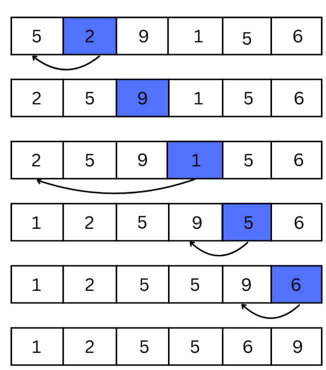
Algorytmy i Struktury Danych

Wojciech Typer

Algorithm 1 Insertion Sort

```
procedure InsertionSort(A, n)
    for i = 1 to n − 1 do
        key = A[i]
        j = i − 1
        while j ≥ 0 and A[j] > key do
            A[j + 1] = A[j]
            j = j − 1
        end while
        A[j + 1] = key
    end for
end procedure
```

Złożoność czasowa: $O(n^2)$
Best case: w najlepszym przypadku złożoność czasowa będzie wynosić $O(n)$
Złożoność pamięciowa: $O(1)$



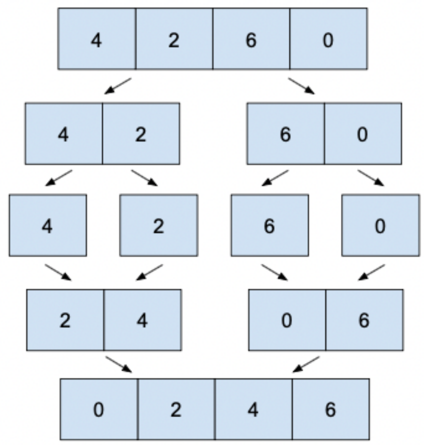
Algorithm 2 Merge Sort

procedure MERGESORT(A, 1, n)
    if |A[1..n]| == 1 **then**
        return A[1..n]
    else
        $B = \text{MergeSort}(A, 1, \lfloor n/2 \rfloor)$
        $C = \text{MergeSort}(A, \lfloor n/2 \rfloor, n)$
        return Merge(B, C)
    end if
end procedure

Algorithm 3 Merge

procedure MERGE(X[1..k], Y[1..n])
    if $X = \emptyset$ **then**
        return Y
    else if $Y = \emptyset$ **then**
        return X
    else if $X[1] \leq Y[1]$ **then**
        return $[X[1]] \times \text{Merge}(X[2..k], Y[1..n])$
    else
        return $[Y[1]] \times \text{Merge}(X[1..k], Y[2..n])$
    end if
end procedure

Złożoność czesowa Merge Sort: $O(n \log n)$
Złożoność pamięciowa Merge Sort: $O(n)$



Istnieje również iteracyjna wersja algorytmu Merge, sort, która została przedstawiona poniżej w postaci pseudokodu.

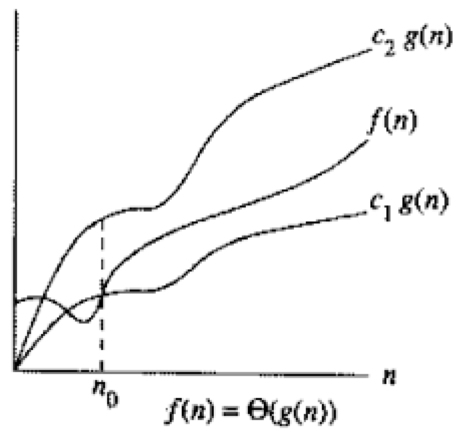
Algorithm 4 IterativeMergeSort

```
procedure ITERATIVEMERGESORT(A[1..n])
    for size = 1 to n − 1 by size × 2 do
        for left = 0 to n − 1 by 2 × size do
            mid ← min(left + size − 1, n − 1)
            right ← min(left + 2 × size − 1, n − 1)
            MERGE(A, left, mid, right)
        end for
    end for
end procedure
```

Złożoność czasowa Iterative Merge Sort: $O(n \log n)$ - dzieje się tak, ponieważ size jest podwajany o 2 w każdej iteracji, więc potrzebujemy około $\log_2 n$ iteracji, a w każdej z nich wykonujemy $O(n)$ operacji.

