmation dynamique.

Nous avons vu, tout au long de l'année, de nombreux algorithmiques. On peut, dans une certaine mesure, trouver des similitudes entre certains de ces algorithmes, qui agissent selon une même philosophie

I. Méthodes exhaustives

Ce sont souvent les méthodes les plus naïves. Elles consistent à énumérer et tester une à une toutes les possibilités de solution, en espérant tomber rapidement sur l'une d'entre elles si l'on cherche simplement une solution, et en allant au bout de cette énumération si on les veut toutes.
Nous avons vu plusieurs algorithmes de ce type au cours de l'année:

- calcul de la table de vérité d'une formule logique pour en étudier la satisfiabilité et rechercher ses modèles;
- la recherche séquentielle peut être considérée comme une méthode exhaustive;
- recherche naïve de motif dans un texte.

La complexité temporelle de ces algorithmes est généralement prohibitive. Ils ne peuvent être utilisés que sur des données de petites tailles, lorsque l'on souhaite une solution simple et rapide à implémenter.

II. Paradigme backtracing

Les algorithmes de type *backtracing* (retour sur trace) vont tenter différentes progressions vers la solution. Lorsque qu'une piste se révèle être une impasse, l'algorithme est capable de revenir en arrière (d'où le terme de *backtracing*) pour retrouver et poursuivre une autre piste.
Nous avons vu plusieurs algorithmes de ce type au cours de l'année:

- algorithme de parcours en profondeur (DFS)
- algorithme de résolution de labyrinthe (cf exercice de colle 1er semestre)
- algorithme de Quine pour la résolution de la satisfiabilité d'une formule logique

Ces algorithmes peuvent être écrits sous forme récursive, l'arrivée dans une impasse correspondant à un cas d'arrêt de l'imbrication récursive, ou sous forme itérative, souvent à l'aide de piles.
Souvent, ces algorithmes sont coûteux. Comme ils tentent tous les chemins, une implémentation *backtracing* naïve peut s'avérer proche, en terme de complexité, de l'approche exhaustive. Toutefois, ces algorithmes peuvent souvent être améliorés par des heuristiques bien choisies, qui éliminent prématurément des chemins qui sont des impasses évidentes (pensez aux améliorations issues de l'algorithme de Quine). Et ils sont également, assez souvent, les seuls disponibles!

Exercice 1 (Exercice de recherche pour les vacances).
Implémenter un solveur récursif de Sudoku par une méthode de *backtracing*.

III. Paradigme glouton

Les algorithmes dits gloutons (*greedy algorithm* en anglais) sont des algorithmes qui, à l'inverse des algorithmes de *backtracing*, n'autorisent aucun retour en arrière. On choisit, à chaque étape de l'algorithme, d'avancer dans une direction qui optimise localement la solution, sans se préoccuper de savoir si ce choix qui est localement systématiquement avantageux va conduire à une solution globalement avantageuse.

III. 1. Coloriage de graphes

Nous avons évoqué dans la séquence sur la logique le problème de k-coloriage d'un graphe. Ce problème consiste à se demander si un graphe peut être colorié avec k couleurs sans que deux sommets adjacents n'aient la même couleur. Nous avons vu que ce problème peut faire l'objet d'une modélisation SAT, qui devient de plus en plus difficile à résoudre au fur et à mesure que la taille du graphe augmente (augmentation très rapide du nombre de clauses de la modélisation SAT)
Il est toutefois très facile de proposer des coloriages sans que deux sommets adjacents n'aient la même couleur, pour peu que l'on n'impose pas a priori un nombre de couleurs

k. Cela peut se faire en écrivant un algorithme glouton assez naïf.

Définition 1 (Coloriage d'un graphe)

Étant donné un graphe non orienté et non pondéré $G = (V, E)$, on appelle *k-coloration* de G, $k \in \mathbb{N}^*$, une application c qui associe à chacun des sommets de G un entier de $[0, k-1]$ de sorte que si deux sommets u et v sont voisins alors leurs couleurs sont différentes.

$$\forall (u, v) \in E \qquad c(u) \neq c(v)$$

Le problème de la *coloration d'un graphe* consiste à trouver une k-coloration pour laquelle la valeur de k est minimale.

Considérons le graphe ci-dessous et un premier ordre de parcours des sommets : $v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7$. Un algorithme glouton affecte une première couleur, par exemple l'entier 0, au sommet

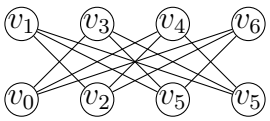


Figure 1: Coloration d'un graphe

v_0 : $c(v_0) = 0$. Puis le sommet v_1 est visité. N'étant pas adjacent au sommet v_0, il peut être colorié avec la même couleur : $c(v_1) = 0$. Vient alors la coloration de v_2, sommet adjacent à v_1. Il ne peut pas être colorié avec la couleur 0; l'algorithme glouton alors lui attribue la plus petite couleur différente de celle de ses voisins, en l'occurrence 1 puisque le seul sommet déjà colorié est v_1 : $c(v_2) = 1$. Une même analyse mène à la coloration de v_3 : $c(v_3) = 1$. Le sommet v_4 est adjacent à v_1 et v_3, sommets déjà coloriés. L'algorithme lui attribue la couleur $c(v_4) = 2$. Et ainsi de suite avec tous les autres sommets. Au final, la coloration obtenue en suivant l'ordre choisi est :

