Wstęp do Programowania potok funkcyjny

Marcin Kubica

2012/2013

Outline

Rodzaje kodów

- Kody stałej długości (np. ASCII).
- Kody zmiennej długości (np. Morse'a).
- Problem separatora — kody (bez-)prefiksowe.
- Kod zmiennej długości może być efektywniejszy niż kod stałej długości.

Kody stałej długości i prefiksowe

Example

- Wiadomość: BACADAEAFABBAAAGAH
- Kod stałej długości:

  A 000 C 010 E 100 G 110
  B 001 D 011 F 101 H 111

  001000010000011000100000101000001001000000000110000111
  (54 bity)

-

  A 0 C 1010 E 1100 G 1110
  B 100 D 1011 F 1101 H 1111

  100010100101101100011010100100000111001111 (42 bity).

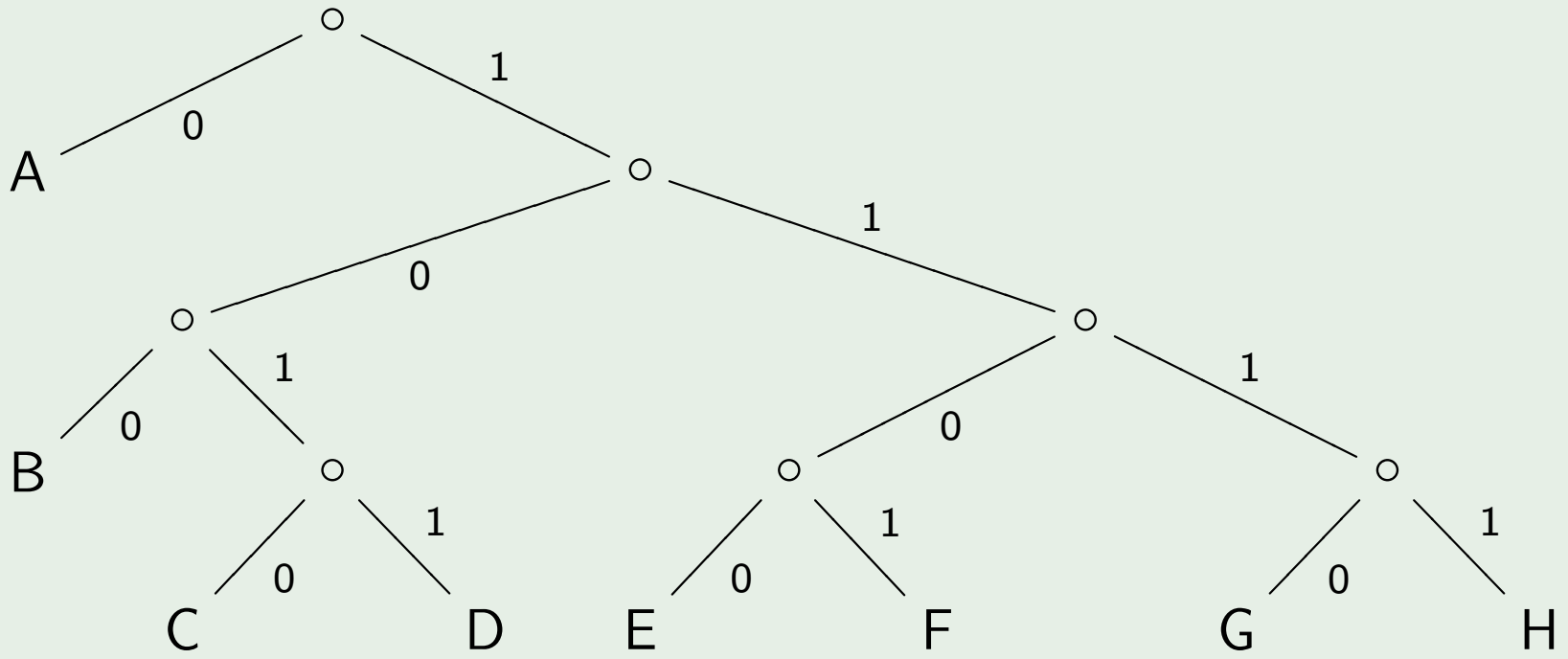
Reprezentacja kodu w postaci drzewa

- Problem optymalnego kodu prefiksowego (przy znanych częstościach występowania znaków)