Złożoność pamięciowa Iterative Merge Sort: $O(n)$

Notacja asymptotyczna O:f(n) = O(g(n)) $\rightarrow (\exists c > 0)(\exists n_0 \in N) : \forall n \geq n_0 \rightarrow 0 \leq f(n) \leq c \cdot g(n)$



$f(n) = O(g(n)) \rightarrow lim_{n\rightarrow\infty} \frac{|f(n)|}{|g(n)|} < \infty$

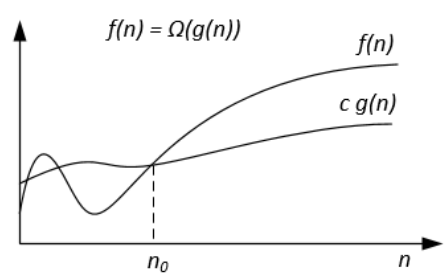
Notacja asymptotyczna - własności

a) $f(n) = n^3 + O(n^2) \rightarrow (\exists h(n) = O(n^2))(f(n) = n^3 + h(n))$

b) $n^2 + O(n) = O(n^2) \rightarrow (\forall f(n) = O(n))(\exists h(n) = O(n^2))(n^2 + f(n) - h(n))$

Notacja Ω

$f(n) = \Omega(g(n)) \rightarrow (\exists c > 0)(\exists n_0 \in N)(\forall n \geq n_0)(c * g(n) \leq |f(n))$



Notacja Ω - własności

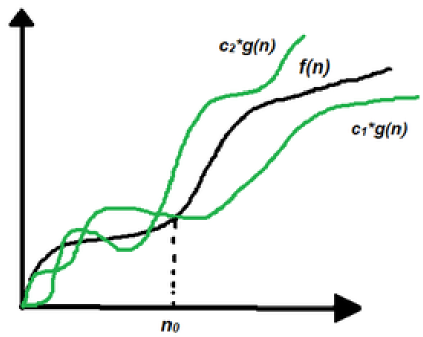
a) $n^3 = \Omega(2n^2)$

b) $n = \Omega(log(n))$

c) $2n^2 = \Omega(n^2)$

Notacja Θ

$f(n) = \Theta(g(n)) \rightarrow (\exists c_1, c_2 > 0)(\exists n_0 \in N)(\forall n \geq n_0)(c_1 g(n) \leq |f(n) \leq c_2 g(n))$



$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

Notacja o- małe

$f(n) = o(g(n)) \rightarrow (\forall c > 0)(\exists n_0 \in N)(\forall n \geq n_0)(|f(n)| < c * |g(n)|)$

Notacja o- małe - przykłady

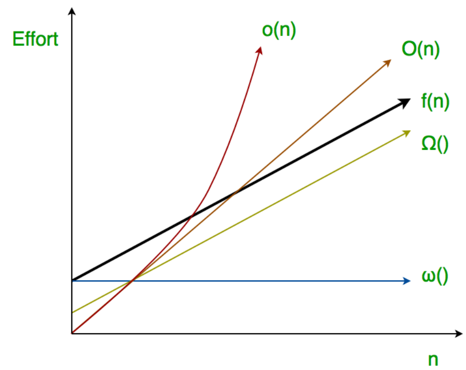
a) $117nlog(n) = o(n^2)$

b) $n^2 = o(n^3)$

Notacja ω

$f(n) = \omega(g(n)) \rightarrow (\forall c > 0)(\exists n_0 \in N)(\forall n \geq n_0)(|f(n)| > c * |g(n)|)$

$lim_{n\rightarrow\infty} \frac{|f(n)|}{|g(n)|} = \infty$



Rekurencje

Metoda podstawiania (metoda dowodzenia indukcyjnego)

1. Zgadnij odpowiedź (bez stałych)
2. Sprawdź przez indukcję, czy dobrze zgadliśmy
3. Znajdź stałe

Przykład 1:

$T(n) = 4T(\frac{n}{2}) + n$
Pierwszy strzał: $T(n) = O(n^3)$
Cel: pokazać, że $(\exists c > 0) T(n) \leq c * n^3$
Krok początkowy: $T(1) = \Theta(1) = c * 1^3 = c$
Krok indukcyjny: zał. że, $(\forall_(k < n))(T(k) \leq c * k^3) =$
Dowód: $T(n) = 4T(\frac{n}{2}) + n \leq 4c * (\frac{n}{2})^3 + n = \frac{1}{2}cn^3 + n =$
$= cn^3 - \frac{1}{2}cn^3 + n = cn^3 - (\frac{1}{2}cn^3 - n) \leq cn^3$
Pokazaliśmy, że $T(n) = O(n^2)$

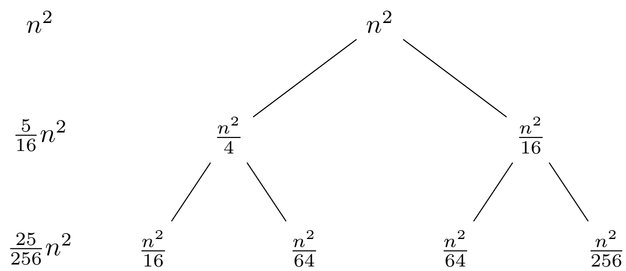
Spróbujmy wzmocnić zał. indukcyjne: $T(n) \leq c_1n^2 - c_2n$
$T(n) \leq 4T(\frac{n}{2}) + n \leq 4(c_1(\frac{n}{2})^2 - c_2\frac{n}{2}) + n =$
$= c_1n^2 - 2c_2n + n = c_1n^2 - (2c_2 - 1)n \leq c_1n^2 - c_2n$
Musimy dobrać takie $c_1 i c_2$, aby $2c_1 \geq c_2$
Wówczas otrzymamy $T(1) = O(1) \leq c_11^2 - c_21$

Przykład 2:

$T(n) = 2T(\sqrt{n}) + log(n)$
Załóżmy, że n jest potęgą dwójki $n = 2^m \rightarrow m = log(n)$
$T(2^m) = 2T(2^{m/2}) + m$
oznaczmy $T(2^m) = S(m)$
$T(2^m) = 2T(2^{m/2}) + m \rightarrow 2S(m/2) + m$
$S(m) = O(mlog(m))$
$T(n) = O(log(n)log(log(n)))$ (formalnie powinniśmy to udowodnić)

Drzewo rekursji

Przykład : $T(n) = T(\frac{n}{2}) + T(\frac{n}{4}) + n^2$



Trzeba pamiętać, że drzewo rekursji samo w sobie nie jest formalnym rozwiązaniem problemu. Nie można go urzywać do dowodzenia złożoności algorytmów. Jest to jedynie intuicyjne podejście do problemu. Formmalnie T(n) należałoby policzyć jako sumę wszystkich wierzchołków w drzewie rekursji:

$$T(n) = \sum_{k=0}^{\infty} \left(\frac{5}{16}\right)^k \cdot n^2 = n^2 \sum_{k=0}^{\infty} \left(\frac{5}{16}\right)^k = n^2 \frac{1}{1 - \frac{5}{16}} = n^2 \frac{16}{11} = \frac{16}{11} n^2$$

Widzimy zatem, że $T(n) = O(n^2)$

Master Theorem

Niech $a \geq 1, b > 1, f(n), d \in N$ oraz $f(n)$ będzie funkcją nieujemną. Rozważmy rekurencję:

$$T(n) = aT(\frac{a}{b}) + \Theta(n^d)$$

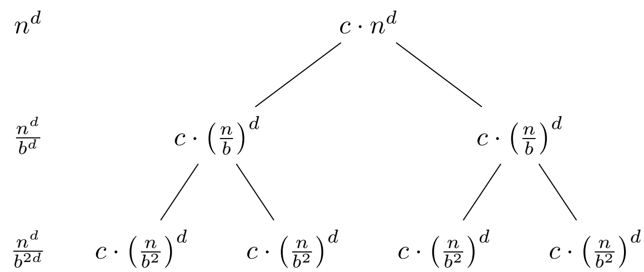
Wówczas:

- $\Theta(n^d)$, jeśli $d > log_b a$

- $\Theta(n^d log(n))$, jeśli $d = log_b a$
- $\Theta(n^{log_b a})$, jeśli $d < log_b a$

Do przedstawienia problemu użyjemy drzewa rekursji. Rozważmy rekurencję:

$$T(n) = aT(\frac{n}{b}) + \Theta(n^d)$$



1. suma kosztoów w k–tym kroku
$$a^k c(\frac{n}{b^k})^d = c(\frac{a}{b^d})^k n^d$$
gdzie $c(\frac{n}{b^k})^d$ to koszt jednego podproblemu w k–tym kroku
2. obliczenie wysokości drzewa:
$$\frac{n}{b^h} = 1 \to h = \log_b n$$
3. Obliczenie $T(n)$
$$\begin{aligned} T(n) &= \Theta\left(\sum_{k=0}^{\log_b n} c\frac{a}{b^k} n^d\right) \\ &= \Theta\left(c \cdot n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right) \\ &= \Theta\left(c \cdot n^d \frac{1 - \left(\frac{a}{b^d}\right)^{\log_b n+1}}{1 - \frac{a}{b^d}}\right) \\ &\implies T(n) = \Theta(n^d) \end{aligned}$$
4. rozważmy 3 przypadki:
   (a) $d > \log_b a$
   $$T(n) = \Theta(n^d)$$
   (b) $d = \log_b a$
   $$T(n) = \Theta(n^d \log n)$$
   (c) $d < \log_b a$
   $$T(n) = \Theta(n^{\log_b a})$$

Przykłady

- $T(n) = 4T(\frac{n}{2}) + 11n$
  Wtedy kożystając z **Master Theorem** mamy:
  $$a = 4, b = 2, d = 1$$
  Jak i również
  $$\log_b a = \log_2 4 = 2 > 1 = d \implies T(n) = \Theta(n^2)$$
- $T(n) = 4T(\frac{n}{3}) + 3n^2$
  Wtedy
  $$a = 4, b = 3, d = 2$$
  Jak i również
  $$\log_b a = \log_3 4 < 2 = d \implies T(n) = \Theta(n^2)$$
- $T(n) = 27T(\frac{n}{3}) + \frac{n^2}{3}$
  Wtedy
  $$a = 27, b = 3, d = 2$$
  Jak i również
  $$\log_b a = \log_3 27 = 3 > 2 = d \implies T(n) = \Theta(n^3 \log n)$$

Metoda dziel i zwyciężaj (D&C)

Na czym ona polega?

1. Podział problemu na mniejsze podproblemy
2. Rozwiazanie rekurencyjnie mniejsze podpoblemy
3. połącz rozwiązania podproblemów w celu rozwiązania problemu wejściowego

Algorytm – Binary Search

- **Input**: posortowania tablica `A[1..n]` oraz element `x`
- **Output**: indeks `i` taki, że `A[i] = x` lub `0` jeśli `x` nie występuje w `A`
- przebieg algorytmu:

Algorithm 5 Binary Search

```
procedure BinarySearch(A, x)
    l = 1
    r = |A|
    while l ≤ r do
        m = ⌊(l+r)/2⌋
        if A[m] = x then
            return m
        else if A[m] < x then
            l = m + 1
        else
            r = m − 1
        end if
    end while
    return 0
end procedure
```

- **Asypmtotyka** Algorytm spełnia następująca rekurencje:

$$T(n) = T(\frac{n}{2}) + \Theta(1)$$

Rozwiązując za pomocą **Master Theorem** otrzymujemy:

$$T(n) = \Theta(\log n)$$

Divide & Conquer

Problem: Obliczenie x^n.