$$c(v_0) = 0 \quad c(v_2) = 1 \quad c(v_4) = 2 \quad c(v_6) = 3$$

$$c(v_1) = 0 \quad c(v_3) = 1 \quad c(v_5) = 2 \quad c(v_7) = 3$$

Il s'agit d'une 4-coloration. Pourtant, une rapide observation du graphe montre que ce dernier est biparti. L'ensemble V de ses sommets peut être partitionné en deux sous-ensembles disjoints $V_1 = \{v_1, v_3, v_5, v_7\}$ et $V_2 = \{v_0, v_2, v_4, v_6\}$. En conséquence, une 2-coloration est possible sous la forme :

$$c(v_0) = 0 \quad c(v_2) = 0 \quad c(v_4) = 0 \quad c(v_6) = 0$$

$$c(v_1) = 1 \quad c(v_3) = 1 \quad c(v_5) = 1 \quad c(v_7) = 1$$

Ces deux situations illustrent bien le fait que l'algorithme peut trouver une solution optimale, comme par exemple la 4-coloration, mais pas nécessairement la meilleure, comme la 2-coloration. Il ne trouve pas la pire solution, à savoir une 8-coloration, où chaque sommet se verrait attribuée une couleur différente de celle de tous les autres sommets.
Voici une implémentation de cet algorithme dans laquelle les sommets sont traités dans leur ordre d'étiquetage comme dans le premier cas évoqué ci-dessus. On crée d'abord une fonction `first_free` qui prend en entrée la liste `l` des couleurs des successeurs d'un sommet u et qui renvoie le plus petit numéro de couleur qui peut être utilisé pour le sommet u, en éliminant toutes les couleurs non autorisées car déjà attribuées à un sommet successeur de `l`.

```
# let first_free l nv =
  let free = Array.make nv true in
  (List.iter (fun c -> if (c >= 0) then (free.(c) <- false)) l;
   let rec find_smallest_free free i = if (free.(i) = true) then i else find_smallest_free free (i+1) in
   find_smallest_free free 0
  );;
val first_free : int list -> int -> int = <fun>
```

Ensuite, l'algorithme glouton parcours chaque sommet, dans l'ordre d'étiquetage, et attribue à chaque sommet la première couleur possible en appelant cette fonction pour éliminer les couleurs non autorisées:

```
# let greedy_color g =

  let nv = Wgraph.number_of_vertices g in
  let color_tab = Array.make nv (-1) in
  let nb_colors_max = ref (-1) in
  for u = 0 to (nv-1) do
    let lsucc = Wgraph.succ g u in
    let list_neighbours_colors = List.map (fun (v, _) -> color_tab.(v)) lsucc in
    let smallest_available_color = first_free list_neighbours_colors nv in
    (
      color_tab.(u) <- smallest_available_color;
      if ( smallest_available_color > !nb_colors_max ) then
        nb_colors_max := smallest_available_color
    )
  done;
  (!nb_colors_max, color_tab);;
val greedy_color : 'a Wgraph.t -> int * int array = <fun>
```

Remarque (Quel ordre ?). La détermination des couleurs dépend étroitement de l'ordre dans lequel les sommets d'un graphe sont visités. On peut alors s'interroger sur l'existence d'un ordre particulier qui fournirait la coloration optimale. Malheureusement, ce problème est NP-difficile en tant que sous-problème de la coloration optimale d'un graphe, lui-même problème NP-complet [a].

[a] Schématiquement, un problème algorithmique entre dans la *classe* P s'il existe un algorithme de complexité polynomiale qui le résout. Il entre dans la *classe* NP si on ne peut seulement que vérifier la complexité polynomiale d'une solution candidate. Un problème est dit NP-difficile si tout problème de la classe NP peut s'y ramener via une transformation appelée de *réduction polynomiale*. Si en outre, le problème lui-même est NP, on le qualifie alors de NP-complet. L'une des questions fondamentales actuelles de l'informatique est de savoir si P et NP sont une seule et même classe de complexité.

On peut toutefois proposer une implémentation de cet algorithme glouton dans laquelle l'utilisateur peut déterminer l'ordre de traitement des sommets pour l'attribution des couleurs. Dans cette version, l'ordre de visite des différents sommets est transmis grâce à une liste `order`, qui donne la liste des étiquettes dans l'ordre dans lequel on souhaite qu'elles soient traitées:

```
# let greedy_color g order  =

  let nv = Wgraph.number_of_vertices g in
  let color_tab = Array.make nv (-1) in
  let nb_colors_max = ref (-1) in
  let set_color u =
    let lsucc = Wgraph.succ g u in
    let list_neighbours_colors = List.map (fun (v, _) -> color_tab.(v)) lsucc in
    let smallest_available_color = first_free list_neighbours_colors nv in
    (
      color_tab.(u) <- smallest_available_color;
      if ( smallest_available_color > !nb_colors_max ) then
        nb_colors_max := smallest_available_color
    )
  in
  List.iter set_color order;
  (!nb_colors_max, color_tab);;
val greedy_color : 'a Wgraph.t -> int list -> int * int array = <fun>
```

Remarque. Coloration des arêtes Notre exemple présente la *coloration des sommets* d'un graphe. Mais il existe un autre type de coloration appelé *coloration des arêtes*. Cette dernière attribue une couleur à chaque arête d'un graphe en évitant que deux arêtes adjacentes aient la même couleur.