- Reprezentacja optymalnego kodowania w postaci drzewa binarnego.
- Jeśli kod jest prefiksowy, to znaki odpowiadają liściom.
- Drzewo musi być regularne, inaczej kod nie jest optymalny.

Reprezentacja kodu w postaci drzewa

Example

Drzewo reprezentujące kod z poprzedniego przykładu.



Algorytm Huffmana

Algorytm Huffmana:

- Drzewo kodowania budujemy od liści do korzenia.
- Zbiór liści = zbiór znaków.
- Wybieramy dwa elementy o najmniejszych częstościach występowania.
- Sklejamy je i zastępujemy jednym poddrzewem.
  Częstość występowania sumuje się.
  Ich kody będą się różnić ostatnim bitem.
- Powtarzamy, aż pozostanie tylko jedno drzewo.

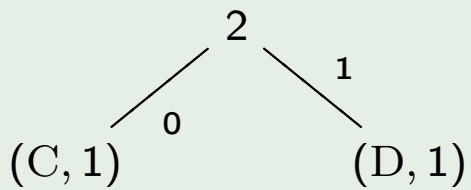
Algorytm Huffmana

Example

Działanie algorytmu Huffmana:

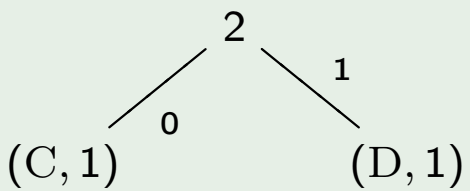
$$(C,1)(D,1)(E,1)(F,1)(G,1)(H,1)(B,3)(A,9)$$

