Interfejsy struktur danych z wykładu

Kolejka priorytetowa

```
module type PRI_QUEUE = sig
    (* Typ kolejek *)
    type a' pri_queue

    (* Pusta kolejka *)
    val empty_queue : a' pri_queue

    (* Czy pusta? *)
    val is_empty : a' pri_queue ->bool

    (* Wlozenie elementu *)
    val put : a' pri_queue -> a' -> a' pri_queue

    (* Wez maximum *)
    val getmax : a' pri_queue -> a'

    (* Usun maximum *)
    val removemax : a' pri_queue -> a' pri_queue

    (* Gdy kolejka pusta *)
    exception Empty_Queue
end;;
```

Mapa

Zamortyzowany $O(\log n)$

```
module type MAP =
    sig
    (* Mapa z wartosci typu 'a w 'b. *)
    type ('a,'b) map

    (* Wyjatek podnoszony,gdy badamy wartosc spoza dziedziny. *)
    exception Undefined

    (* Pusta mapa. *)
    val empty : ('a,'b) map

    (* Predykat charakterystyczny dziedziny mapy. *)
    val dom : ('a,'b) map -> 'a -> bool

    (* Zastosowanie mapy. *)
    val apply : ('a,'b) map -> 'a -> 'b

    (* Dodanie wartosci do mapy. *)
    val update : ('a,'b) map -> 'a -> 'b -> ('a,'b) map
end;;

module Map : MAP
```

Find-union

Zamortyzowany $O(m \cdot \alpha(m,n))$, gdzie $\alpha(m,n)$ funkcją odwrotną do funkcji Ackermanna (w praktyce można przyjąć czynnik za stały).

```
module type FIND_UNION = sig
    (* Typ klas elementów typu 'a. *)
    type 'a set

    (* Tworzy nowa klase zlozona tylko z danego elementu. *)
    val make_set : 'a -> 'a set

    (* Znajduje reprezentanta danej klasy. *)
    val find : 'a set -> 'a

    (* Sprawdza, czy dwa elementy sa rownowazne. *)
```

```
    val equivalent : 'a set -> 'a set -> bool

    (* Scala dwie dane (rozlaczne) klasy. *)
    val union : 'a set -> 'a set -> unit

    (* Lista elementow klasy. *)
    val elements : 'a set -> 'a list

    (* Liczba wszystkich klas. *)
    val n_of_sets : unit-> int
end;;

module FU : FIND_UNION
```

Backtracking

```
exception Solution of int
exception NoSolution

let rozw input =
    let q = ref (Queue.create()) (* BFS - najblizsze czesciowe rozwiazania do przejrzenia *)
    let visited = Hashtbl.create 424242 (* sprawdzanie czy dana konfiguracja wystapila wczesniej *)
    let gen cfg = ... (* generowanie pelniejszych konfiguracji w oparciu o obecna, dodawanie ich do
        kolejki *)
    let is_good cfg =
        if is_solve (cfg, kr) then raise (Solution kr) (* kr - to np. minimalna liczba krokow *)
        ... (* funkcja sprawdzajaca zgodnosc konfiguracji z zalozeniami / obcinanie galezi *)
    let is_solve cfg = ... (* funkcja sprawdzajaca, czy otrzymano rozwiazanie *)
    let bfs cfg =
        Queue.add (cfg, 0) !q;
        while not (Queue.is_empty !q) do
            gen (Queue.pop !q)
        done
    in
    try (bfs cfg_pocz; raise NoSolution) with
    | Solution x -> x
    | NoSolution -> -1
```

Programowanie dynamiczne

```
let memoize tab f x =
    if Map.dom !tab x then
        Map.apply !tab x
    else
        let wynik = f x
        in begin
            tab := Map.update !tab x wynik;
            wynik
        end

(* Wykorzystanie spamietywania *)

let rozw x =
    let tab = ref Map.empty in
    let f x =
        ...
    in
    memoize tab f x
```

Top-down opiera się na spamiętywaniu i zaczynaniu od całego problemu, rekurencyjnie rozwiązując coraz mniejsze. **Bottom-up** z najmniejszego podproblemu buduje coraz większe w oparciu o optymalną podstrukturę.

