Algorithmique

Nicolas Barnier, Pascal Brisset

ENAC

Avril 2009

Algorithmique

Algorithms = Data Structure + Control (N. Wirth)

Du problème au programme :

- Choix de la représentation des données : structures de données
- Construction de la solution : contrôle
- Test, évaluation, complexité

Plan

Complexité

Ressources utilisées par un algorithme

Mesure de « l'efficacité » d'un algorithme :

- **complexité temporelle** : nombre d'opérations « élémentaires »
- **complexité spatiale** : nombre de « cases mémoire » consommées

Ces complexités sont **fonctions de la taille des données** (n) : nombre de valeurs dans un tableau, nombre de nœuds dans un graphe...

Faire abstraction du matériel

- Résultats **indépendants du matériel** : on ne compte pas précisément le nombre de cycles CPU ou le nombre de bits utilisés
- On s'intéresse au comportement **asymptotique**

$$e.g.\ \mathcal{O}(n),\ \mathcal{O}(n^2),\ \mathcal{O}(n \log n),\ \mathcal{O}(2^n)...$$

Comparaison des algorithmes

On pourra ainsi comparer l'efficacité de deux algorithmes qui résolvent le même problème.

Plusieurs caractérisations

- **en pire cas** sur l'ensemble des données (borne sup.)
- **en moyenne** sur l'ensemble des données (en pratique)

Classes de complexité

- algorithmes polynomiaux : « efficaces »
- algorithmes exponentiels

Structures de données

Modélisation des composants du problème

- Séquence, (multi-)ensemble, file, arbre, réseau... d'éléments du même type (éventuellement disjonctif)
- Taille : statique, dynamique

Opérations élémentaires

- Construction
- Accès, tests : séquentiel, aléatoire, indexation

Abstraction des opérations élémentaires :
écriture de l'algorithme **indépendante** des structures de données

Listes

- Suite finie d'éléments de même type
- Structure de données *récursive* : traitement par des fonctions récursives

Opérations élémentaires (en $\mathcal{O}(1)$)

- Accès au premier élément
- Accès au reste de la liste (tout sauf le premier élément)
- Ajout d'un premier élément à une liste
- Test à vide

Autres opérations : insertion, suppression d'un élément, test d'appartenance, longueur... $\mathcal{O}(n)$

Abstraction de la représentation des listes en OCaml

```
#exception EmptyList;;

#let car = fun l -> match l with
#    [] -> raise EmptyList
#  | x :: _ -> x;;
val car : 'a list -> 'a = <fun>

#let cdr = fun l -> match l with
#    [] -> raise EmptyList
#  | _ :: xs -> xs;;
val cdr : 'a list -> 'a list = <fun>

#let cons = fun x xs -> x::xs;;
val cons : 'a -> 'a list -> 'a list = <fun>

#let nil = []
#let is_empty = fun l -> l = nil;;
```

Abstraction de la représentation des listes (d'entiers) en C

list.c

```
struct cons {int car ; struct cons * cdr; };
typedef struct cons *list;
list nil = (list) 0;
list cons(int x, list y) {
  list c = (list) malloc(sizeof(struct cons));
  c->car = x; c->cdr = y; return(c);
}
int car(list c) { return(c->car); }
list cdr(list c) { return(c->cdr); }
void main() {
  list c = cons(34, cons(25, nil));
  printf("%d %d\n", car(c), car(cdr(c)));
}
```


Utilisation des listes

Avantages de la liste

- Simplicité
- Représentation économique
- Le nombre d'éléments est quelconque
- Partage possible (liste fonctionnelle)

Inconvénients de la liste

- Pas d'accès direct aux éléments sauf le premier : accès **séquentiel** $\mathcal{O}(n)$ (vs *aléatoire* $\mathcal{O}(1)$)
- Ajout d'un élément uniquement **en tête** de liste, sauf en **copiant** une partie de la liste (cf. concaténation, insertion...$\mathcal{O}(n)$)

Tableau

- Structure de **taille fixe** contenant des éléments de même type
- Chaque élément est accessible directement ($\mathcal{O}(1)$) par un index entier $\in [0..n-1]$
- Chaque élément est **modifiable**

Opérations élémentaires

- Création (dimensionnement)
- Taille (en OCaml)
- Accès au $i^{\text{ème}}$ élément
- Modification du $i^{\text{ème}}$ élément

Les tableaux en OCaml

```
#let t1 = [|1; 2; 3|];;
val t1 : int array = [|1; 2; 3|]

#Array.length t1;;
- : int = 3

#let t2 = Array.create 5 'z';;
val t2 : char array = [|'z'; 'z'; 'z'; 'z'; 'z'|]

#t2.(3);;
- : char = 'z'

#t2.(3) <- 'y';;
- : unit = ()

#t2;;
- : char array = [|'z'; 'z'; 'z'; 'y'; 'z'|]

#t2.(5) <- 'w';;
Exception: Invalid_argument "index out of bounds".
```

Les tableaux en C

Attention, en C

- Un tableau **n'est pas initialisé** quand il est déclaré
- Un tableau **ne contient pas sa taille** :
  nécessité d'un argument supplémentaire
- Pour l'accès et la modification, **aucun test sur l'indice** :
  un indice négatif ou trop grand conduit en général à une catastrophe (`segmentation fault`), voire pire...

