Algorytmy i struktury danych (2022/2023)

Krzysztof Diks

Wykład 2 - sortowanie przez porównania: Insertion Sort, Merge Sort, Heap Sort, Quick Sort

Sortowanie (wewnętrzne) przez porównania

Dane: dodatnia liczba całkowita n
tablica a[1..n] = [e_1, e_2, …, e_n] elementów z uniwersum z liniowym porządkiem (U,≤)
Wynik: tablica a[1..n] = [$e_{i_1} \leq e_{i_2} \leq \ldots \leq e_{i_n}$]

Na tym wykładzie rozważamy algorytmy sortowania, w których sortowanie jest wykonywane za pomocą operacji porównywania i zamiany elementów w sortowanej tablicy.

Algorytmy kwadratowe

Insertion Sort (sortowanie przez wstawianie)

```
begin
  a[0] := -∞; { strażnik }
  for i ∈ [2,..,n] do
  begin
    v := a[i]; j := i-1;
    while v < a[j] do                              {liczba porównań: n-1 + Inv(a)}
    begin a[j+1]:= a[j]; j := j-1 end;             {liczba przesunięć: Inv(a)}
    a[j+1] := v
  end
end
```

Własności: w miejscu (+), stabilny (+), łatwy w zapisie (+), złożoność zależna od liczby inwersji (+/-), pesymistyczna złożoność kwadratowa (-)

Bubble Sort (sortowanie bąbelkowe)

```
begin
  for i ∈ [n,..,2] do
    for j ∈ [1,..,i-1] do
      if a[j] > a[j+1] then      { liczba porównań: n-1 + n-2 + … + 1 = n(n-1)/2 }
        a[j] :=: a[j+1];         { liczba zamian: Inv(a)  }
end
```

Własności: prosty w implementacji (+), w miejscu (+), stabilny (+),
kwadratowa liczba porównań, niezależnie do danych! (---)
liczba zamian zależna od liczby inwersji (+/-)

Selection Sort (sortowanie przez wybieranie)

```
begin
  for i ∈ [n,..,2] do
  begin
    i_max := 1;
    for j ∈ [2,..,i] do
      if a[j] > a[i_max] then          { liczba porównań: n-1 + n-2 + … + 1 = n(n-1)/2 }
        i_max := j;
    a[i] :=: a[i_max]                   { liczba zamian: n-1 }
  end
end
```

Własności: prosty w implementacji (+), w miejscu (+), nie jest stabilny! (-),
kwadratowa liczba porównań niezależnie do danych! (---)
mała liczba zamian, zawsze n-1! (+++)

Usprawnienie algorytmu sortowanie przez wstawianie – metoda Donalda L. Shella (1959)

Powiemy, że tablica a[1..n] jest **k-posortowana (uporządkowana)** dla ustalonego k ≥ 1, jeśli a[i] ≤ a[i+k] dla każdego i = 1, …, n-k.

Dla k = 1 tablica a jest oczywiście posortowana, natomiast dla k ≥ n porządek elementów w tablicy może być dowolny.

Sortowanie, w którym dla ustalonego h, 1 ≤ h < n, niezależnie sortujemy (niemalejąco) h podciągów tablicy a
a[1], a[1+h], a[1+2h], …
a[2], a[2+h]. A[2+2h], …
…
a[h], a[2h], a[3h], …
nazywamy **h-sortowaniem**.

Przykład

a = [**3**, **1**, **4**, **7**, **2**, **12**, **17**, **14**, **13**, **20**, **16**, **15**] tablica (ciąg) 3-posortowana (-y)
b = [**0, 1, 0, 0, 3, 0, 0, 1, 2, 0, 2, 3**] wektor inwersji, #inwersji = 12

2-sortowanie

a = [**2**, **1**, **3**, **7**, **4**, **12**, **13**, **14**, **16**, **15**, **17**, **20**] -tablica jest 2- i 3-posortowana
b = [**0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0**] wektor inwersji, #inwersji = 3

Twierdzenie 2.1
Jeśli na k-posortowanej tablicy a[1..n] wykonamy h-sortowanie, to pozostanie ona nadal k-posortowana.
Dowód

Spostrzeżenie: rozważmy dwie tablice b[1..n] i c[1..n] takie, że b[i] ≤ c[i] dla każdego i = 1, 2, …, n. Niech CompExch(i,j,d[1..n]) będzie procedurą porównania/zamiany elementów d[i] oraz d[j] w tablicy d, dla pewnych $1 \le i < j \le n$:

```
CompExch(i,j,d[1..n])::
begin
  if d[i] > d[j] then d[i] :=: d[j]
end;
```

Wówczas po wykonaniu CompExch(i,j,b[1..n]) oraz CompExch(i,j,c[1..n]) mamy nadal b[i] ≤ c[i] oraz b[j] ≤ c[j].

e ≤ f & g ≤ h => MIN(e,g) ≤ MIN(f,h) & MAX(e,g) ≤ MAX (f,h)

Mamy:

$-\infty$	...	$-\infty$	a[1]	a[2]	...	a[n-k]	a[n-k+1]	...	a[n-k]
$\leq$	...	$\leq$	$\leq$	$\leq$	...	$\leq$	$\leq$	...	$\leq$
a[1]	...	a[k]	a[k+1]	a[k+2]	...	a[n]	$+\infty$	...	$+\infty$

Jeśli teraz wykonamy na tablicy a h-sortowanie, to wykonując jednocześnie porównania/zamiany na górnym i dolnym ciągu widzimy, że h-sortowanie zachowuje k-posortowanie.

cnd.

Metoda Shella

```
ShellSort(h[1..k])::
```
{h[1..k] – uporządkowana rosnąco tablica dodatnich liczb całkowitych, h[1] = 1;
elementy tablicy h nazywamy **skokami** }

begin

for i ∈ [k..1] **do**

h[i]-sortowanie metodą sortowania przez wstawianie

end

Dlaczego metoda Shella jest poprawna?