Remarque (Théorème de Brooks). Nous indiquons ci-dessus qu'une 8-coloration est la pire coloration du graphe qui soit. Ce qui constitue une borne supérieure pour la coloration d'un graphe. Toutefois, le théorème de Brooks montre qu'il est possible de faire mieux. Ce théorème précise qu'un graphe connexe non orienté de degré maximal Δ peut être colorié avec au plus Δ couleurs excepté s'il est complet ou est un graphe cyclique de longueur impaire, auxquels cas $1 + \Delta$ couleurs exactement sont nécessaires. Une démonstration constructive de ce résultat peut être établie à l'aide d'un algorithme glouton de coloration.

III. 2. Rendu de monnaie naïf

Considérons un système monétaire constitué de n unités monétaires (billets ou pièces de monnaie) de valeurs entières $v_1 < v_2 < \cdots < v_n$, avec $v_1 = 1$. Pour illustrer notre propos, adoptons le système en vigueur dans la zone euro en omettant les centimes : 1€, 2€, 5€, 10€, 20€, 50€, 100€, 200€, 500€. Le problème du *rendu de monnaie* consiste à rendre un **nombre minimal de billets et de pièces pour une somme** S donnée. Par exemple, la somme de 49€ peut être rendue avec 49 pièces de 1€ mais aussi avec 2 billets de 20€, 1 billet de 5€ et 2 pièces de 2€. Ce qui représente un maximum de 5 billets et pièces rendus au lieu des 49 pièces. Ce nombre 5 est d'ailleurs le plus petit nombre de billets et de pièces rendus.

Il s'agit là encore d'un problème d'optimisation et même si elle ne fournit pas toujours la meilleure solution possible, une **stratégie gloutonne** est souvent adoptée en raison de sa simplicité de mise en œuvre. C'est d'ailleurs la stratégie de rendu que vous, moi et tous les européens utilisons chaque jour.
On choisit d'abord les billets qui permettent de rendre la plus grande valeur possible sur la somme à rendre. Dans notre exemple, on choisit d'abord 2 billets de 20€. Il reste alors à rendre 9€. On choisit 1 billet de 5€, plus grande valeur inférieure à 9€ qui peut être

rendue. Il reste enfin 4€ qui peuvent être rendues avec 2 pièces de 2€.

```
# let print_change change monetary_system =
  let n = Array.length monetary_system in
  for i = 0 to (n-1) do
    if (change.(i)  > 0) then
      Printf.printf "\n%d entité(s) de valeur %d" change.(i) monetary_system.(i)
  done;
  Printf.printf "\n";;
val print_change : int array -> int array -> unit = <fun>
# let change_greedy amount monetary_system =

  assert(monetary_system.(0) = 1);

  let n = Array.length monetary_system in
  let change = Array.make n 0 in (* tableau qui contiendra, pour chaque unité monétaire,
                                    le nombre de ces unités à rendre *)
  let remain = ref amount in (* somme restant à rendre *)
  let i = ref (n-1) in
  let nb_change = ref 0 in

  while (!remain >= monetary_system.(0)) do
    if (monetary_system.(!i) > !remain ) then
      i := !i - 1
    else
      (
        change.(!i) <- change.(!i) + 1;
        nb_change := !nb_change + 1;
        remain := !remain - monetary_system.(!i);
        Printf.printf "Unité de valeur %d choisie:
                       somme restante à rendre = %d\n" monetary_system.(!i) !remain;
      )
  done;
  print_change change monetary_system; (* affichage du rendu *)
  !nb_change;; (* nombre d'unités (pieces) à rendre *)
val change_greedy : int -> int array -> int = <fun>
# let euro_system = [| 1; 2; 5; 10; 20; 50; 100; 200; 500|];;
val euro_system : int array = [|1; 2; 5; 10; 20; 50; 100; 200; 500|]
# change_greedy 13 euro_system;;
Unité de valeur 10 choisie:  somme restante à rendre = 3
Unité de valeur 2 choisie:  somme restante à rendre = 1
Unité de valeur 1 choisie:  somme restante à rendre = 0
1 entité(s) de valeur 1
1 entité(s) de valeur 2
1 entité(s) de valeur 10
- : int = 3
```

En pratique, **cette stratégie gloutonne donne la solution optimale avec le système monétaire de la zone Euro**. À ce titre, le système monétaire de la zone Euro est qualifié de *canonique*.

Pour d'autres systèmes monétaires, elle ne fournit pas la solution optimale globale. Par exemple, le système monétaire en vigueur en la Grande Bretagne jusqu'en 1971 utilisait les valeurs monétaires 1, 3, 6, 12, 24, 30. Pour rendre une valeur monétaire de 49, la stratégie gloutonne choisit 30, 12, 6 et 1 alors que 2 unités de valeur 24 et 1 unité de 1 (3 unités au total) suffisent. Ce système monétaire n'est pas canonique. Depuis, la Grande Bretagne a adopté un système canonique en Livres Sterlings.[1]

```
# let uk_system_until_1971 = [| 1; 3; 6; 12; 24; 30|];;
val uk_system_until_1971 : int array = [|1; 3; 6; 12; 24; 30|]
# change_greedy 49 uk_system_until_1971;;
Unité de valeur 30 choisie:  somme restante à rendre = 19
Unité de valeur 12 choisie:  somme restante à rendre = 7
Unité de valeur 6 choisie:  somme restante à rendre = 1
Unité de valeur 1 choisie:  somme restante à rendre = 0

1 entité(s) de valeur 1
1 entité(s) de valeur 6
1 entité(s) de valeur 12
1 entité(s) de valeur 30
- : int = 4 (* non optimal!! On peut rendre 3 pièces: 2*24 + 1 *)
```

[1] https://fr.wikipedia.org/wiki/Decimal_Day

Remarque (Une alternative). Le problème du rendu de monnaie peut se décomposer en sous-problèmes dépendants. Il se prête naturellement à un traitement par *programmation dynamique* que nous allons voir en TD.