Utilisation des tableaux

- Le type `array` n'est pas récursif, donc le parcours *uniforme* d'un tableau par une fonction récursive n'est pas forcément adapté
- **Boucles** `for`, **itérateurs** (`Array.iter`, `Array.map`...)

```
#for i = 0 to Array.length t2 - 1 do
#  t2.(i) <- 'a'
#done;;

#Array.map Char.uppercase t2;;
- : char array = [|'A'; 'A'; 'A'; 'A'; 'A'|]

#Array.init 4 (fun i -> i);;
- : int array = [|0; 1; 2; 3|]
```

array.c

```
main () {
  int t[10]; int i;
  for(i = 0; i < 10; i++) t[i] = i;}
```

Pile

Structure analogue à la liste mais **sans partage**

Opérations élémentaires

- Lecture d'éléments indexés par rapport au sommet de pile
- Ajout d'un élément sur la pile (`push`)
- Suppression du sommet de pile (`pop`)
- Test à vide

LIFO : Last In First Out

Exemple d'utilisation :

- Contrôle du parcours d'un arbre (profondeur)
- Gestion des appels fonctionnels dans un langage de programmation
- `Undo`

Implémentation : vision fonctionnelle (*rare !*)

On représente une pile par une liste :

```
#let create_stack = [];;

#let top =  fun stack ->
#  match stack with
#    [] -> raise EmptyStack
#  | x :: _ -> x;;

#let push = fun x stack -> x :: stack;;

#let pop = fun stack ->
#  match stack with
#    [] -> raise EmptyStack
#  | top :: stack -> (top, stack);;

#let empty = fun stack -> stack = [];;
```

La pile sera résultat de toute fonction susceptible de la modifier

Implémentation : vision impérative

Toujours avec une liste, mais au sein d'un objet **modifiable** (cf. module `Stack`) :

```
#let create_stack = fun () -> ref [];;

#let top = fun stack ->
#  match !stack with
#    [] -> raise EmptyStack
#  | x :: _ -> x;;

#let push = fun x stack -> stack := x :: !stack;;

#let pop = fun stack ->
#  match !stack with
#    [] -> raise EmptyStack
#  | top :: s -> stack := s; top;;

#let empty = fun stack -> !stack = [];;
```


Implémentation avec un tableau (classique)

```
#type 'a stack = {array : 'a array; mutable top : int};;
 type 'a stack = { array : 'a array; mutable top : int; }

#let create_stack = fun size default ->
#  {array = Array.create size default; top = -1};;
val create_stack : int -> 'a -> 'a stack = <fun>

#let top = fun {array = a ; top = t} ->
#  if t = -1 then raise EmptyStack else a.(t);;
val top : 'a stack -> 'a = <fun>

#let push = fun x stack ->
#  if stack.top >= Array.length stack.array - 1 then
#    raise Overflow
#  else begin
#    stack.top <- stack.top + 1;
#    stack.array.(stack.top) <- x end;;
val push : 'a -> 'a stack -> unit = <fun>
```

Implémentation avec un tableau (cont.)

```
#let pop = fun stack ->
#  if stack.top = -1 then
#    raise EmptyStack
#  else begin
#   stack.top <- stack.top - 1;
#   stack.array.(stack.top + 1) end;;
val pop : 'a stack -> 'a = <fun>

#let empty = fun stack -> stack.top = -1;;
val empty : 'a stack -> bool = <fun>
```

Utilisation :

```
#let s = create_stack 10 0;;

#push 3 s; push 2 s;;


#top s;;
- : int = 2

#let a = pop s and b = pop s in pop s;;
Exception: EmptyStack.
```


Implémentation en C (tableau)

stack.c

```
struct s {int* array; int top; int size; };
typedef struct s * stack;
stack create(int size){
  stack s = (stack) malloc(sizeof(struct s));;
  s->array = (int*) malloc(sizeof(int)*size);
  s->top = -1;
  s->size = size;
  return s;
}
int top(stack s){
  if (s->top == -1) exit(1);
  return s->array[s->top];
}
void push(int x, stack s){
  s->top = s->top + 1;
  if (s->top == s->size) exit(1);
  s->array[s->top] = x;
}
int pop(stack s){
  if (s->top == -1) exit(1);
  s->top = s->top - 1;
  return s->array[s->top+1];
}
int empty(stack s){
  return s->top == -1;
}
```

File

Suite finie d'éléments de même type où :

- le premier élément est accessible
- on ne peut ajouter un élément qu'à la fin

Opérations élémentaires

- Création
- Ajout d'un élément
- Retrait d'un élément
- Test à vide

FIFO : First In First Out

- Contrôle du parcours d'un arbre (largeur)
- Utilisation intensive dans les communications (asynchrones) : *buffer*