Złożoność obliczeniowa metody Shella

autor/autorzy	ciąg skoków	złożoność
Papernov-Stasevitch, 1965; Pratt, 1971	1, 3, 7, 15, …, 2^j -1, …	$\Theta(n^{3/2})$
Sedgewick, 1982	1, 8, 23. …, $4^{j+1}+3\cdot 2^j+1$, …	$O(n^{4/3})$
Incerpi-Segewick, 1985; Selmer 1987	Istnieje ciąg skoków o długości O(log n)	$O(n^{1+1/k})$
Plaxton, Poonen, Suel, 1992	dowolny	$\Omega\left(n\left(\frac{\log n}{\log\log n}\right)^2\right)$
Pratt, 1971	1, 2, 3, 4, 6, 8, 9, …, 2^p3^q, …	$O(n\log^2 n)$

Złożoność algorytmu Pratta:

- długość ciągu skoków - $O(\log^2 n)$
- liczba inwersji w ciągu jednocześnie 2- i 3-posortowanym jest mniejsza od n/2

… a[i-5] a[i-4] a[i-3] a[i-2] a[i-1] a[i]

Usprawnienie Bubble Sort – Comb Sort (Włodzimierz Dobosiewicz, 1980; Stephen Lacey, Richard Box, 1991)

Idea: podobnie do algorytmu Shella; dany jest ciąg skoków i dla ustalonego skoku wykonujemy 1 przebieg z algorytmu Bubble Sort właśnie z tym krokiem. Kolejne kroki to

$$h_1 = \lfloor n/1.3 \rfloor, h_2 = \lfloor h_1/1.3 \rfloor, h_3 = \lfloor h_2/1.3 \rfloor, \ldots$$

Z eksperymentów wynika, że jeżeli otrzymujemy skok równy 9 lub 10, to zamieniamy go na 11.

```
Skok(h)::
begin
  h := ⌊h/1.3⌋;
  if (h = 9) OR (h = 10) then
    h := 11;
  if h = 0 then h := 1;
  return h
end;
```

```
Sortowanie Dobosiewicz (Comb Sort)::
begin
    h := n;
    repeat
        zamiana := false;
        h := Skok(h); i := 1;
        while  ((i+h) ≤ n) do
          if a[i] > a[i+h] then
          begin
            a[i] :=: a[i+h]; zamiana := true;
            i := i+h
          end
     until (h = 1) AND (NOT zamiana)
end;
```

Sortowanie przez wybieranie raz jeszcze, w ogólniejszej postaci:

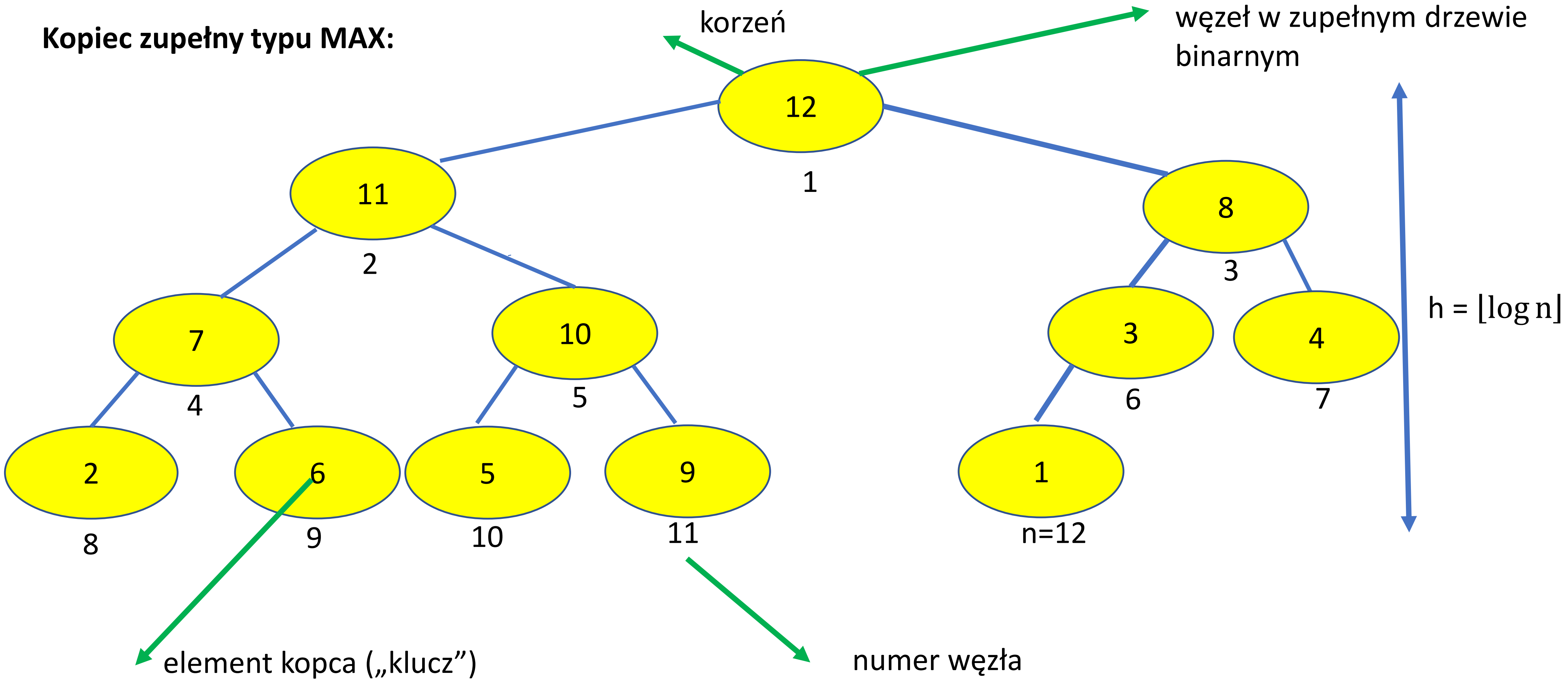
```
begin
  for i ∈ [n,..,2] do
  begin
    i_max := 1;
    for j ∈ [2,..,i] do
      if a[j] > a[i_max] then
        i_max := j;
    a[i] :=: a[i_max]
  end
end
```

```
begin
  for i ∈ [n,..,2] do
  begin

    i_max := IndexMax(i);
    { a[i_max] = max(a[1], a[2], ..., a[i]) }

    a[i] :=: a[i_max]
  end
end
```

Jak szybko znajdować maksimum w a[1..i]?