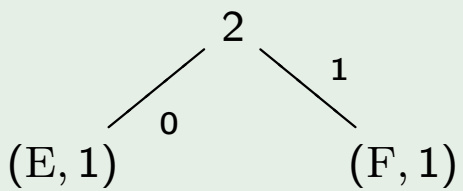
$(E,1)(F,1)(G,1)(H,1)$



$(B,3)(A,9)$

$(G,1)(H,1)$

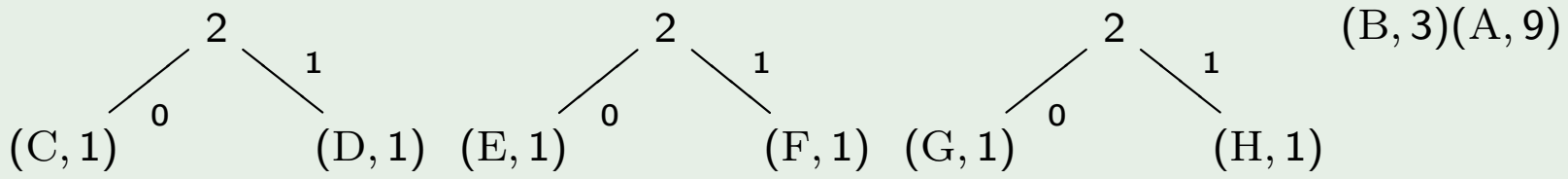


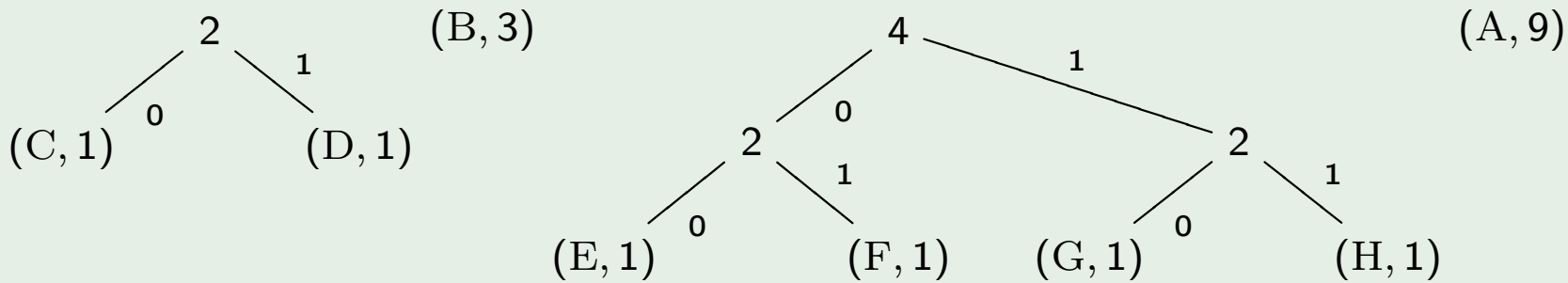


$(B,3)(A,9)$

Algorytm Huffmana

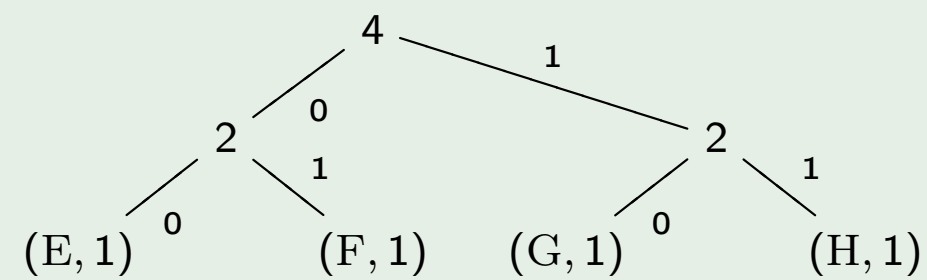
Example

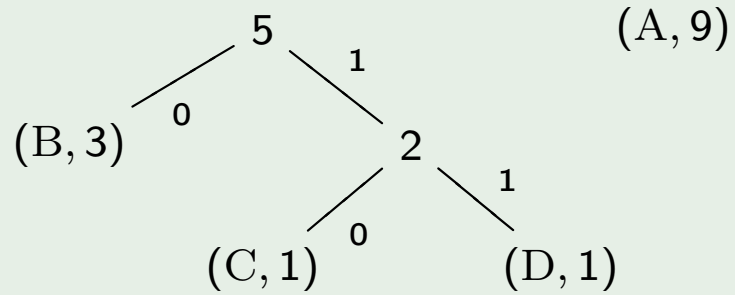


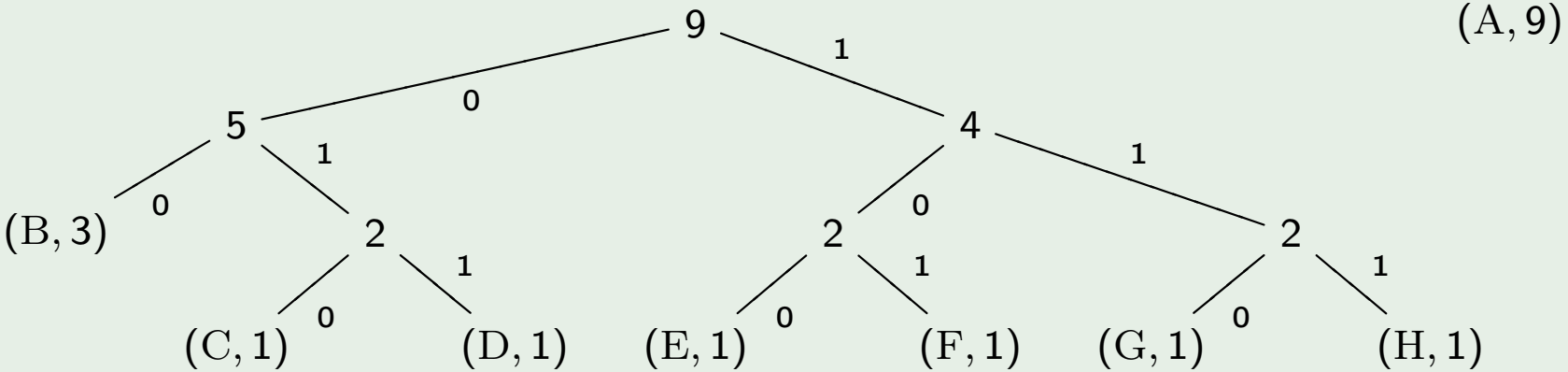