Programowanie zachłanne

Problemy, w których zastosowanie ma programowanie zachłanne, mają optymalną podstrukturę jak w DP. Jednak oprócz tego możliwe jest w nich wykonywanie lokalnie optymalnych wyborów, które zwalniają z potrzeby rozpatrywania na pewno niekorzystnych gałęzi. Najpierw

można wykonać dany problem dynamicznie, a później znaleźć zachłanną cechę. Chociaż czasem może być prościej od razu zachłannie. *Zasada dziel i zwyciężaj — własność optymalnej podstruktury: optymalne rozwiązanie jest funkcją optymalnych rozwiązań podproblemów i łączącego je zachłannego wyboru.*

Inne

Sumy prefiksowe

```
    (* Sumy prefiksowych tablicy arr *)
    let n = Array.length arr in
    let prf = Array.make (n+1) 0 in
    prf.(0) <- 0;
    for i = 1 to n do
        prf.(i) <- arr.(i-1) + prf.(i-1)
    done;
```

Wówczas `prf.(j)- prf.(i)= arr.(i)+ arr.(i+1)+ ... + a.(j-1)`.

Różne rozwiązania

Gwoździe

```
let gwozdzie lst =
    let srt = sort (fun x y -> compare (snd x) (snd y)) lst in
    let (a,b) = hd srt
    and gwozdzie = ref 0 in
    let rec aux prev x = function
        | [] -> ()
        | (p,q)::[] ->
                Printf.printf "wbij koncowy x=%d\n" x;
                gwozdzie := !gwozdzie + 1;
        | (p,q)::t ->
                if p <= x then
                    (Printf.printf "maksymalizacja dla p=%d\n" p;
                    aux q x t)
                else begin
                    Printf.printf "wbij x=%d\n" prev;
                    gwozdzie := !gwozdzie + 1;
                    aux q (fst (hd t)) t
                end
    in
    aux b a (tl srt);
    !gwozdzie
```

Dysponowanie kotami

```
let myszy k arr =
    let tab = ref Map.empty
    and n = Array.length arr in
    let prf = Array.make (n+1) 0 in
    let rec dp j l =
        if j = 0 || l = 0 then 0 else
        let res = ref (min_int) in
        for i = 1 to j do
            res := max (!res) ((dp (i-1) (l-1)) + prf.(j) - prf.(i) - pow (j-i-1))
        done;
        max (dp (j-1) l) (!res)
    in
    prf.(0) <- 0;
    for i = 1 to n do
        prf.(i) <- arr.(i-1) + prf.(i-1)
    done;
    memoize tab dp n k
```

Wyspa

```
let wyspa map =
    let n = Array.length map and m = Array.length map.(0) in
    let mapa = Array.make_matrix (n + 2) (m + 2) false in
    for i = 1 to n do
        for j = 1 to m do
            mapa.(i).(j) <- map.(i - 1).(j - 1)
        done
    done;
    let kol = ref (Queue.create())
    and ruchy_lad x y = [(x + 1, y); (x - 1, y);
                         (x, y + 1); (x, y - 1)] in
    let ruchy_woda x y =
        (ruchy_lad x y) @ [(x + 1, y + 1); (x + 1, y - 1);
                           (x - 1, y + 1); (x - 1, y - 1)]
    and maks = ref 0 and ranga = Array.make_matrix (n + 2) (m + 2) (-1) in
    let is_good a b =
        a >= 0 && b >= 0 && a < n + 2 && b < m + 2 && ranga.(a).(b) = -1 in
    let bfs waga (x, y) =
        let ruchy = if mapa.(x).(y) then ruchy_lad else ruchy_woda in
            Queue.add (x, y) !kol;
            while not (Queue.is_empty !kol) do
                let (a, b) = Queue.pop !kol in
                    ranga.(a).(b) <- waga;
                    List.iter (fun (c, d) ->
                    if is_good c d && mapa.(c).(d) = mapa.(a).(b) then
                        Queue.add (c, d) !kol) (ruchy a b)
            done
    in bfs 0 (0, 0);
    for i = 1 to n + 1 do
        for j = 1 to m + 1 do
            if ranga.(i).(j) = -1 then
                begin
                    let waga = if mapa.(i).(j) then ranga.(i - 1).(j) + 1
                    else ranga.(i - 1).(j) in
                    bfs waga (i, j);
                    maks := max !maks waga
                end
        done
    done;
    !maks
```

Antyczny kijek

```
let klej lst =
    let kawalki = ref lst
    and n = List.length lst in
    let koszty = Array.make (n-1) (max_int, 0)
    and t = ref 0 in (* t := liczba wykonanych sklejen *)
    let sklej i l =
        let rec pom curr = function
            | [] -> []
            | h::t -> if curr = i then
                match t with
                | [] -> [h]
                | hc::tc -> (h+hc)::tc
            else
                h::(pom (curr+1) t)
        in
        pom 0 l
    in
    while n - !t > 1 do (* n - !t to pozostale kawalki do sklejenia *)
        let arr = Array.of_list !kawalki in
        for i = 0 to n - !t - 2 do
            if max arr.(i) arr.(i+1) < fst koszty.(!t) then
                koszty.(!t) <- (max arr.(i) arr.(i+1), i);
        done;
        (* Printf.printf "%d %d %d koszt: %d\n" !rem arr.(snd koszty.(!t))
        arr.(snd koszty.(!t)+1) (fst koszty.(!t)); *)
        kawalki := sklej (snd koszty.(!t)) !kawalki;
        t := !t + 1;
    done;
```

```
    Array.fold_left (fun sum (a,_) -> a + sum) 0 koszty
```