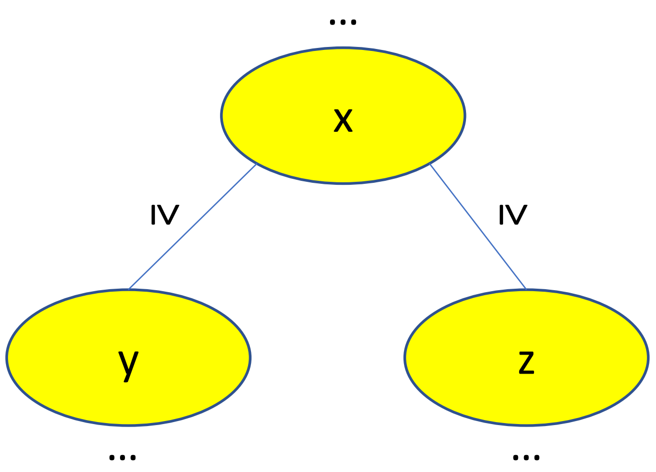
Odpowiedź: zastosować odpowiednią strukturę danych!

Kopiec zupełny typu MAX:

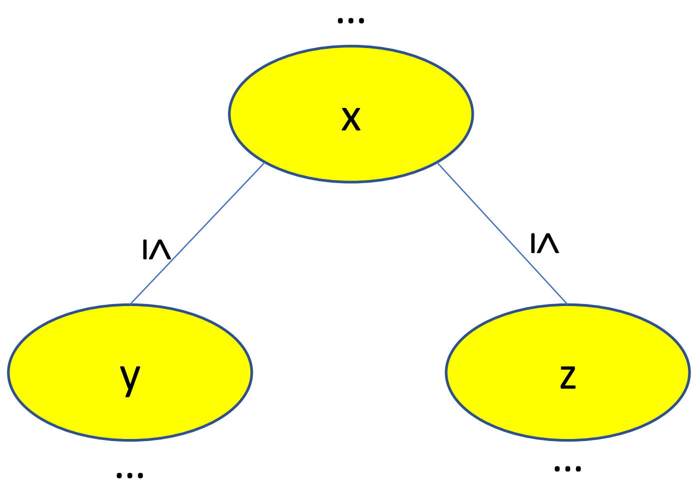


Rodzicem węzła o numerze i>1 jest węzeł $\lfloor i/2 \rfloor$, lewym dzieckiem węzła o numerze i jest węzeł o numerze 2i (o ile istnieje), prawym dzieckiem węzła o numerze i jest węzeł 2i+1 (o ile istnieje).

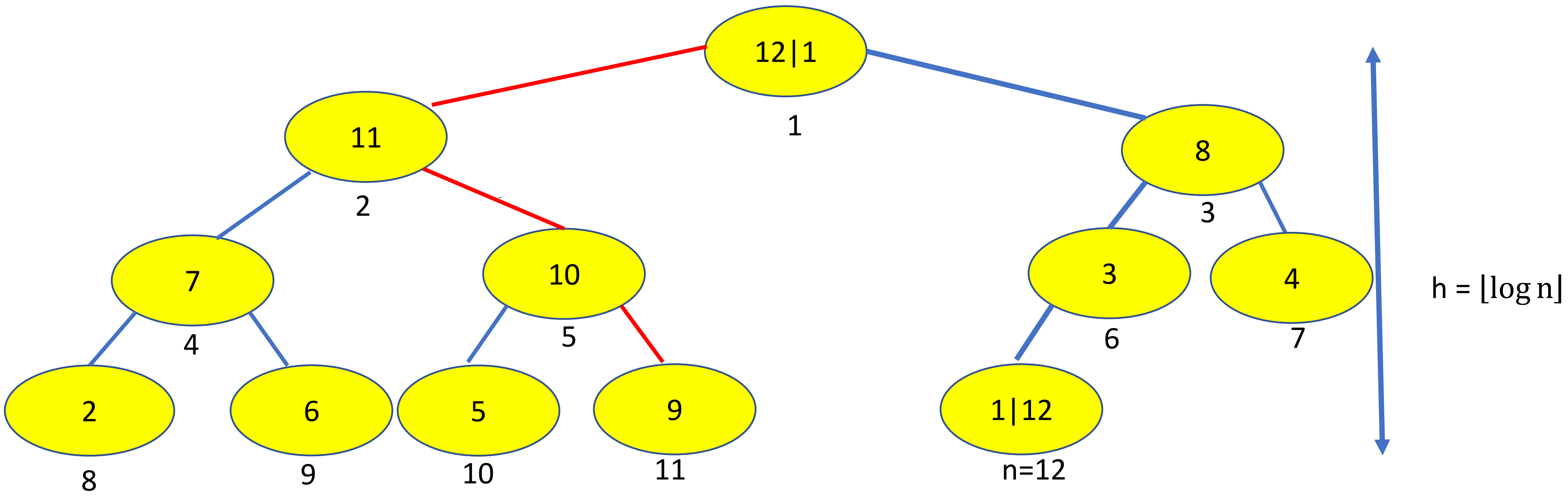
a[1..12] = [12,11,8,7,10,3,4,2,6,5,9,1]