Algorytm Huffmana

Example

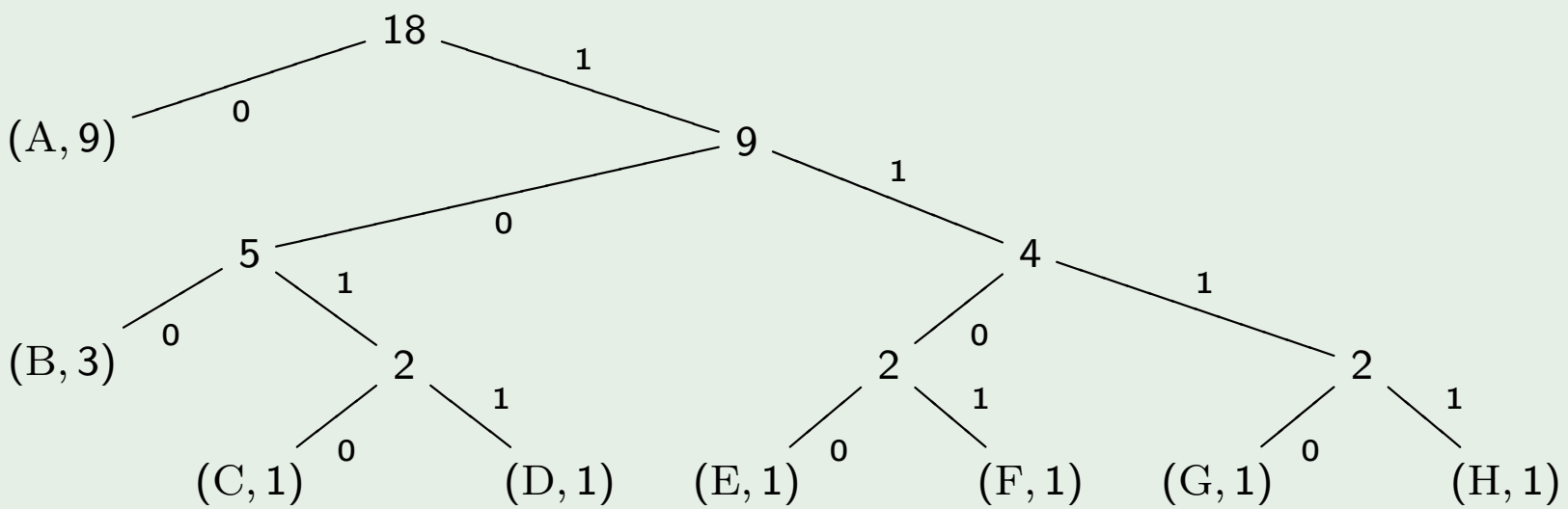






Algorytm Huffmana

Example



Optymalność wyboru

Lemma

$f(c)$ – częstość występowania.
x i y – para znaków o najmniejszej częstości wystąpień.
Istnieje optymalny kod prefiksowy, w którym kody dla x i y mają tę samą długość i różnią się tylko ostatnim bitem.