III. 3. Autres algorithmes gloutons vus cette année

Nous avons vu ou allons voir d'autres algorithmes gloutons: la création de l'arbre dans l'algorithme de Huffman, l'algorithme de remplissage du sac à dos...

Attention (A retenir). **Le problème des algorithmes gloutons est qu'ils ne renvoient pas toujours la solution globalement optimale**, car il procèdent en choisissant systématique une solution optimale localement, ce qui ne garantit nullement que la solution ainsi obtenue soit optimale globalement. Toutefois, lorsque l'on peut démontrer qu'un algorithme glouton renvoie la solution optimale, ces algorithmes sont souvent performants!

IV. Paradigme Diviser pour régner

L'approche diviser pour régner, DPR, s'applique à des problèmes que l'on peut découper en sous-problèmes similaires **indépendants**.

IV. 1. Principe

Les algorithmes construits sur ce paradigme consistent à découper le problème en plusieurs sous-problèmes **indépendants**, puis à rassembler les deux solutions indépendantes pour reconstruire une solution globale (le fameux coût de recomposition)
En général, les étapes suivantes apparaissent clairement dans ces algorithmes:

1. découpage du problème en b sous-problèmes indépendants
2. résolution de a sous-problèmes indépendants de même nature de taille $\left\lceil \frac{n}{b} \right\rceil$ (à 1 près)
3. recomposition des résultats pour reconstituer une solution globale

Nous avons vu que, lorsqu'ils sont écrits sous forme itératives, on peut exprimer la complexité temporelle de tels algorithmes sous la forme d'une relation de récurrence du type:

$$T(n) = aT(\left\lceil \frac{n}{b} \right\rceil) + f(n)$$

où la fonction f représente le coût temporel de la recomposition.
Nous avons vu de nombreux exemples d'algorithmes DPR:

- algorithme de recherche dichotomique dans un tableau
- algorithme d'exponentiation rapide
- algorithme de Karatsuba pour la multiplication de polynômes
- recherche des racines d'une fonction numérique monotone sur un intervalle $[a, b]$

La formulation récursive est souvent assez naturelle dans le cadre du paradigme DPR, mais elle n'est absolument pas obligatoire.

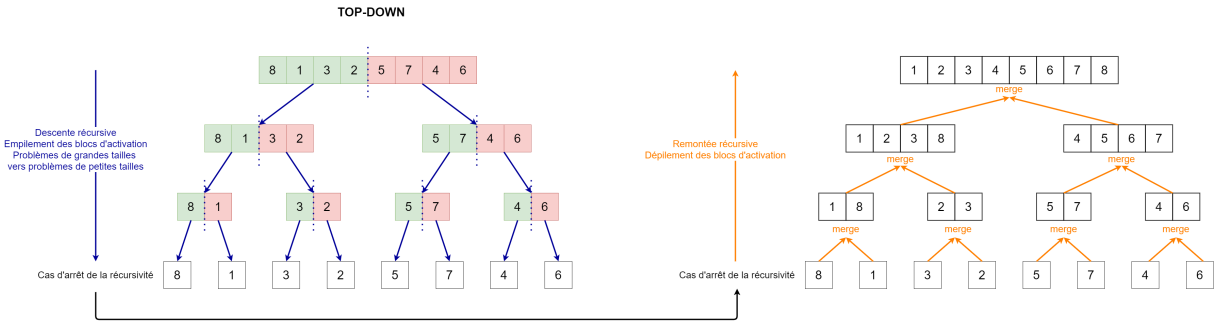
IV. 2. De la formulation récursive à la formulation itérative: visions top-down vers bottom-up

Nous avons vu, tout au long de l'année, plusieurs exemples où il nous a été possible de réécrire des algorithmes naturellement récursifs en algorithme itératif, ce qui permet d'améliorer la complexité spatiale dans le segment de pile. Pour cela, nous avons dû changer de point de vue. Je vous invite à aller revoir comment, par exemple, nous avons pu proposer des versions itératives de l'exponentiation rapide ou de la mutliplication russe, avec des stratégies d'accumulation.

Pour illustrer à nouveau ce concept, dans la perspective de la programmation dynamique, nous allons étudier le tri fusion. Le tri fusion est un exemple typique du paradigme diviser pour régner.

IV. 2. a. Formulation récursive top-down

La façon la plus naturelle de voir le tri fusion est une **vision descendante**, appelée *top-down* en anglais: l'algorithme procède en triant indépendamment la moitié gauche, puis la moitié droite du tableau, et fusionne ensuite les deux tableaux pour reconstituer un tableau trié. On part donc du grand tableau, que l'on sépare en deux parties. On envisage alors récursivement les deux sous-problèmes de tri des deux moitiés de tableaux...etc On a ainsi une résolution en cascade de ces sous-problèmes, du tableau de grande taille vers les tableaux de petite taille, le cas d'arrêt étant le tableau de taille 1, facile à trier. Lorsque ce cas d'arrêt est atteint, les appels récursifs vont successivement terminer et on va remonter la pile des appels récursifs jusqu'à obtenir une réponse pour le grand tableau initial.