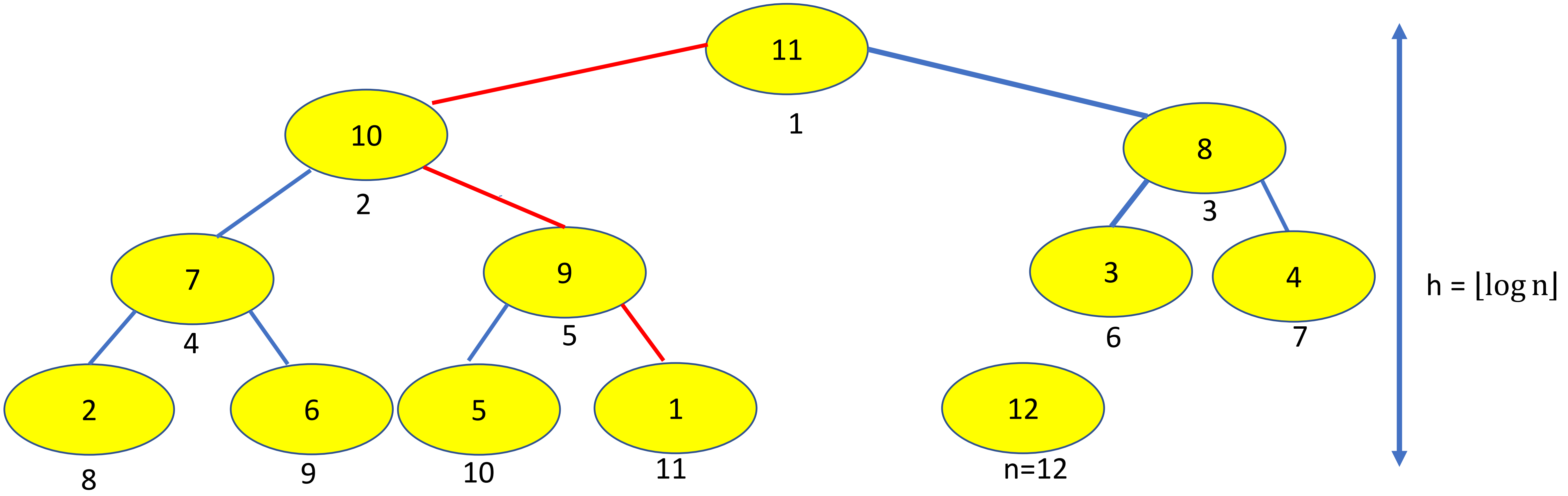
porządek kopcowy typu MAX
$x \geq y$ i $x \geq z$



porządek kopcowy typu MIN
$x \leq y$ i $x \leq z$



Zamiana elementu maksymalnego z korzenia z elementem w węźle o największym numerze i przywrócenie porządku kopcowego po czerwonej ścieżce.



Kopiec po odcięciu węzła z największym numerem i z przywróconym porządkiem kopcowym.

Dwie podstawowe operacje na (zaburzonym) kopcu zupełnym ukrytym w tablicy a[1..n]

Dla $1 \le l \le r \le n$ definiujemy warunek

$$heap(l,r) :: \forall_{l \le i \le r}\ (2i \le n \Rightarrow a[i] \ge a[2i]) \land (2i+1 \le n \Rightarrow a[i] \ge a[2i+1])$$

```
DownHeap(l,r)::
{ (1 ≤ l < r ≤ n) ∧ heap(l+1,r)
        ⇓ DownHeap(l,r) ⇓
  (1 ≤ l < r ≤ n) ∧ heap(l,r) }
begin
  i := l; j := 2*i; v := a[i];
  while j ≤ r do
  begin
    if j+1 ≤ r then
      if a[j] < a[j+1] then j := j+1;
      if v < a[j] then
      begin a[i] := a[j]; i := j; j := 2*i end
      else
        j := r+1; {siłowe wyjście z pętli}
  end;
  a[i] := v
end;
```

liczba obrotów pętli **while** $\le \left\lfloor \log \frac{r}{l} \right\rfloor$

```
UpHeap(l,r)::
{ (1 ≤ l < r ≤ n) ∧ heap(l,r-1)
        ⇓ UpHeap(l,r) ⇓
  (1 ≤ l < r ≤ n) ∧ heap(l,r) }
begin
  i := r; j := ⌊i/2⌋; v := a[i];
  while j ≥ l do
    if v > a[j] then
    begin
      a[i] := a[j];
      i := j; j := ⌊i/2⌋
    end
    else
      j := l-1; { siłowe zakończenie pętli}
   a[i] := v
end;
```

Heap Sort (sortowanie przez kopcowanie
John W. J. Williams, Robert W. Floyd, 1964)

```
HeapSort::
begin
{ budowa kopca }
   for i ∈ [⌊n/2⌋..1] do
   { heap(i+1,n) }
     DownHeap(i,n);

{ właściwe sortowanie }
   for i ∈ [n..2] do
   { a[1..i] ≤ a[i+1] ≤ … ≤ a[n]
     oraz
     heap(1,i) }
     a[1] :=: a[i];
     DownHeap(1,i-1)
end;
```

Analiza złożoności

B.o. przyjmijmy, że $n = 2^{h+1} - 1$ dla pewnego $h \geq 0$.

h – wysokość kopca

Liczymy pesymistyczną liczbę porównań.