Dowód.

Rozważmy optymalne kodowanie.
b i c – dwa sąsiednie liście o maksymalnej głębokości $h_b = h_c$,
Bez straty ogólności zakładamy, że $f(b) \leq f(c)$ oraz $f(x) \leq f(y)$.
$f(x) \leq f(b) \quad f(y) \leq f(c) \quad h_x, h_y \leq h_b = h_c$.
Zamieniamy b z x i c z y. Długość kodu zmienia się o:

$$f(x)(h_b - h_x) + f(y)(h_c - h_y) + f(b)(h_x - h_b) + f(c)(h_y - h_c) =$$

$$(f(x) - f(b))(h_b - h_x) + (f(y) - f(c))(h_c - h_y) \leq 0$$

Powstały kod musi być również optymalny. □

Optymalność wyboru

Lemma

$f(c)$ – częstość występowania.
x i y – para znaków o najmniejszej częstości wystąpień.
Niech z będzie nowym symbolem, $f(z) = f(x) + f(y)$.
Niech T będzie drzewem optymalnego kodowania $(C \setminus \{x, y\}) \cup \{z\}$.
Drzewo T' powstałe przez dodanie do z synów x i y jest drzewem optymalnego kodowania dla C.

Dowód.

Gdyby T' nie było optymalne, to istniałoby lepsze drzewo kodowania, w którym x i y byliby braćmi.
Oznaczając ich ojca przez z i usuwając je uzyskalibyśmy drzewo kodowania lepsze od T, co nie jest możliwe. □

Implementacja algorytmu Huffmana

Example

Dane: lista znaków posortowana wg rosnących częstości występowania.
Wynik: drzewo optymalnego kodu.

```
let huffman l =
  let rec process col =
    if huff_size col = 1 then
      huff_first col
    else
      let t1 = huff_first col
      and col1 = huff_remove col
      in
        let t2 = huff_first col1
        and col2 = huff_remove col1
        in
          process (huff_put col2 (merge t1 t2))
  in process (make_huff_col l);;
```

Implementacja drzew Huffmana

Example

```
type 'a huff_tree =
  Letter of ('a * float) |
  Node of ('a huff_tree * 'a huff_tree * float);;

let frequency (t : 'a huff_tree) =
  match t with
    Letter (_, f) -> f |
    Node (_, _, f)   -> f;;

let merge t1 t2 =
  Node (t1, t2, frequency t1 +. frequency t2);;
```

Implementacja kolekcji drzew Huffmana

Example

```
type 'a huff_col =
  'a huff_tree fifo * 'a huff_tree fifo;;

let make_huff_col l =
  (fold_left
    (fun q x -> put q (Letter x))
    empty_queue l,
  empty_queue);;

let huff_size ((q1, q2): 'a huff_col) =
  size q1 + size q2;;

let huff_put ((q1, q2): 'a huff_col) t =
  ((q1, put q2 t): 'a huff_col);;
```

Implementacja kolekcji drzew Huffmana

Example

```
let huff_first ((q1, q2): 'a huff_col) =
  if q1 = empty_queue then first q2 else
  if q2 = empty_queue then first q1 else
    let f1 = first q1 and f2 = first q2
    in if frequency f1 <= frequency f2 then f1 else f2;;

let huff_remove ((q1, q2): 'a huff_col) =
  if q1 = empty_queue then (q1, remove q2) else
  if q2 = empty_queue then (remove q1, q2) else
    let f1 = first q1 and f2 = first q2
    in
      if frequency f1 <= frequency f2 then
        (remove q1, q2)
      else ((q1, remove q2): 'a huff_col);;
```