Implémentation naïve (mauvaise) en OCaml

```
#exception EmptyFifo;;
exception EmptyFifo

#let empty = [];;
val empty : 'a list = []

#let add = fun x fifo -> fifo @ [x];;
val add : 'a -> 'a list -> 'a list = <fun>

#let take = fun fifo ->
#  match fifo with
#    [] -> raise EmptyFifo
#  | x :: xs -> (x, xs);;
val take : 'a list -> 'a * 'a list = <fun>

#take (add 12 (add 13 empty));;
- : int * int list = (13, [12])
```

Implémentation avec un tableau (classique)

```
#type 'a fifo =
# {array: 'a array;
#  mutable first : int; mutable last : int};;

#let create = fun size default ->
#  {array = Array.create size default; first=0; last=0};;
val create : int -> 'a -> 'a fifo = <fun>

#let add = fun x fifo ->
#  let size = Array.length fifo.array in
#  let new_last = (fifo.last + 1) mod size in
#  if fifo.first = new_last then raise Overflow else begin
#  fifo.array.(fifo.last) <- x;
#  fifo.last <- new_last end;;
val add : 'a -> 'a fifo -> unit = <fun>
```


Implémentation avec un tableau (cont.)

```
#let take = fun fifo ->
#  if fifo.first = fifo.last then raise EmptyFifo else begin
#  let result = fifo.array.(fifo.first)
#  and size = Array.length fifo.array in
#  fifo.first <- (fifo.first + 1) mod size;
#  result end;;
val take : 'a fifo -> 'a = <fun>

#let f = create 3 "" in
#add "un" f;
#add "deux" f;
#print_string (take f); print_newline ();
#print_string (take f); print_newline ();
#print_string (take f);;
un
deux
Exception: EmptyFifo.
```

Table d'associations

- Ensemble de couples clé-valeur (cf. hashtable)
- Taille dynamique
- Espace des clés non-connu

Opérations élémentaires

- Ajout d'un couple
- Recherche d'une valeur à partir de la clé
- Modification de la valeur associée à une clé

```
#type ('a,'b) assoc = {key : 'a; mutable value : 'b}
#and ('a,'b) assocs = ('a,'b) assoc list;;

#let create_assocs = fun () -> [];;
#let add = fun k v table -> {key = k; value = v} :: table;;
```


Implémentation en OCaml

```
#let lookup = fun k t ->
#  let rec lkup = fun l ->
#    match l with
#      [] -> raise Not_found
#    | {key = a; value = b} :: _ when a = k -> b
#    | _ :: table -> lkup table in
#  lkup t;;
#let modify = fun k new_value t ->
#  let rec mdfy = fun l ->
#    match l with
#      [] -> raise Not_found
#    | couple :: table ->
#        if couple.key = k then couple.value <- new_value
#        else mdfy table in
#  mdfy t;;
```

Utilisation

```
#let t = create_assocs ();;
val t : 'a list = []

#let t1 = (add 3 "trois" (add 2 "deux" (add 1 "un" t)));;
val t1 : (int, string) assoc list =
  [{key = 3; value = "trois"}; {key = 2; value = "deux"};
   {key = 1; value = "un"}]

#lookup 3 t1;;
- : string = "trois"

#lookup 4 t1;;
Exception: Not_found.

#modify 2 "two" t1;;
- : unit = ()

#lookup 2 t1;;
- : string = "two"
```


Tri à bulles

Tri par échanges d'éléments consécutifs

```
#let tri = fun inf t ->
#  for i = Array.length t - 1 downto 1 do
#    for j = 1 to i do
#      if inf t.(j) t.(j-1) then swap j (j-1) t
#    done
#  done;;
 val tri : ('a -> 'a -> bool) -> 'a array -> unit = <fun>

#let tab = [|3;4;1;2;6;2;6;8;1|] in tri (<=) tab; tab;;
- : int array = [|1; 1; 2; 2; 3; 4; 6; 6; 8|]
```

Tri par insertion : sur une liste

```
#let tri = fun inf l ->
#  let rec insert = fun x l ->
#    match l with
#      [] -> [x]
#    | y :: ys ->
#       if inf x y then x::y::ys else y::(insert x ys) in
#  let rec tri_rec = fun l ->
#    match l with
#      [] -> []
#    | x::xs -> insert x (tri_rec  xs) in
#  tri_rec l;;


#tri (<=) [3;4;1;2;6;2;6;8;1];;
- : int list = [1; 1; 2; 2; 3; 4; 6; 6; 8]
```


Tri par insertion : sur un tableau

Sur un tableau : on insère dans un début de tableau trié.

```
#let tri = fun inf t ->
#  for i = 1 to Array.length t - 1 do
#    let j = ref i in
#    while !j > 0 && inf t.(!j) t.(!j-1) do
#      swap !j (!j-1) t;
#      j := !j-1
#    done
#  done;;
val tri : ('a -> 'a -> bool) -> 'a array -> unit = <fun>

#let tab = [|3;4;1;2;6;2;6;8;1|] in tri (<=) tab; tab;;
- : int array = [|1; 1; 2; 2; 3; 4; 6; 6; 8|]
```