$BK(n) = \sum_{i=h-1}^{0} 2(h-i)2^{i} \leq 2^{h+2} < 2n$

$WS(n) = \sum_{i=n}^{2} 2\lfloor \log i \rfloor \leq 2(n+1)\lfloor \log n \rfloor - 2^{\lfloor \log n \rfloor + 2} + 4$

Własności: $W_{HeapSort}(n) = \Theta(n\log n)$ (++), w miejscu (+++), nie jest stabilny (-)

Merge Sort (sortowanie przez scalanie, John von Neumann, 1945)

b[1..n] – globalna tablica pomocnicza

```
Merge(l,r,s)::
```

{ 1 ≤ l ≤ s <r ≤n; a[l] ≤ a[l+] ≤ … ≤ a[s]; a[s+1] ≤ a[s+2] ≤ … ≤ a[r]

Merge(l,r,s)

a[l] ≤ a[l+] ≤ … ≤ a[s] ≤ a[s+1] ≤ a[s+2] ≤ … ≤ a[r]

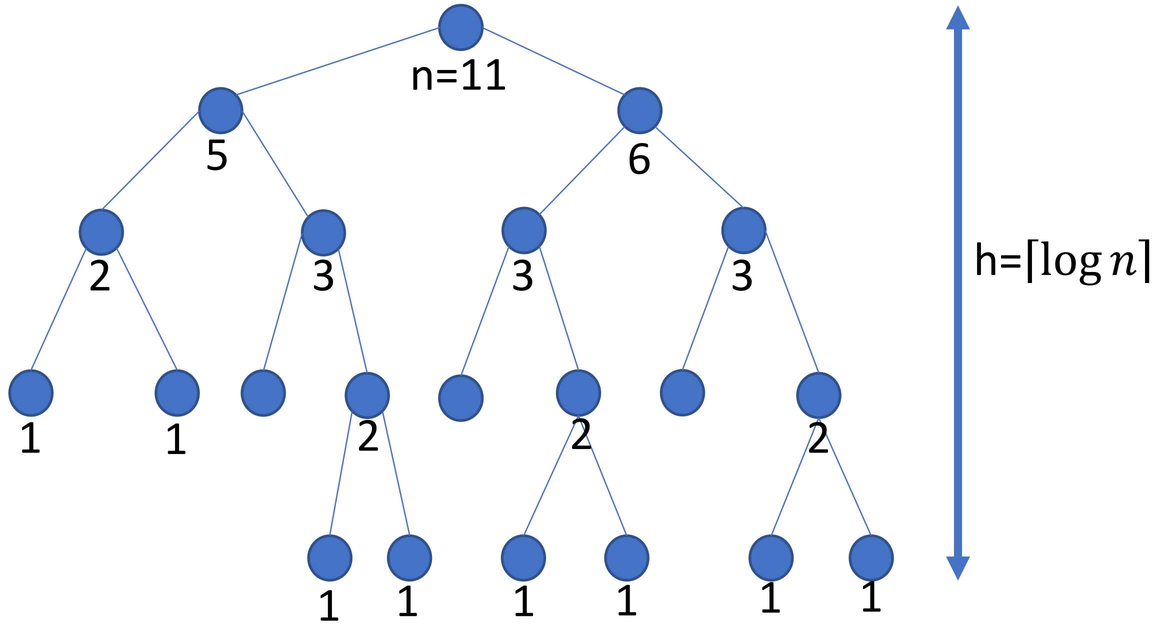
}

```
begin
   i := l; j := s+1; k := l-1;
   while (i ≤ s) AND (j ≤ r) do
   begin
      k := k+1;
      if a[i] ≤ a[j] then
      begin b[k] := a[i]; i := i+1 end
      else
      begin  b[k] := a[j]; j := j+1 end;
      if i ≤ s then a[k+1..r] := a[i..s];
      a[l..k] := b[l..k]
   end
end;
```

#porównań ≤ r-l

#przypisań na tablicach ≤ 2(r-l) + 1

```
MergeSort(l,r)::
{ dla 1 ≤ l ≤ r ≤ n po wykonaniu MergeSort(l,r) mamy
              a[l] ≤ a[l+1] ≤ … ≤ a[r]
}
begin
     if l < r then
     begin
          s := ⌊(l + r)/2⌋;
          MergeSort(l,s); MergeSort(s+1,r);
          Merge(l,r,s)
     end
end;
```



Żeby posortować całą tablicę wywołujemy MergeSort(1,n).

Niech C(n) oznacza liczbę porównań w algorytmie MergeSort. Mamy

$$C(n) < \begin{cases} 0 & n = 0,1 \\ C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lceil \frac{n}{2} \right\rceil\right) + n & n > 1 \end{cases}$$

Ostatecznie $\mathrm{C(n)} < \mathrm{n}\lfloor\log \mathrm{n}\rfloor + 2\mathrm{n} - 2^{\lfloor\log \mathrm{n}\rfloor+1}$.

$W_{MergeSort}(n) = \Theta(n\log n)$

- stabilny (+)
- bardzo mało porównań (+)
- sporo przypisań, możliwa redukcja (-/+)
- nie jest w miejscu = tablica b + rekursja!

„Sortowanie przez scalanie" w miejscu
Jyrki Katajanen, Tomi Pasanen, Jukka Teuhola, 1994

Pokażemy, że żeby scalić dwa uporządkowane ciągi o długościach odpowiednio d ≤ d' wystarczy pomocnicza tablica o rozmiarze d! Więcej, zadbamy o to żeby zawartość tej pomocniczej tablicy została zachowana, ale nie koniecznie w tym samym porządku.

```
MergeBis(l,r,s,b[p..q])::
{ 1 ≤ l ≤ s <r ≤n; a[l] ≤ a[l+] ≤ … ≤ a[s]; a[s+1] ≤ a[s+2] ≤ … ≤ a[r]; s-l+1 ≤  r-s; q-p = s-l
                    Merge(l,r,s)
      a[l] ≤ a[l+] ≤ … ≤ a[s] ≤ a[s+1] ≤ a[s+2] ≤ … ≤ a[r] }
begin
  b[p..q] :=: a[l..s]; {a[l..s] bufor}
  i := p; j := s+1; k := l-1;
  while (i ≤ q) AND (j ≤ r) do
  begin
    k := k+1;
    if b[i] ≤ a[j] then
    begin a[k] :=: b[i]; i := i+1 end
    else
    begin a[k] :=: a[j]; j := j+1 end;
    if i ≤ q then a[k+1..r] :=: b[i..q];
  end
end;
```