Schemat programowania zachłannego

- Szukamy optymalnego rozwiązania,
  złożonego z ciągu kolejno wybieranych elementów.
- Własność wyboru zachłannego — ciąg wyborów optymalnych lokalnie prowadzi nas do rozwiązania optymalnego globalnie,
- Jeśli istnieje rozwiązanie optymalne zgodne z dotychczasowymi wyborami, to istnieje rozwiązanie optymalne zgodne również z kolejnym wyborem.
  (W szczególności istnieje rozwiązanie optymalne zgodne z pierwszym wyborem.)
- Zasada dziel i zwyciężaj — własność optymalnej podstruktury: optymalne rozwiązanie jest funkcją optymalnych rozwiązań podproblemów i łączącego je zachłannego wyboru.

Problem kajakowy

Example

Problem kajakowy:

- Dane są wagi n osób, w postaci listy uporządkowanej niemalejąco.
- Jak usadzić te osoby w minimalnej liczbie kajaków?
- Kajaki są dwuosobowe i mają określoną wyporność.
- W kajaku może płynąć jedna lub dwie osoby.
- Zakładamy, że żadna z osób nie przekracza wyporności kajaka, ale dwie osoby mogą już przekroczyć wyporność kajaka.

Problem ten można rozwiązać zachłannie i to na kilka sposobów. Będziemy korzystać z kolejek dwustronnych.

Problem kajakowy – algorytm I

Example

Algorytm I:

- Najgrubszego kajakarza nazwiemy grubasem.
  Ci co mieszczą się z nim w kajaku to chudzielce.
  Pozostali to też grubasy.
- Idea: największy grubas płynie z najgrubszym chudzielcem.
- Im chudszy grubas, tym grubsi są chudzielce.
  Podział na grubasów i chudzielców będzie się zmieniał.
  Chudsze grubasy będą przechodzić do puli chudzielców.
  W razie braku grubasów, najgrubszy chudzielec zostanie grubasem.
- Początkowo zakładamy, że wszyscy to grubasy.
- Każdy tylko raz może się zmienić z grubasa w chudzielca i raz z chudzielca w grubasa.

Problem kajakowy – algorytm I

Example

```
let kajaki l wyp =
  let rec dobierz g ch = ...
  in
    let rec sadzaj gp chp 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;;
[[3; 3]; [6; 3]; [6; 4]; [8]; [8; 2]; [8; 2]; [9]; [9; 1]; [10]]
```

Problem kajakowy – algorytm I

Example

```
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)
```

Problem kajakowy – algorytm I

Example

```
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)
```

Problem kajakowy – algorytm I

Example

Uzasadnienie poprawności:

- Istnieje optymalne rozwiązanie, w którym największy grubas g i najgrubszy chudzielec c płyną razem.
  - Jeśli g płynie sam, to można do niego dosadzić c.
  - Załóżmy, że g płynie z jakimś x, więc $x \leq c$.
    c i x można zamienić miejscami.
- Własność optymalnej podstruktury — oczywista.
  Po obsadzeniu pierwszego kajaka, pozostali kajakarze powinni płynąć minimalną liczbą kajaków.

Problem kajakowy – algorytm II

Example

Algorytm II:

- Najchudszy kajakarz jest chudzielcem.
  Grubasy, to ci, którzy nie mieszczą się z nim w kajaku.
  Pozostali to chudzielce.
  Jeśli jest tylko jeden chudzielec, to przyjmujemy, że jest on grubasem.
- Idea: najchudszego chudzielca sadzamy z najgrubszym chudzielcem.
  Na koniec grubasy siadają samotnie.
- Im grubszy jest najchudszy chudzielec, tym mniej jest chudzielców.
  Początkowo wszyscy to chudzielce.
  Potem stopniowo przerzucamy chudzielców do puli grubasów.

