

彩色問題の SAT への還元

```

let rec gen m n f =      (* m から n までに対して f を適用し, そのリストを返す *)
  if m <= n then f m :: gen (m+1) n f else []
val gen : int -> int -> (int -> 'a) -> 'a list
let int = string_of_int      (* 変換関数の略 *)
val int : int -> string

let color_choice v n =    (* 各頂点に色がなければならない *)
  gen 1 n (fun i -> v ^ int i, true)
val color_choice : string -> int -> (string * bool) list
let color_conflict v1 v2 n = (* 辺でつながった頂点の色が違う *)
  gen 1 n (fun i -> [v1^int i, false; v2^int i, false])
val color_conflict : string -> string -> int -> (string * bool) list list

let rec flat_map f l =
  match l with
  | [] -> []
  | a::t -> f a @ flat_map f t      (* 結果のリストをくっ付ける *)
val flat_map : ('a -> 'b list) -> 'a list -> 'b list

let color_problem g n =    (* グラフと色の数から問題を生成 *)
  List.map (fun (v,_) -> color_choice v n) g @
  flat_map (fun (v1, e) -> flat_map (fun v2 -> color_conflict v1 v2 n) e) g
val color_problem :
  (string * string list) list -> int -> (string * bool) list list

let graph =
  ["A", ["B";"C";"D"]; "B", ["E"]; "C", ["F"]; "D", ["F"]; "E", ["F"]; "F", []]
val graph : (string * string list) list
# color_problem graph 3;;
- : (string * bool) list list =
[["A1", true]; ("A2", true); ("A3", true)]; ...;
[["A1", false]; ("B1", false)]; [("A2", false); ("B2", false)];
[("A3", false); ("B3", false)]; ... ]
# fast_sat (color_problem graph 3);;
- : Val.key list option = Some ["A1"; "B2"; "C2"; "D2"; "E1"; "F3"]

```

8 動的計画法 (DP)

複雑に見える問題でも, それをうまく解析すればそれを多項式時間で解ける場合もある. 動的計画法は部分的な問題から全体の問題が計算できる場合に有用である.

Fibonacci

とても単純な例で基本原理を見る.

```

let rec fibo n =
  if n < 2 then 1 else fibo (n-1) + fibo (n-2)
val fibo : int -> int
# fibo 40;;

```

```

- : int = 165580141 (* 何秒か待たされる *)
let fibo_dp n = (* DP を使った解 *)
  let rec fibo n =
    if n < 2 then (1,1) else
    let (a,b) = fibo (n-1) in (a+b, a) (* fibo(n-1) も一緒に返す *)
  in fst (fibo n)
val fibo_dp : int -> int
# fibo_dp 40;;
- : int = 165580141 (* 瞬時に出る *)

```

資源の利用の最適化

例えば、ホテルの一室の一年間の利用計画を立てる。申込はある日から別の日までの連続的な期間で、各申込を完全に受け入れるか拒否するか選んで、利用されている日数を最大にする。

```

type period = {start: int; stop:int} (* start <= stop *)

let sum = List.fold_left (+) 0 (* 汎関数を使う *)
val sum : int list -> int

let size free = sum (List.map (fun p -> p.stop - p.start + 1) free)
val size : period list -> int (* 空室期間のリストから日数を計算する *)

let rec alloc p ~free = (* 予約を入れる。できなければ、Not_found *)
  match free with
  [] -> raise Not_found
| p' :: rem ->
  if p'.start > p.start then raise Not_found else
  if p'.stop < p.stop then p' :: alloc p ~free:rem else
  let free' =
    if p'.stop = p.stop then rem else
    {start = p.stop+1; stop = p'.stop} :: rem
  in
  if p'.start = p.start then free' else
  {start = p'.start; stop = p.start-1} :: free'
val alloc : period -> free:period list -> period list

let rec solve ~free requests =
  match requests with [] -> 365 - size free (* 利用日数 *)
| p :: rem ->
  let sz = solve ~free rem in (* p を拒否 *)
  try
    let free' = alloc p free in (* p を予約 *)
    max sz (solve ~free:free' rem)
  with Not_found -> sz
val solve : free:period list -> period list -> int

let solve_alloc = solve ~free:[{start=1; stop=365}] (* 1日~365日が空室 *)
val solve_alloc : period list -> int