a:	2	5	12	20	1	3	4	13	17	19
⇓					↓					
	d	i	k	s	1	3	4	13	17	19
	⇓					↓				
	1	i	k	s	d	3	4	13	17	19
		⇓				↓				
	1	2	k	s	d	3	4	13	17	19
			⇓				↓			
	1	2	3	s	d	k	4	13	17	19
				⇓				↓		
	1	2	3	4	d	k	s	13	17	19
					⇓			↓		
	1	2	3	4	5	k	s	13	17	19

b:	d	i	k	s
	↓			
	2	5	12	20
	↓			
	2	5	12	20
		↓		
	i	5	12	20
		↓		
	i	5	12	20
		↓		
	i	5	12	20
			↓	
	i	d	12	20

				⇓			↓		
1	2	3	4	5	k	s	13	17	19

		↓	
i	d	12	20

					⇓		↓		
1	2	3	4	5	12	s	13	17	19

			↓
i	d	k	20

						⇓		↓	
1	2	3	4	5	12	13	s	17	19

			↓
i	d	k	20

							⇓		↓
1	2	3	4	5	12	13	17	s	19

			↓
i	d	k	20

								⇓		↓
1	2	3	4	5	12	13	17	19	s	

			↓
i	d	k	20

1	2	3	4	5	12	13	17	19	20

		↓	
i	d	k	s

Idea algorytmu sortowania przez scalanie w miejscu:

operacja	posortowana podtablica	koszt sortowania	koszt scalania
MergeSort(n/2+1,n), bufor a[1..n/2]	a[n/2+1..n]	$< c\frac{n}{2}\mathrm{nlog}\frac{n}{2}$	
MergeSort(n/4+1,n/2], bufor a[1..n/4]	a[n/4+1..n/2]	$< c\frac{n}{4}\mathrm{nlog}\frac{n}{4}$	
MergeBis(n/4+1,n/2,n), bufor a[1..n/4]	a[n/4+1..n]		< c'n
MergeSort(n/8+1,n/4], bufor a[1..n/8]	a[n/8+1..n/4]	$< c\frac{n}{8}\mathrm{nlog}\frac{n}{8}$	
MergeBis(n/8+1,n/4,n), bufor a[1..n/8]	a[n/8+1..n]		< c'n
MergeSort(n/16+1,n/8], bufor a[1..n/16]	a[n/16+1..n/8]	$< c\frac{n}{8}\mathrm{nlog}\frac{n}{8}$	
MergeBis(n/16+1,n/8,n), bufor a[1..n/16]	a[n/16+1..n]		< c'n
…			

$$< \mathrm{clog}\, n \sum \frac{n}{2^i} = \mathrm{cnlog}\, n \qquad c'\mathrm{nlogn}$$

Quick Sort (sortowanie szybkie, Hoare, 1962)

Ogólna idea

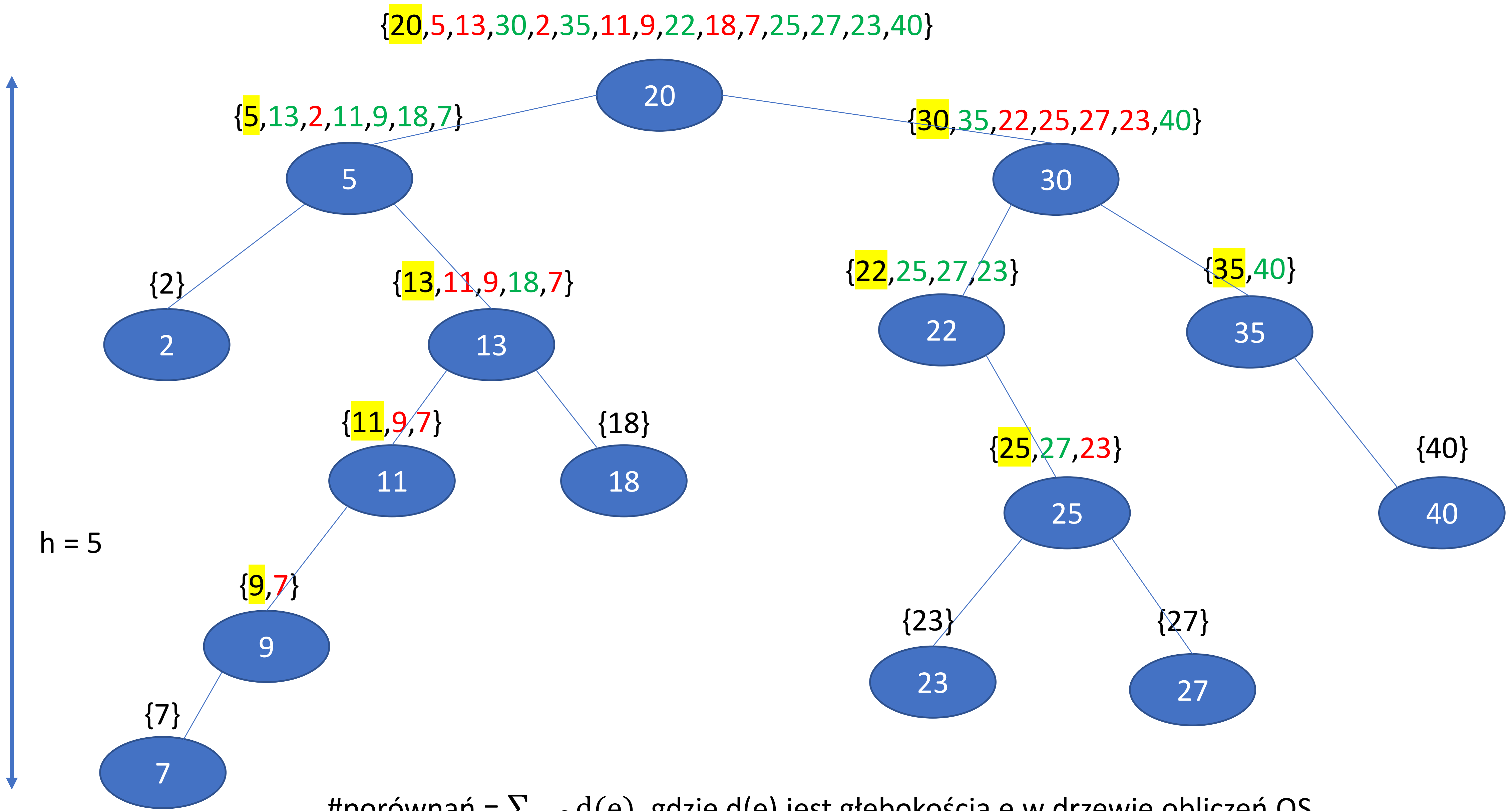
```
QS(S)::
{ S – skończony podzbiór uniwersum z liniowym porządkiem (U,≤)
                              QS
      uporządkowany rosnąco ciąg elementów z S }
begin
  if |S| ≤ 1 then
    output e ∈ S
  else
  begin
    x := Pivot(S); {element dzielący}
    S′ := {x′ ∈ S: x′ < x}; S″ := {x″ ∈ S: x″ > x};
    QS(S′); output x; QS(S″)
  end
end
```