Tri par sélection : sur un tableau

```
#let tri = fun inf t ->
#  for i = 0 to Array.length t - 1 do
#    let minimum = ref i in
#    for j = i + 1 to Array.length t - 1 do
#      if inf t.(j) t.(!minimum)
#      then minimum := j
#    done;
#    swap i !minimum t
#  done;;
val tri : ('a -> 'a -> bool) -> 'a array -> unit = <fun>

#let tab = [|3;4;1;2;6;2;6;8;1|] in tri (<=) tab; tab;;
- : int array = [|1; 1; 2; 2; 3; 4; 6; 6; 8|]
```


Tri rapide (*quicksort*)

```
#let tri = fun inf a ->
#  let rec trirapide = fun min max ->
#    if min < max then begin (* plus d'un element *)
#      let i = ref min and j = ref (max-1)
#      and pivot = a.(max) in
#      while !i < !j do (* croisement *)
#        while !i < max & inf a.(!i) pivot do incr i done;
#        while !j > min & inf pivot a.(!j) do decr j done;
#        if !i < !j then swap !i !j a (* echange *)
#      done;
#      if inf pivot a.(!i) then swap !i max a;
#      trirapide min (!i-1); trirapide (!i+1) max
#    end in
#  trirapide 0 (Array.length a - 1);;

#let tab = [|3;4;1;2;6;2;6;8;1|] in tri (<=) tab; tab;;
```

Tri fusion

```
#let rec fusion = fun inf l1 l2 ->
#  match (l1, l2) with
#    (xs, []) -> xs
#  | ([], ys) -> ys
#  | ((x::xs), (y::ys)) ->
#      if inf x y
#      then x::fusion inf xs (y::ys)
#      else y::fusion inf (x::xs) ys;;
```

```
#let rec coupe = fun l ->
#  match l with
#    [] -> ([], [])
#  | [x] -> ([x], [])
#  | x1::x2::xs ->
#      let (l1, l2) = coupe xs in (x1::l1,x2::l2);;
```


Tri fusion

```
#let rec coupe = fun l l1 l2 ->
#  match l with
#    [] -> (l1, l2)
#  | [x] -> (x::l1, l2)
#  | x1::x2::xs -> coupe xs (x1::l1) (x2::l2);;
```

```
#let tri = fun inf l ->
#  let rec tri_rec = fun l ->
#    match l with
#      [] -> []
#    | [x] -> [x]
#    | l ->
#        let (l1, l2) = coupe l [] [] in
#        fusion inf (tri_rec l1) (tri_rec l2) in
#  tri_rec l;;
```

```
#tri (<) [3;4;1;2;6;2;6;8;1];;
- : int list = [1; 1; 2; 2; 3; 4; 6; 6; 8]
```

Tri fusion

Pour éviter de couper-copier : trier les n premiers éléments et rendre le reste de la liste tel quel.

```
#let tri = fun inf l ->
#  let rec tri_rec = fun n l ->
#    match (n, l) with
#      (1, (x::xs)) -> ([x],xs)
#    | (n, l) ->
#        let (l1, l2) = tri_rec (n/2) l in
#        let (la, lb) = tri_rec (n - n/2) l2 in
#        (fusion inf l1 la, lb) in
#  fst (tri_rec (List.length l) l);;
val tri : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>

#tri (<) [3;4;1;2;6;2;6;8;1];;
- : int list = [1; 1; 2; 2; 3; 4; 6; 6; 8]
```


Tri fusion

```
#let tri = fun inf l ->
#  let rec singles = fun l ->
#    match l with
#      [] -> []
#    | e::es -> [e] :: singles es in
#  let rec fusion2 = fun l ->
#    match l with
#      l1::l2::rest -> fusion inf l1 l2 :: fusion2 rest
#    | x -> x in
#  let rec fusions = fun l ->
#    match l with
#      [] -> []
#    | [l] -> l
#    | llist -> fusions (fusion2 llist) in
#  fusions (singles l);;
val tri : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>

#tri (<) [3;4;1;2;6;2;6;8;1];;
- : int list = [1; 1; 2; 2; 3; 4; 6; 6; 8]
```

Tri par compartiments

Valeurs dont les clés sont des entiers entre 0 et $M-1$ tous distincts :

```
#let tri = fun m (t : (int * 'a) array) ->
#  let a = Array.create m (-1, snd t.(0)) in
#  for i = 0 to Array.length t - 1  do
#    a.(fst t.(i)) <- t.(i)
#  done;
#  let i = ref 0 in
#  for j = 0 to m-1 do
#    if fst a.(j) <> -1 then (t.(!i) <- a.(j); incr i)
#  done;;

#let t = [| (3,'t');(2,'d');(1,'u') |];;


#tri 10 t; t;;
- : (int * char) array = [|(1, 'u'); (2, 'd'); (3, 't')|]
```


Représentation des ensembles

On veut stocker un ensemble de couples clé-valeur en vue d'une recherche efficace via la clé.