```

上記のアルゴリズムは NP であり、普通に実行すると、日数に対して指数的な計算量が要る。

DP による解

しかし、ある期間の最適な利用を、もっと短い期間の最適な利用から計算できることに着目する。

```
let period_matrix l =                                     (* 申込のリストを行列に変換 *)
  let m = Array.create_matrix 366 366 false in
  List.iter (fun p -> m.(p.start).(p.stop) <- true) l;
  m
val period_matrix : period list -> bool array array

let rec next_best m bests =                               (* n-1 日目までの最適な解が分かる *)
  let n = Array.length bests in
  let best = ref bests.(n-1) in
  for i = 1 to n do
    let best' = bests.(i-1) + (n-i+1) in
    if m.(i).(n) && best' > !best then best := best'
  done;
  Array.append bests [| !best |]                         (* n 日目の最適な解を追加 *)
val next_best : bool array array -> int array -> int array

let rec repeat n f x =                                    (* f を n 回繰り返す *)
  if n = 0 then x else repeat (n-1) f (f x)
val repeat : int -> ('a -> 'a) -> 'a -> 'a

let solve_dp requests =                                  (* 365 日目の最適な解を順に計算する *)
  let m = period_matrix requests in
  let bests = repeat 365 (next_best m) [|0|] in
  bests.(365)
val solve_dp : period list -> int
```

上のアルゴリズムの計算量は日数に対して $O(n^2)$ である。

配列を使わない方法

問題の計算空間を定めるのが動的計画法ならば、そこで配列の利用が本質的に見える。しかし、考え方を変えずにそれが避けられる。

```
let rec best_dp2 m n =
  if n = 0 then 0 else                                    (* 0 日間ならば 0 *)
  let rec iter i =
    if i = n then best_dp2 m (n-1) else                  (* n 日目に予約なし *)
    let use =                                             (* i+1 日目から n 日目までに予約を入れる *)
      if m.(i+1).(n) then best_dp2 m i + (n-i) else 0
    in max use (iter (i+1))
  in iter 0                                              (* i を 0 から n まで調べる *)
val best_dp2 : bool array array -> int -> int

let solve_dp2 requests =
  let m = period_matrix requests in
  best_dp2 m 365
val solve_dp2 : period list -> int
```

上のプログラムは配列をなくしたが、best_dp2 が同じ引数で何度も呼ばれるので、また指数的な計算量を要する。

配列版みたいに各結果を記憶すれば，その問題がなくなる．

```
let cache tbl f x =
  try Hashtbl.find tbl x
  with Not_found ->
    let y = f x in
      Hashtbl.add tbl x y;
    y
val cache : ('a, 'b) Hashtbl.t -> ('a -> 'b) -> 'a -> 'b

let cached_dp2 m =
  let tbl = Hashtbl.create 37 in
  let rec cached_dp2 n = cache tbl best_dp2 n
  and best_dp2 n =
    if n = 0 then 0 else
      let rec iter i =
        if i = n then cached_dp2 (n-1) else
          let use =
            if m.(i+1).(n) then cached_dp2 i + (n-i) else 0
          in max use (iter (i+1))
        in iter 0
      in cached_dp2
  val cached_dp2 : bool array array -> int -> int

let solve_cached_dp2 requests =
  let m = period_matrix requests in
  cached_dp2 m 365
val solve_cached_dp2 : period list -> int
```

部屋の数を増やす

部屋の数が増えると帰納条件が複雑になる．例えば，2室のとき

$$best_{kl} = \max(\{best_{ij} + (k-i) + (l-j) + 2 \mid i < k, j < l, (i, k) \in req, (j, l) \in req\} \cup \{best_{k-1l}, best_{kl-1}\})$$

そこで $best_{kl}$ とは，ある部屋が $k+1$ 日以降は空室で，もう一方の部屋が $l+1$ 日目以降空室のときの最大利用日数である．

部屋がさらに増えると，アルゴリズムが部屋の数に対して指数的になる．

練習問題 8.1 1. 以下の関数でデータを生成し，`solve_alloc` と `solve_dp` の計算量を比較せよ．申込数が 20 を越えると差が明かになる．

```
let gen_request () =
  let start = Random.int 359 + 1 in
  let stop = start + Random.int 7 in
  {start=start; stop=stop}

let gen_requests n =
  repeat n (fun l -> gen_request() :: l) []
```

2. *Fibonacci* にキャッシュを付けることで，最初の定義の効率を上げよ．

3. 部屋が二つのときのプログラムを書きなさい．同じ申込が両方の部屋で予約されないように気を付けなさい．