Przykład

W przykładzie na następnych slajdach ilustrujemy działanie procedury QS dla przykładowego ciągu (zbioru)

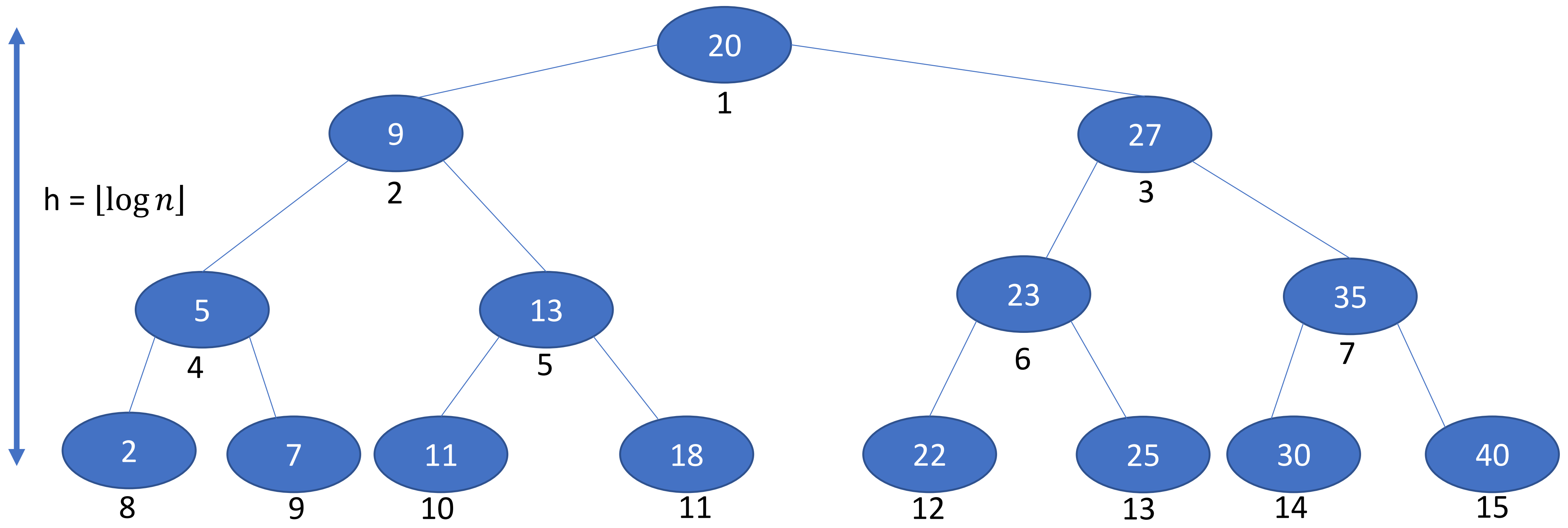
S = {20,5,13,30,2,35,11,9,22,18,7,25,27,23,40}

Na potrzeby przykładu przyjmujemy, że elementem dzielącym jest zawsze pierwszy element ciągu (w zbiorze) – żółte tło. Kolejność elementów w podzbiorach S’, S” jest taka sama jak w S. Elementy podzbioru S’ są zapisane czcionką czerwoną, natomiast elementy podzbioru S” są zapisane czcionką zieloną.