Solutions basiques

Liste de couples

- Extensible
- Recherche de complexité linéaire

Tableau trié

- Non extensible
- Recherche dichotomique : complexité logarithmique

Arbre binaire de recherche

Définition (Arbre binaire de recherche)

Arbre binaire étiqueté qui vérifie la propriété suivante : pour tout nœud d'étiquette E de fils gauche G et de fils droit D, les étiquettes de G sont strictement inférieure à E et les étiquettes de D sont supérieures à E.

```
#type ('c,'v) abr =
#    Vide
#  | Noeud of 'c*'v*(('c,'v) abr)*(('c,'v) abr);;

#let rec insertion = fun inf cle valeur arbre ->
#  match arbre with
#    Vide -> Noeud (cle,valeur,Vide,Vide)
#  | Noeud (c,v,g,d) ->
#      if inf cle c
#      then Noeud (c,v,insertion inf cle valeur g,d)
#      else Noeud (c,v,g,insertion inf cle valeur d);;
```


Recherche d'un élément

```
#let ii = fun elt ->
#  insertion (<) elt (string_of_int elt);;

#let a = (ii 1 (ii 2 (ii 3 Vide)));;
val a : (int, string) abr =
  Noeud (3, "3", Noeud (2, "2", Noeud (1, "1", Vide, Vide), Vide), Vide)

#let rec recherche = fun inf cle arbre ->
#  match arbre with
#    Vide -> raise Not_found
#  | Noeud (c,v,g,d) ->
#      if c = cle then v
#      else if inf cle c
#      then recherche inf cle g
#      else recherche inf cle d;;

#recherche (<) 2 a;;
- : string = "2"
```

Complexité logarithmique... **si** l'arbre est bien *équilibré*.

Balancer un arbre : la moitié à gauche, la moitié à droite

On commence par extraire la liste triée des éléments :

```
#let rec couples = fun arbre ->
#  match arbre with
#    Vide -> []
#  | Noeud (c,v,g,d) ->
#      (couples g) @ [(c,v)] @ (couples d);;
val couples : ('a, 'b) abr -> ('a * 'b) list = <fun>

#let couples = fun arbre ->
#  let rec c_rec = fun reste arbre ->
#    match arbre with
#      Vide -> reste
#    | Noeud (c,v,g,d) -> c_rec ((c,v)::c_rec reste d) g in
#  c_rec [] arbre;;
val couples : ('a, 'b) abr -> ('a * 'b) list = <fun>
```


Balancer un arbre

```
#let ins_equil = fun liste ->
#  let rec ie_rec = fun liste n ->
#    if n = 0 then (Vide, liste) else
#    match ie_rec liste (n/2) with
#      (_, []) -> failwith "impossible !"
#    | (gauche,(c,v)::reste) ->
#        let (droite, reste2) = ie_rec reste (n-n/2-1) in
#        (Noeud (c,v,gauche,droite), reste2) in
#  fst (ie_rec liste (List.length liste));;
val ins_equil : ('a * 'b) list -> ('a, 'b) abr = <fun>

#let balance = fun arbre -> ins_equil (couples arbre);;
val balance : ('a, 'b) abr -> ('a, 'b) abr = <fun>

#balance a;;
- : (int, string) abr =
Noeud (2, "2", Noeud (1, "1", Vide, Vide), Noeud (3, "3", Vide
```

Hachage : adressage dispersé

Même principe que celui du tri par compartiments

- Tableau à M éléments (index compris entre 0 et $M-1$)
- $\mathcal{C}$ espace des clés. Fonction de *hachage* : $\mathcal{C} \longrightarrow 0..M-1$
- On range la donnée de clé c dans la case d'index $h(c)$

Gestion des *collisions*

- Par chaînage : chaque entrée de la table est une liste
- Par hachage linéaire

Gestion des collisions par chaînage

```
#let cree_table = fun taille -> Array.create taille [];;

#let ajoute = fun cle x h table ->
#  let index = h cle in
#  table.(index) <- (cle, x) :: table.(index);;

#let cherche = fun cle h table ->
#  let rec cherche_liste = fun l ->
#    match l with
#      [] -> raise Not_found
#    | (cle_x, x) :: xs ->
#        if cle = cle_x then x else cherche_liste xs in
#    cherche_liste table.(h cle);;
```

Gestion des collisions par chaînage

```
#let table = cree_table 5;;

#let hachage = fun x -> x mod 5;;

#type t = {cle: int; data: string};;

#let a = fun x -> ajoute x.cle x.data hachage table;;

#a {cle=1; data="un"}; a {cle=6; data="six"};
#a {cle=5; data="cinq"};;


#cherche 1 hachage table;;
- : string = "un"

#cherche 2 hachage table;;
Exception: Not_found.
```


Gestion des collisions par hachage linéaire

Pour insérer x

- Si la case $h(x)$ est occupée, essayer $(h(x)+1) mod\ M$, puis $(h(x)+2) mod\ M \ldots$
- Si la case $h(x)$ est occupée, essayer $(h(x)+h_1(x)) mod\ M$, puis $(h(x)+h_2(x)) mod\ M$ . . .où h_1, h_2 sont des fonctions de hachage secondaires. Pour le hachage linéaire : $h_i(x) = i$