Fonction de recomposition `merge`. La fonction de recomposition, qui fusionne deux tableaux triés en un grand tableau trié, est décrite ci-dessous. Nous l'avons codé sur des listes en OCaml également:

```c
// cette fonction fusionne les deux segments triés [l, m[ et [m, r[
// du tableau tab et range le résultat de cette fusion dans le segment
// [l, r[ de res
void merge(int *res, int *tab, int l, int m, int r)
{
  // préconditions
  assert(l >= 0); assert(l <= m); assert(m <= r);
  assert(tab != NULL); assert(res != NULL);

  int k;
  int idx1 = l; // indice d'avancée dans la partie gauche [l, m]
  int idx2 = m; // indice d'avancée dans la partie droite [m+1, r[

  for (k = l; k < r; k++)
  {
    // insertion d'un élément de [l,m] dans res
    if ( (idx1 < m) && (idx2 < r ) && (tab[idx1] <= tab[idx2]) )
    {
      res[k] = tab[idx1];
      idx1 = idx1+1;
    }
    // insertion d'un élément de [m+1,r[ dans res
    else if ( (idx1 < m) && (idx2 < r ) && (tab[idx2] < tab[idx1]) )
    {
      res[k] = tab[idx2];
      idx2 = idx2+1;
    }
    // il n'y a plus d'éléments à merger issus du segment [m+1, r[
    else if ( (idx1 < m) && (idx2 >= r) )
    {
      res[k] = tab[idx1];
      idx1 = idx1 + 1;
    }
     // il n'y a plus d'éléments à merger issus du segment [l, m]
    else if ( (idx1 >= m) && (idx2 < r) )
    {
      res[k] = tab[idx2];
      idx2 = idx2 + 1;
    }
    else
    {
      printf("Erreur k=%d idx1=%d idx2=%d l=%d r=%d m=%d\n", k, idx1, idx2, l, r, m);
      return;
    }
  }
  return;
}
```

Pour trier l'ensemble des valeurs, il suffit de mettre à jour deux indices `idx1` et `idx2` de parcours dans chacun des deux sous-tableaux à fusionner, un indice `k` indiquant la prochaine case à remplir dans dans le tableau fusionné résultat. A chaque itération, on regarde quel est le plus petit des deux éléments entre celui d'indice `idx1` dans le sous-tableau gauche et celui d'indice `idx2` dans le tableau de droite. On sélectionne la plus petite des deux valeurs, que l'on range dans le tableau résultat à l'indice `k`, indice que l'on incrémente pour qu'il désigne la prochaine case à remplir. Si la plus petite des deux valeurs était dans le tableau de gauche, on incrémente `idx1`, sinon, on incrémente `idx2`. On répéte cette opération jusqu'à ce que tous les éléments des deux sous-tableaux aient été traités. Si un sous-tableau est terminé avant l'autre, on purge le sous-tableau restant intégralement dans le tableau résultat. Il n'est pas difficile de démontrer que cette fonction est linéaire, en $\Theta(n)$ où n est le nombre d'éléments à fusionner.

Algorithme global récursif *top-down* `merge_sort`. Voyons maintenant la formule récursive globale:

```c
void aux_merge_sort(int *tmp, int *res, int l, int r)
{

  assert(tmp != NULL);
  assert(res != NULL);

  if (l >= r-1) // cas d'arrêt, segment de taille 0 ou 1
    return;

  int m = (r+l)/2;
  printf("Segment [l=%d, m=%d[\t ", l, m);
  printf("Segment [m=%d, r=%d[\n", m, r);

  aux_merge_sort(tmp, res, l, m); // appel récursif sur la moitié gauche.
  // entrée: tmp, sortie dans res.
  // A l'issue de cet appel, la moitié gauche [l, m] de res  est triée

  aux_merge_sort(tmp, res, m, r); // appel récursif sur la moitié droite
    // entrée: tmp, sortie dans res.
   // A l'issue de cet appel, la moitié droite [m+1, r] de res  est triée

  // copie des deux segement triés dans tmp
  copy_array(tmp, res, l, r); // copie res dans tmp

  // merge les deux segments [l, m] et [m+1, r[ de tmp et range le résultat dans res
  // entrée: tmp
  // sortie: res
  merge(res, tmp, l, m, r);

  return;
}
void merge_sort(int *tab, int n)
{
  assert(tab != NULL);
  assert(n >= 0);

  int *tmp = malloc(n*sizeof(int)); // allocation du tableau de travail
  assert(tmp != NULL); // check allocation ok

  copy_array(tmp, tab, 0, n);

  aux_merge_sort(tmp, tab, 0, n); // renvoie le resultat trié dans tab

  free(tmp);
  return;
}
```

La complexité temporelle de cette fonction vérifie la relation de récurrence suivante:

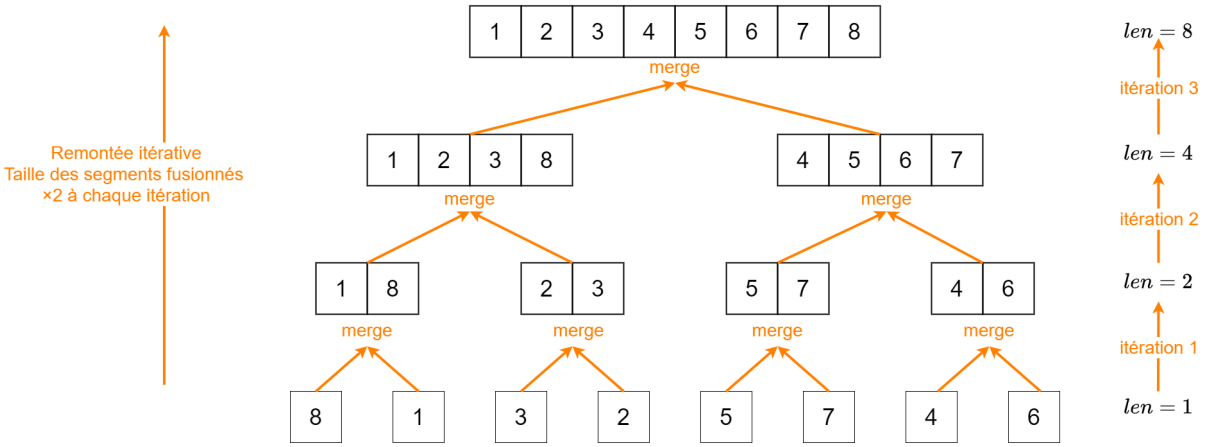
$$T(n) = \underbrace{T(\lfloor \frac{n}{2} \rfloor)}_{\text{appel moitié gauche}} + \underbrace{T(\lceil \frac{n}{2} \rceil)}_{\text{appel moitié droite}} + \underbrace{\Theta(n)}_{\text{coût recomposition fusion}}$$

On sait résoudre cette récurrence (réinjection...faites le, c'est une excellente manière de réviser) et on obtient une complexité temporelle de cette algorithme en $T(n) \in \Theta(n\log(n))$ ce qui est optimal dans tous les cas. C'est mieux que l'algorithme de tri rapide qui est optimal en moyenne seulement. Toutefois, cet algorithme requiert l'utilisation d'un tableau tampon `tmp` pour garantir la non interaction des appels récursifs. La complexité spatiale dans le segment de tas est donc en $\Theta(n)$, alors que tout se faisait en place, en $\Theta(1)$, pour l'algorithme de tri rapide.

IV. 2. b. Formulation itérative bottom-up

La formulation récursive précédente n'est pas récursive terminale, ce qui entraîne une complexité spatiale dans le segment de pile qui est en $O(\log(n))$ (hauteur de l'arbre des

appels = nombre maximal de blocs d'activation empilés).
On peut rendre cette complexité spatiale dans le segment de pile constante en reformulant cet algorithme de manière itérative par une approche ascendante, appelée *bottom-up* en anglais.
Il s'agit dans cette formulation de s'intéresser directement à la remontée des résultats, du cas de base jusqu'à la solution sur le grand tableau, sans passer par l'étape de descente récursive.



Dans ce cas, l'approche devient itérative: à chaque itération, on fusionne des segments du tableau pour obtenir un tableau avec des segments triés de plus en plus gros. La taille des segments fusionnés est multipliée par deux à chaque itération, jusqu'à atteindre un unique segment trié... qui correspond à l'intégralité du grand tableau de départ. La fonction de fusion de deux segments reste identique.

```c
void merge_sort_bottom_up(int *tab, int n)
{
  assert(tab != NULL);
  assert(n >= 0);

  int *tmp = malloc(n*sizeof(int)); // allocation du tableau de travail
  assert(tmp != NULL); // check allocation ok


  int l, r, m;
  // boucle sur la longueur len des segments fusionnés
  // a chaque tour, on multiplie la longueur len des segments fusionnés par 2
  for (int len = 1; len <= n; len = 2*len)
  {
    copy_array(tmp, tab, 0, n); // copie tab dans tmp

    for (l = 0; l <= n-len; l += 2*len) // on fusionne les segment deux par 2. Boucle sur les couples de segments [l, m[ et [m, r[
    {
      r = MIN(l + 2*len, n);
      m = l + len;
      merge(tab, tmp, l, m, r); // fusionne les segments [l, m[ et [m, r[ de tmp et stocke le résultat dans [l, r[ de tab
    }

  }

  free(tmp); // libération de l'espace mémoire du tableau de travail
  return;
}
```

V. Paradigme de la programmation dynamique

L'approche par programmation dynamique s'applique à des problèmes que l'on peut découper en sous-problèmes similaires éventuellement **non disjoints**.