Kajaki

```
open List;;
#use "fifo-lifo.ml";;

(* P R O B L E M    K A J A K O W Y

   W kajaku moze byc jedna lub dwie osoby
   Kajak = lista wag siedzacych w nim osob

   Rozwiazanie I
   Dla kazdego elementu maksymalnego dobieramy najwiekszy,
   ktory sie z nim miesci.  *)

let kajaki l wyp =
  (* Najgrubszego kajakarza nazwiemy grubasem.
     Tych sposrod pozostalych kajakarzy, ktorzy sie mieszcza
     z nim w kajaku nazwiemy chudzielcami.
     Pozostalych rowniez nazwiemy grubasami.  *)

  (* Skoryguj podzial na chudych i grubych. *)
  let rec dobierz g ch =
    if (is_empty_queue g) && (is_empty_queue ch) then (g, ch) else
    if is_empty_queue g then
      (make_queue [last ch], remove_last ch) else
    if queue_size g = 1 then (g, ch) else
    if first g + last g <= wyp then
      dobierz (remove_first g) (put_last ch (first g))
    else (g, ch)
  in
    (* Obsadz jeden kajak. *)
    let rec sadzaj gp chp acc =
      let (g, ch) = dobierz gp chp
      in
        if is_empty_queue g then acc else
    if is_empty_queue ch then
      sadzaj (remove_last g) ch ([last g]::acc)
    else
      sadzaj
        (remove_last g)
        (remove_last ch)
        ([last g; last ch]::acc)
    in
      sadzaj (make_queue l) empty_queue [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;


(* Rozwiazanie II
   Dla kazdego elementu minimalnego dobieramy najwiekszy,
   ktory sie z nim miesci. *)

let kajaki l wyp =
  (* Najchudszy kajakarz jest chudzielcem.
     Grubasy, to ci, ktorzy nie mieszcza sie z nim w kajaku.
     Pozostali to chudzielce.
     Jesli jest tylko jeden chudy, to przyjmujemy, ze jest on gruby. *)

  (* Skoryguj podzial na chudych i grubych. *)
  let rec dobierz ch g =
    if is_empty_queue ch then (ch, g) else
    if queue_size ch = 1 then
      (empty_queue, put_first g (last ch)) else
    if first ch + last ch > wyp then
      dobierz (remove_last ch) (put_first g (last ch))
    else
      (ch, g)
in
  (* Obsadz jeden kajak. *)
  let rec sadzaj chp gp acc =
    let (ch, g) = dobierz chp gp
```

```
    in
      if (is_empty_queue ch) && (is_empty_queue g) then acc else
      if is_empty_queue ch then
        sadzaj ch (remove_first g) ([first g]::acc)
      else
        sadzaj (remove_first (remove_last ch)) g
               ([first ch; last ch]::acc)
  in
    sadzaj (make_queue l) empty_queue [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;


(* Rozwiazanie III
   Jesli sie da, to laczymy najgrubszego z najchudszym. *)

let kajaki l wyp =
  let rec sadzaj q acc =
    if is_empty_queue q then acc else
    if queue_size q = 1 then [first q]::acc else
    if first q + last q <= wyp then
      sadzaj (remove_first (remove_last q)) ([first q; last q]::acc)
    else
      sadzaj (remove_last q) ([last q]::acc)
  in
    sadzaj (make_queue l) [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;



(* Rozwiazanie IV
   Maksymalne dwa kolejne *)

let kajaki l wyp =
  let rec dobierz ch g =
    match (ch, g) with
      (_, []) -> (ch, []) |
      ([], h::t) -> dobierz [h] t |
      (chh::cht, gh::gt) ->
        if chh + gh <= wyp then
          dobierz (gh::ch) gt
        else
          (ch, g)
  in
    let rec sadzaj chp gp acc =
      let (ch, g) = dobierz chp gp
      in
        match ch with
          [] -> acc |
          [h] -> sadzaj [] g ([h]::acc) |
          h1::h2::t -> sadzaj t g ([h2; h1]::acc)
    in
      sadzaj [] l [];;

kajaki [1; 2; 2; 3; 3; 3; 4; 6; 6; 8; 8; 8; 9; 9; 10] 10;;
```