En cas de débordement

- Abandon
- Doublement de la table avec re-hachage de toutes les entrées

Module Hashtbl

- `type ('a, 'b) t`
  The type of hash tables from type `'a` to type `'b`.
- `val create : int -> ('a,'b) t`
  `Hashtbl.create n` creates a new, empty hash table, with initial size `n`.
- `val clear : ('a, 'b) t -> unit`
  Empty a hash table.
- `val add : ('a, 'b) t -> 'a -> 'b -> unit`
  `Hashtbl.add tbl x y` adds a binding of `x` to `y` in table `tbl`. Previous bindings for `x` are not removed, but simply hidden. That is, after performing `Hashtbl.remove tbl x`, the previous binding for `x`, if any, is restored. (Same behavior as with association lists)
- `val find : ('a, 'b) t -> 'a -> 'b`
  `Hashtbl.find tbl x` returns the current binding of `x` in `tbl`, or raises `Not_found` if no such binding exists.

Module Hashtbl

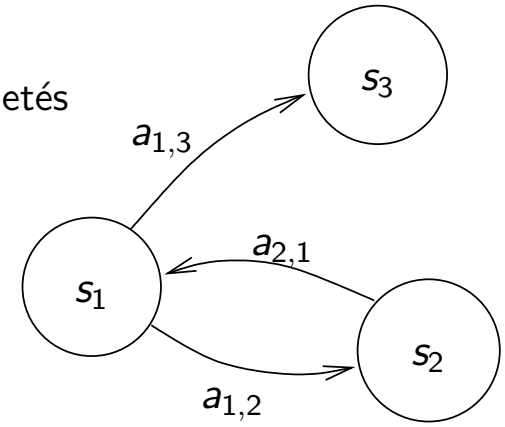
- `val find_all : ('a, 'b) t -> 'a -> 'b list`
  `Hashtbl.find_all tbl x` returns the list of all data associated with `x` in `tbl`. The current binding is returned first, then the previous bindings, in reverse order of introduction in the table.
- `val remove : ('a, 'b) t -> 'a -> unit`
  `Hashtbl.remove tbl x` removes the current binding of `x` in `tbl`, restoring the previous binding if it exists. It does nothing if `x` is not bound in `tbl`.
- `val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit`
  `Hashtbl.iter f tbl` applies `f` to all bindings in table `tbl`. `f` receives the key as first argument, and the associated value as second argument. The order in which the bindings are passed to `f` is unspecified. Each binding is presented exactly once to `f`.

Graphes orientés

Un graphe orienté est constitué de :

- Un ensemble $\mathcal{S}$ de *nœuds* (ou sommets)
- Un ensemble $\mathcal{A} \in \mathcal{S} \times \mathcal{S} \times \mathcal{E}$ d'*arcs* étiquetés

Un graphe est représenté graphiquement par :



Définition

- *On dit que le nœud s_3 est* **adjacent** *au nœud s_1.*
- *Un* **chemin** *est une suite de nœuds reliés par des arcs.*
- *Un* **circuit** *est un chemin bouclé.*

Exemple de graphe : réseau de communication

Représentation

Par la matrice d'adjacence

Si les nœuds sont numérotés de 1 à n

	1	$\ldots j \ldots$	n
1	$a_{1,1}$	$\ldots a_{1,j} \ldots$	$a_{1,n}$
i	$a_{i,1}$	$\ldots a_{i,j} \ldots$	$a_{i,n}$
n	$a_{n,1}$	$\ldots a_{n,j} \ldots$	$a_{n,n}$

où $a_{i,j}$ est l'étiquette de l'arc entre le nœud i et le nœud j (ou $\emptyset$).

```
#type 'e graphe1 = 'e array array;;
```

`'e` vaut `bool` si les arcs ne sont pas étiquetés
Représentation coûteuse si le graphe est *creux*

Représentation

Liste de listes d'adjacences

À chaque nœud on associe la liste de ses nœuds adjacents :

```
#type ('s,'e) graphe2 = (('s,'e) noeud) list
#and ('s,'e) noeud =
#  {etiq_noeud:'s; adjacents:(('s,'e) arc) list}
#and ('s,'e) arc = {etiq_arc:'e; adjacent:'s};;
```

Pour pouvoir utiliser les fonctions prédéfinies sur les listes d'associations :

```
#type ('s,'e) graphe3 = ('s*(('s*'e) list)) list;;
```

Représentation économique mais pas d'accès direct aux adjacences.

```
#[("s1",[("s2", 3); ("s3",2)]); ("s2",[("s1", 1)]); ("s3",[])];;
- : (string * (string * int) list) list =
[("s1", [("s2", 3); ("s3", 2)]); ("s2", [("s1", 1)]); ("s3", [])]
```


Exemple : une liste de nœuds est-elle un chemin ?

```
#let rec chemin = fun (graphe: bool graphe1) c ->
#  match c with
#    [] -> failwith "chemin vide"
#  | [s] -> true
#  | s1::s2::ss ->
#      graphe.(s1).(s2) && chemin graphe (s2::ss);;
val chemin : bool graphe1 -> int list -> bool = <fun>
```

```
#let rec chemin = fun (graphe: ('s,'e) graphe3) c ->
#  match c with
#    [] -> failwith "chemin vide !"
#  | [s] -> true
#  | (s1::s2::ss) ->
#      (try List.mem_assoc s2 (List.assoc s1 graphe) with
#        Not_found -> false) && chemin graphe (s2::ss);;
val chemin : ('a, 'b) graphe3 -> 'a list -> bool = <fun>
```