V. 1. Exemple très classique: Fibonacci

On souhaite calculer le terme F_n de rang n de la suite de Fibonacci à partir de sa définition par la relation de récurrence suivante: $F_0 = 0$, $F_1 = 1$

$$\forall n \in \mathbb{N} \qquad F_{n+2} = F_{n+1} + F_n$$

V. 1. a. Approche récursive naïve.

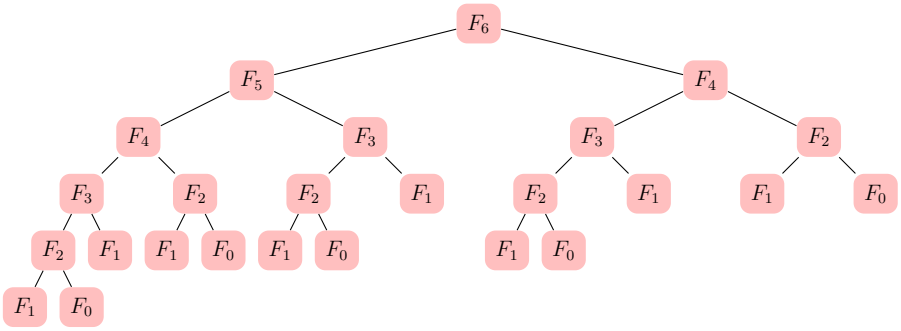
Une première implémentation consiste à écrire directement la relation de récurrence sous la forme d'un algorithme récursif:

```
# let rec fibo n =
  assert(n >= 0);
  match n with
  |0 -> 0
  |1 -> 1
  | _ -> (fibo (n-1)) + (fibo (n-2));;
val fibo : int -> int = <fun>
```

Avantage: l'implémentation est immédiate à partir de la relation de récurrence, le code est très proche de la formulation mathématique

Inconvénients: la complexité temporelle est exponentielle, en $O(\varphi^n)$ où $\varphi = (1+\sqrt{5})/2$ est le nombre d'or. On a une complexité spatiale dans le segment de pile qui est linéaire, en $O(n)$, liée à l'empilement des blocs d'activation des appels récursifs, car la récursivité n'est pas terminale.

La complexité temporelle exorbitante vient du fait que de très nombreux calculs sont redondants, et que cette implémentation récursive naïve recalcule plusieurs fois des termes qui ont déjà été calculés:



Voici, par exemple, le nombre de calculs redondants pour l'exemple dont l'arbre des appels a été dessiné ci-dessus:

- $F_2 \to 5$
- $F_3 \to 3$
- $F_4 \to 2$
- $F_5 \to 1$

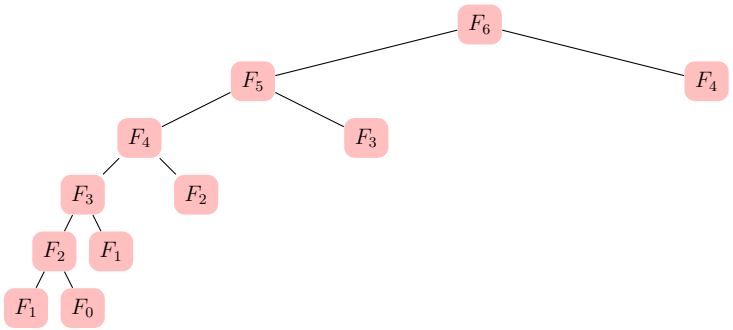
V. 1. b. Approche top-down (récursive) avec mémoïsation

Pour éviter de refaire plusieurs fois les mêmes calculs, une stratégie évidente consiste à stocker les résultats calculés et à tester, pour le calcul de chaque terme, s'il n'a pas déjà été effectué. Le fait de stocker des valeurs calculées pour éviter de recalculer plusieurs fois les solutions des mêmes sous-problèmes s'appelle la **mémoïsation**. Finalement, cette approche récursive où l'on stocke les résultats intermédiaires s'appelle **approche top-down avec mémoïsation**.

Voici donc une deuxième implémentation récursive (top down) utilisant une technique de mémoïsation avec un simple tableau association: le résultat du sous-problème F_i étant stocké dans la case d'indice i du tableau:

```
# let fibo_top_down_memo n =
  assert(n >= 0);
  let rec aux n memo =  (* top-down -> récursif *)
    match n with
    |0 -> 0
    |1 -> 1
    | _ -> if (memo.(n) = 0) then
             (memo.(n) <- (aux (n-1) memo) + (aux (n-2) memo); memo.(n) )
           else (* sous-problème deja resolu *)
             memo.(n)
  in
  let memo = Array.make (n+1) 0 in (* tableau de stockage, memoisation *)
  memo.(1) <- 1;
  aux n memo;;;
val fibo top down memo : int -> int = <fun>
```

On peut alors redessiner l'arbre des appels, en prenant en compte le fait que certains sous-arbres d'appels ont disparu, car on a utilisé un résultat pré-calculé, ce qui nous a évité de lancer à nouveau la cascade d'appels récursifs engendrés par ce calcul:



On atteint grâce à cette méthode une complexité temporelle en $O(n)$ car chaque terme de la suite n'est calculé qu'une seule fois. Par contre, cette amélioration temporelle s'est faite au prix d'une augmentation de la complexité spatiale, puisqu'il est nécessaire d'allouer un tableau de stockage de taille n. On a donc maintenant une complexité spatiale dans le segment du tas en $\Theta(n)$, alors qu'elle était en $\Theta(1)$ auparavant.