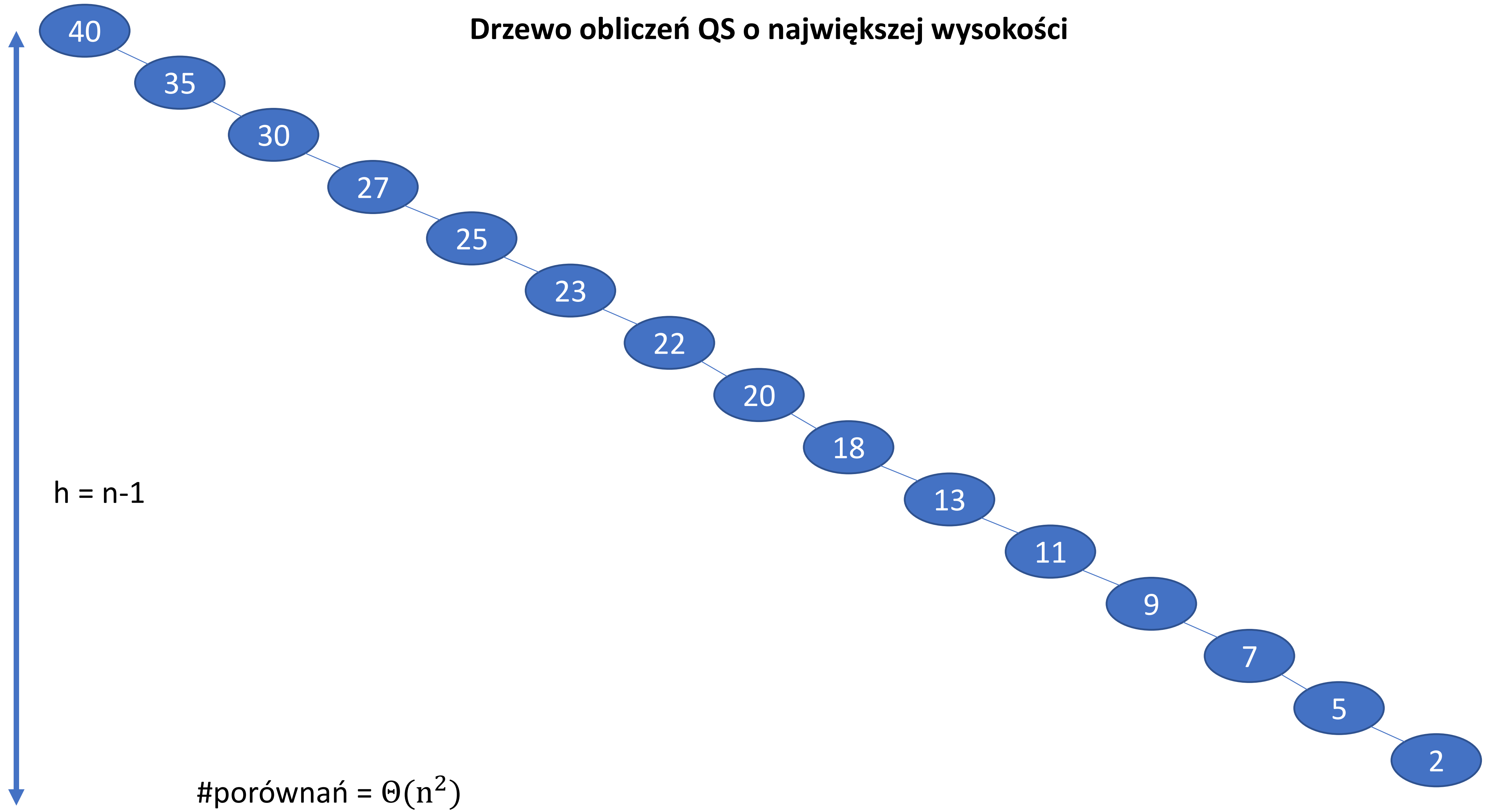
#porównań = $\sum_{e \in S} d(e)$, gdzie d(e) jest głębokością e w drzewie obliczeń QS

Drzewo obliczeń QS o najmniejszej wysokości



#porównań = $\sum_{i=1}^{n} \lfloor \log i \rfloor = (n+1)\lfloor \log(n+1) \rfloor - 2^{\lfloor \log(n+1) \rfloor + 1} + 2 = n\log n + O(n)$

Drzewo obliczeń QS o największej wysokości



#porównań = $\Theta(n^2)$

Dlaczego Quick Sort jest szybki?

Przyjmijmy, że wybór Pivot(S) elementu dzielącego dokonuje się losowo i tak, że wynikiem z jednakowym prawdopodobieństwem $1/|S|$ jest każdy element z S.

Załóżmy, że z pomocą QS z losowym Pivot sortujemy zbiór $S = \{e_1 < e_2 \ldots < e_n\}$. Niech wartością X_n (zmienna losowa) będzie liczba porównań wykonanych w wyniku wywołania QS(S).

Ile wynosi wartość oczekiwana $E[X_n]$?

Niech $X_{i,j}$ będzie zmienną losową, której wartością jest 1, gdy elementy e_i oraz e_j są ze sobą porównywane, natomiast 0, w przeciwnym przypadku, $1 \le i < j \le n$.

$$X_{i,j} = \begin{cases} 0 & e_i \text{ nie jest porównywane z } e_j \\ 1 & e_i \text{ jest porównywane z } e_j \end{cases}$$

$$\Pr(e_i \text{ jest porównywane z } e_j) = 2/(j-i+1).$$

Mamy

$$E[X_n] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{i,j}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 2/(\mathrm{j-i+1}) = \sum_{i=1}^{n-1} \sum_{j=1}^{n-i} 2/(\mathrm{j+1}) =$$

$$= 2\sum_{i=1}^{n-1}\left(\frac{1}{2}+\frac{3}{3}+\cdots+\frac{1}{n-i+1}\right) = 2\sum_{i=1}^{n}(H_i - 1) = 2\sum_{i=1}^{n} H_i - 2n = 2(n+1)H_n - 4n$$

Wiemy

$$H_n \le \ln n + 1$$

Zatem

$$E[X_n] \le 2(n+1)(\ln n + 1) - 4n = 2n\ln n + O(n) = \frac{\mathbf{2}}{\mathbf{log}\,\boldsymbol{e}}\mathbf{nlog}\,\boldsymbol{n} + \boldsymbol{O}(\boldsymbol{n})$$

$$\frac{\mathbf{2}}{\mathbf{log}\,\boldsymbol{e}} \approx \mathbf{1.4}$$

Wykład opracowano między innymi na podstawie książek:

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Cliford Stein, Wprowadzenie do algorytmów, PWN 2012
- Lech Banachowski, Krzysztof Diks, Wojciech Rytter, Algorytmy i struktury danych, PWN 2018
- Donald E. Knuth, Sztuka programowania, Tom 3: Sortowanie i wyszukiwanie, WNT 2002