Problem kajakowy – algorytm II

Example

```
let kajaki l wyp =
  let rec dobierz ch g = ...
in
  let rec sadzaj chp gp 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;;
[[10]; [9]; [8]; [3; 4]; [3; 6]; [3; 6]; [2; 8]; [2; 8]; [1; 9]]
```

Problem kajakowy – algorytm II

Example

```
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)
```

Problem kajakowy – algorytm II

Example

```
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)
```

Problem kajakowy – algorytm II

Example

Uzasadnienie poprawności:

- Jeśli brak chudzielców, to każdy płynie osobno.
- Jeśli są chudzielcy, to jest ich przynajmniej dwóch.
  Wówczas istnieje optymalne rozwiązanie, w którym najgrubszy g i najchudszy chudzielec c płyną razem.
  - Jeśli c płynie sam, to można do niego dosadzić g.
  - Załóżmy, że c płynie z jakimś x, więc $x \leq g$.
    g i x można zamienić miejscami.
- Własność optymalnej podstruktury — oczywista.
  Po obsadzeniu pierwszego kajaka, pozostali kajakarze powinni płynąć minimalną liczbą kajaków.

Problem kajakowy – algorytm III

Example

Algorytm III:

- Najgrubszego kajakarza sadzamy z najchudszym, o ile się zmieszczą.
- Jeśli nie, to najgrubszy kajakarz płynie sam.

Problem kajakowy – algorytm III

Example

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

Problem kajakowy – algorytm III

Example

Uzasadnienie poprawności:

- Jeśli najgrubszy kajakarz g nie może płynąć z najchudszym c, to musi płynąć sam.
- Jeśli g może płynąć z c, to istnieje optymalne rozwiązanie, w którym płyną razem.
  - Jeśli g płynie sam, to można do niego dosadzić c.
  - Załóżmy, że g płynie z jakimś x.
  - Jeśli c płynie sam, to można zamienić miejscami c i x.
  - Załóżmy, że c płynie z jakimś y.
    Wówczas $y \leq g$ i można zamienić g i y miejscami.
- Własność optymalnej podstruktury — oczywista.
  Po obsadzeniu pierwszego kajaka, pozostali kajakarze powinni płynąć minimalną liczbą kajaków.

Problem kajakowy – algorytm III

Example

- To rozwiązanie generuje takie same wyniki (z dokładnością do kolejności) jak poprzednie. Dlaczego?

Problem kajakowy – algorytm IV

Example

Algorytm IV:

- Wsadzamy razem dwóch kolejnych, jak najgrubszych, kajakarzy.
- Grubi są ci, którzy nie mieszczą się do jednego kajaka razem z kolejnym chudszym.
  Najchudszy jest z definicji chudy.
- Sadzamy razem dwóch najgrubszych chudych.
- Stopniowo grubi mogą przenosić się do puli chudych.

Problem kajakowy – algorytm IV

Example

```
let kajaki l wyp =
  let rec dobierz ch g = ...
  in
    let rec sadzaj chp gp acc = ...
    in sadzaj [] l [];;
```

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

Problem kajakowy – algorytm IV

Example

```
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)
```

Problem kajakowy – algorytm IV

Example

```
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)
```

Problem kajakowy – algorytm IV

Example

Uzasadnienie poprawności:

- Jeśli nawet dwaj najchudsi nie mogą płynąć razem, to każdy musi płynąć oddzielnie.
- Oznaczmy dwóch kolejnych najgrubszych, którzy mogą płynąć razem przez g i c, $g \geq c$.
  Istnieje optymalne rozwiązanie, w którym g i c płyną razem.
  - Jeśli g płynie sam, to można do niego dosadzić c.
    Jeśli c płynie sam, to można do niego dosadzić g.
  - Załóżmy, że g płynie z jakimś x, a c z jakimś y.
    Mamy: $x \leq c \leq g$.
  - y może płynąć razem z x, a g razem z c.
- Własność optymalnej podstruktury — oczywista.
  Po obsadzeniu pierwszego kajaka, pozostali kajakarze powinni płynąć minimalną liczbą kajaków.