V. 1. c. Approche bottom-up (itérative) avec mémoïsation

On peut poursuivre l'amélioration en changeant de point de vue: plutôt que d'adopter une approche descendante, du problème global vers des sous-problèmes de taille de plus en plus petite, on peut adopter une approche constructive, de bas en haut: on résout des sous-problèmes, on en rassemble les solutions pour déterminer la solution du sous-problème

de taille supérieure...etc à la manière d'une construction inductive. Cette approche, dans le contexte de la programmation dynamique, est appelée **approche bottom-up avec mémoïsation**.
En voici une première implémentation avec un tableau:

```
# let fibo_bottom_up_memo n =
  assert(n >= 0);
  let memo = Array.make (n+1) 0 in (* tableau de stockage, memoisation *)
  (
    memo.(1) <- 1;
    for i = 2 to n do (* bottom-up -> iteratif *)
      memo.(i) <- memo.(i-1) + memo.(i-2)
    done;
    memo.(n)
  );;
val fibo_bottom_up_memo : int -> int = <fun>
```

Mais en fait, nous n'avons besoin que des deux derniers termes pour calculer le suivant. Si l'on souhaite juste obtenir in fine le terme F_n, on peut donc se contenter de deux variables entières de stockage, plutôt que d'allouer tout un tableau:

```
# let fibo_bottom_up_memo_optim n =
  assert(n >= 0);
  let un = ref 1 and unm1 = ref 0 in (* 2 variables de stockage uniquement *)
  (
    for i = 2 to n do
      let a = !un in
      (
        un := !un + !unm1;
        unm1 := a
      )
    done;
    !un
  );;
val fibo_bottom_up_memo_optim : int -> int = <fun>
```

Finalement, on aboutit... à l'algorithme que tout être humain mais naturellement en œuvre lorsqu'il calcule les termes successifs de la suite de Fibonacci!

Cet algorithme est optimal car:

la complexité temporelle est en $\Theta(n)$,

la complexité spatiale dans le segment de pile est en $\Theta(1)$ car il s'agit d'un algorithme itératif, sans empilement de blocs d'activation dans le segment de pile,

la complexité spatiale dans le segment du tas est en $\Theta(1)$ car nous n'avons besoin que de 2 entiers de stockage intermédiaire pour la mémoisation

V. 2. Concepts de la programmation dynamique

Définition 2 (Programmation dynamique (DP) - Mémoïsation)

La programmation dynamique consiste à résoudre un problème en le **décomposant en sous-problèmes similaires, généralement interdépendants (avec recouvrement)**, puis à résoudre les sous-problèmes, des plus petits aux plus grands (bottom-up) en **stockant** les résultats de ces sous-problèmes pour éviter les calculs redondants et les cascades récursives inutiles.

Le fait de stocker les résultats des sous-problèmes pour éviter de recalculer plusieurs fois le même sous-problème s'appelle la **mémoisation**.

Remarque. Lorsque l'on cherche à trouver une solution optimale, on cherche en fait deux choses:

- la valeur optimale
- la solution ayant mené à cette valeur optimale

Par exemple, dans le problème du rendu de monnaie, on cherche à savoir le nombre de pièces minimale que l'on peut rendre, mais on cherche également à savoir quelles sont exactement ces pièces. Par exemple, la solution du problème pour une somme à rendre de 13€ est: 3 pièces au minimum (valeur optimale), et la solution qui est amène cette valeur optimale est: 1 billet de 10€, 1 pièce de 2€ et 1 pièce de 1€

La programmation dynamique s'applique à des problèmes vérifiant certaines conditions:

- L'ensemble des éléments constituant le problème doit être *discret* et *fini*.
- Une solution optimale au problème global doit induire des solutions optimales aux sous-problèmes.
  (**Principe d'optimalité de Bellman**)
- Les *sous-problèmes* peuvent ne *pas* être *indépendants*. On parle de *recouvrement* (*overlap*) ou de *sous-problèmes superposés*.

Sous ces conditions, la programmation dynamique apporte une solution optimale au

problème posé.

Méthodologie 1 (Programmation dynamique).
En général, on procède par étapes:

1. Formulation du problème et de son découpage en sous-problèmes, en écrivant ce découpage de manière récursive par une approche top-down
2. Introduction de la mémoïsation, écriture d'une formulation récursive top-down avec mémoisation pour éviter les calculs redondants
3. Reformulation de l'algorithme précédent de manière itérative par une approche bottom-up avec mémoïsation
4. Tentative d'amélioration de la complexité spatiale dans le tas en ne mémoïsant que les résultats nécessaires au calcul des prochains sous-problèmes
5. Reconstitution éventuelles des solutions optimales ayant menée à la valeur optimale

L'expression *programmation dynamique* a été proposée dans les années 1950 par Richard Bellman, chercheur à la Rand Corporation, pour éviter les mots recherche et mathématiques.

- L'adjectif *dynamique* a été choisi par Bellman pour insister sur l'aspect temporel des problèmes mais aussi parce que le terme impressionnait !
- Le mot *programmation* est à comprendre au sens de *planification*, de l'*ordonnancement* et non au sens de codage. Historiquement, il désignait l'utilisation d'une méthode pour trouver un programme optimal dans un sens militaire : emploi du temps ou de la logistique.

L'un des principaux champs d'application de la *programmation dynamique* est la résolution de *problèmes d'optimisation*. Il s'agit le plus souvent de problèmes dont chaque solution possède une *valeur*. On cherche alors une *solution de valeur optimale*.

Exemple 1

Problème des pyramides de nombres, sac à dos, rendu de monnaie, plus court chemin, voyageur de commerce, distance d'édition de Levenshtein, plus longue sous-séquence croissante, ...

Il existe d'autres méthodes d'optimisation:

- les méthodes gloutonnes, mais nous avons vu qu'elles ne fournissent pas toujours une solution globalement optimale
- méthodes numériques