Exemple : une liste de nœuds est-elle un chemin ?

```
#let rec cherche_noeud = fun s1 g ->
#  match g with
#    [] -> []
#  | {etiq_noeud=s;adjacents=adjs}::l ->
#      if s = s1 then adjs else cherche_noeud s1 l;;

#let rec cherche_adjacent = fun s2 adjs ->
#   match adjs with
#     [] -> false
#   | {adjacent=s}::l ->
#       s = s2 || cherche_adjacent s2 l;;

#let rec chemin = fun (g : ('s,'e) graphe2) c ->
#  match c with
#    []  -> failwith "chemin vide !"
#  | [s] -> true
#  | s1::s2::ss ->
#      cherche_adjacent s2 (cherche_noeud s1 g)
#      && chemin g (s2::ss);;
```


Plus courts chemins

Algorithme de Floyd

- Graphe dont les arcs sont *pondérés* par un nombre positif
- Représentation par la matrice d'adjacence $p_{i,j}$
- $p_{i,j} = 0$ s'il n'y a pas d'arcs entre i et j
- Calcul de $l_{i,j}$ longueur du plus court chemin entre i et j
- Initialement $l_{i,j} = p_{i,j}$ pour $i \neq j$ et $l_{i,i} = 0$ $l_{i,j} = \infty$ si $p_{i,j} < 0$

n itérations : à la $k^{\text{ème}}$ itération, $l_{i,j}$ est le plus court chemin entre i et j ne passant pas par un nœud d'indice supérieur à k.

$$l^k_{i,j} = min \begin{cases} l^{k-1}_{i,j} \\ l^{k-1}_{i,k} + l^{k-1}_{k,j} \end{cases}$$

Algorithme de Floyd

```
#let infini = 1. /. 0.;;
val infini : float = infinity

#let floyd = fun (graphe: float graphe1) ->
#  let n = Array.length graphe in
#  let l = Array.create_matrix n n 0. in
#  for i = 0 to n-1 do for j = 0 to n-1 do
#      l.(i).(j) <-
#        if i = j then 0.
#        else if graphe.(i).(j) < 0. then infini
#        else graphe.(i).(j)
#  done done;
#  for k = 0 to n-1 do
#    for i = 0 to n-1 do for j = 0 to n-1 do
#      l.(i).(j) <- min l.(i).(j) (l.(i).(k) +. l.(k).(j))
#  done done done;
#  l;;
val floyd : float graphe1 -> float array array = <fun>

#floyd [|[|0. ; 8.; 5. |];
#        [|3. ; 0.; -1.|];
#        [|-1.; 2.; 0. |]|];;
- : float array array = [|[|0.; 7.; 5.|]; [|3.; 0.; 8.|]; [|5.; 2.; 0.|]|]
```

Complexité : $O(n^3)$

Application : centre d'un graphe

Centre : nœud d'*excentricité* minimale.

$$\textit{excentricité}(s) \stackrel{def}{=} \max_{t \in \mathcal{S}} \{\text{plus court chemin entre } s \text{ et } t\}$$

```
#let centre = fun graphe ->
#  let n = Array.length graphe in
#  let extremum = fun comp ligne ->
#    let maxi = ref 0 in
#    for i = 1 to n-1 do
#      if comp ligne.(i) ligne.(!maxi) then maxi := i
#    done; !maxi in
#  extremum (<)
#    (Array.map
#       (fun l -> l.(extremum (>) l)) (floyd graphe));;
val centre : float graphe1 -> int = <fun>

#centre [|[|0. ; 8.; 5. |];
#         [|3. ; 0.; -1.|];
#         [|-1.; 2.; 0. |]|];;
- : int = 2
```

Stratégies algorithmiques : diviser pour régner

- Diviser un problème en sous-problèmes (plus petits)
- Résoudre les sous-problèmes
- Recombiner les solutions (cf. tri fusion et tri rapide)

Exemple : multiplication de grands entiers

Multiplication de deux nombres X et Y écrits sur n bits. On « coupe » X et Y en deux nombre de $n/2$ bits chacun :

$$X = X_1 2^{n/2} + X_2 \quad Y = Y_1 2^{n/2} + Y_2$$
$$XY = X_1 Y_1 2^n + (X_1 Y_2 + X_2 Y_1) 2^{n/2} + X_2 Y_2$$

4 multiplications de nombres sur $n/2$ bits : on ne gagne rien !

$$XY = X_1 Y_1 2^n - ((X_1 - X_2)(Y_1 - Y_2) - X_1 Y_1 - X_2 Y_2) 2^{n/2} + X_2 Y_2$$

3 multiplications de nombres sur $n/2$ bits : $O(n^{\log_2 3})$

Stratégies algorithmiques : programmation dynamique

- La division d'un problème en sous problèmes peut engendrer de nombreux sous-problèmes identiques
- Mémoriser les résultats des calculs intermédiaires et les réutiliser

Exemple : match en n manches gagnantes

- Soit un match entre 2 joueurs de force égale dont le gagnant est le premier à remporter n manches
- Calcul de l'espérance de victoire à tout moment de la partie : $P(i,j)$ espérance pour i manches restant à remporter, contre j manches pour l'adversaire

$$P(i,j) = \begin{cases} 1 \text{ si } i = 0 \text{ et } j > 0 \\ 0 \text{ si } j = 0 \text{ et } i > 0 \\ (P(i-1,j) + P(i,j-1))/2 \text{ si } i > 0, j > 0 \end{cases}$$

Match : implémentation naïve

```
#let rec p = fun i j ->
#  match (i, j) with
#    (0, 0) -> failwith "impossible"
#  | (0, _) -> 1.
#  | (_, 0) -> 0.
#  | _ -> (p (i-1) j +. p i (j-1)) /. 2.;;
val p : int -> int -> float = <fun>

#p 2 3;;
- : float = 0.6875
```

Complexité exponentielle : $O(2^n)$

Match : calculs redondants

Remarque

Pour calculer $P(i,j)$, il est nécessaire de calculer $P(k,l)$ pour tout $k \leq i$ et $l \leq j$:

	0	1	2	3	4
0		1	1	1	1
1	0	1/2	3/4	...	
2	0	1/4	...		
3	0	...			

Match : programmation dynamique

```
#let p_prog_dyn = fun i j ->
#  let p = Array.create_matrix (i+1) (j+1) 0. in
#  for l = 1 to j do p.(0).(l) <- 1. done;
#  for k = 1 to i do
#    for l = 1 to j do
#      p.(k).(l) <- (p.(k-1).(l) +. p.(k).(l-1)) /. 2.;
#    done;
#  done;
#  p.(i).(j);;
val p_prog_dyn : int -> int -> float = <fun>

#p_prog_dyn 2 3;;
- : float = 0.6875

#let s = Sys.time () in let p = p 13 14 in
#(p, Sys.time () -. s);;
- : float * float = (0.577490508556366, 8.24)

#let s = Sys.time () in let p = p_prog_dyn 13 14 in
#(p, Sys.time () -. s);;
- : float * float = (0.577490508556366, 0.)
```


Stratégies algorithmiques : algorithmes gloutons (*greedy*)

À chaque étape, choix de la solution **localement optimale**.
Exemple : algorithme de Dijkstra pour trouver le plus court chemin dans un graphe.

Exemple : rendre la monnaie

Avec des billets de 10 €, 5 € et des pièces de 2 € et 1 € en utilisant un minimum de billets et de pièces : utiliser la plus grosse pièce possible, la soustraire du total et continuer.
Remarque : en général l'optimum **global** n'est pas atteint.
Exemple : rendre 4 € avec des pièces de 3 € (!), 2 € et 0.5 €.

```
#let rec monnaie = fun s pieces ->
#  if s = 0 then [] else match pieces with
#    p::ps -> if p <= s
#             then p :: monnaie (s-p) ps
#             else monnaie s ps
#  | [] -> failwith "Impossible";;
```

Exemple : problème du voyageur de commerce (*TSP*)

Comment passer dans toutes les villes de France, une fois et une seule, en parcourant une distance minimale.
Précisément : recherche d'un *circuit hamiltonien* de poids minimal dans un graphe non-orienté.

Algorithme glouton

1. Choisir l'arête de poids minimal
2. Vérifier qu'aucun sommet n'est utilisé plus de d'une fois
3. Vérifier qu'aucun circuit n'est produit (sauf pour la dernière arête)

Ne produit pas en général la solution optimale

Stratégies algorithmiques : recherche exhaustive

Parcours d'un arbre de recherche :

- Variables de décision à déterminer (n)
- À chaque étage de l'arbre, une variable est instanciée (*décision*)
- Toute les valeurs possibles pour cette variable sont essayées (d valeurs)
- Ensemble des feuilles de l'arbre = espace de recherche (taille : d^n)

Exemple : problème du voyageur de commerce

1. Construire l'arbre de tous les circuits hamiltoniens du graphe
2. Choisir le circuit le plus court

Plusieurs parcours d'un arbre possibles

- En profondeur d'abord (+ backtrack)
- En largeur d'abord (complexité spatiale exponentielle)
- Selon une heuristique (A* : meilleur d'abord)

En pire cas : complexité exponentielle

Stratégies algorithmiques : recherche locale

Exploration d'un espace de recherche par mouvements « locaux » dans le but de trouver une « bonne » solution.

Optimum local

1. Solution initiale « aléatoire »
2. Modifier la solution en choisissant un « voisin » qui l'améliore
3. Répéter tant qu'il y a amélioration

Remarques :

- La solution trouvée n'est pas nécessairement optimale
- Le critère d'arrêt est « flou »
- Problème des minima locaux

Tactiques d'échappement des minima locaux (recuit simulé, multi-start...)

Application : problème du voyageur de commerce