Rozwiązanie naiwną metodą iteracyjną:

$$x^n = x \cdot x \cdot \dots \cdot x \quad \Rightarrow \quad \Theta(n)$$

Rozwiązanie za pomocą Divide & Conquer:

$$x^n = \begin{cases} (x^{\frac{n}{2}}) \cdot (x^{\frac{n}{2}}), & \text{gdy } n \text{ jest parzyste} \\ (x^{\frac{n-1}{2}}) \cdot (x^{\frac{n-1}{2}}) \cdot x, & \text{gdy } n \text{ jest nieparzyste} \end{cases}$$

Rekurencyjna złożoność czasowa:

$$T(n) = T(n/2) + \Theta(1) = \Theta(\log n)$$

—

Problem: Obliczenie n-tej liczby Fibonacciego

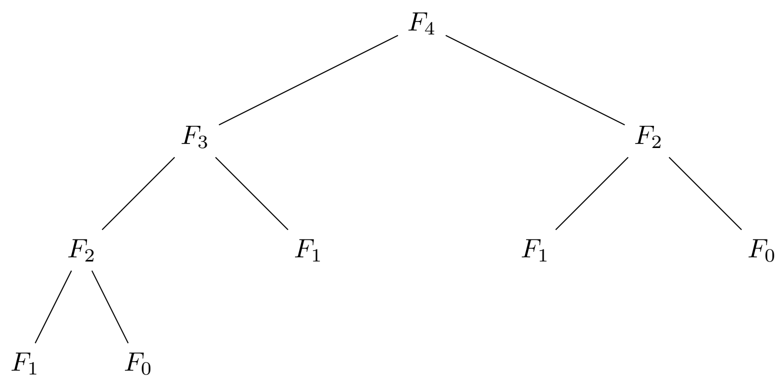
Metoda rekurencyjna:

$$F(n) = F(n-1) + F(n-2)$$

Ma ona złożoność wykładniczą:

$$\Theta(\phi^n), \quad \text{gdzie } \phi = \frac{1+\sqrt{5}}{2}$$

Drzewo rekurencyjne dla F_4:



Wzór jawny:

$$F_n = \frac{1}{\sqrt{5}} \left(\phi^n - (-\phi)^{-n}\right)$$

Obliczanie F_n macierzą: Zamiast rekurencji można użyć potęgowania macierzy, co daje optymalną złożoność. Dla każdego $n \geq 0$ zachodzi:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Potęgowanie macierzy metodą szybkiego potęgowania daje czas:

$$\Theta(\log n)$$

co jest znaczną poprawą w porównaniu do wykładniczej rekurencji.

Mnożenie liczb binarnych metodą Divide & Conquer

Wejście: x, y **Wyjście:** $x \cdot y$

Każdą liczbę można rozbić na dwie połowy:

$$x = x_L \cdot 2^{\frac{n}{2}} + x_R$$

$$y = y_L \cdot 2^{\frac{n}{2}} + y_R$$

Podstawiając do iloczynu:

$$xy = (x_L \cdot 2^{\frac{n}{2}} + x_R) \cdot (y_L \cdot 2^{\frac{n}{2}} + y_R)$$

Po rozwinięciu:

$$xy = x_L y_L \cdot 2^n + (x_L y_R + x_R y_L) \cdot 2^{\frac{n}{2}} + x_R y_R$$

Rekurencyjna zależność czasowa:

$$T(n) = 4T(n/2) + \Theta(n)$$

Zastosowanie **Master Theorem** daje:

$$T(n) = \Theta(n^2)$$

co pokazuje, że metoda ta nie poprawia złożoności względem standardowego mnożenia.

Optymalizacja: metoda Gaussa
Zamiast wykonywać 4 mnożenia rekursywne, można zastosować zasadę Gaussa:

$$xy = x_L y_L \cdot 2^n + ((x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R) \cdot 2^{\frac{n}{2}} + x_R y_R$$

Dzięki temu zamiast 4 mnożeń wykonujemy tylko 3:

$$T(n) = 3T(\frac{n}{2}) + \Theta(n)$$

Zastosowanie **Master Theorem** daje:

$$T(n) = \Theta(n^{\log_2 3})$$

Algorithm 6 Multiply - Mnożenie dużych liczb binarnych metodą Gaussa

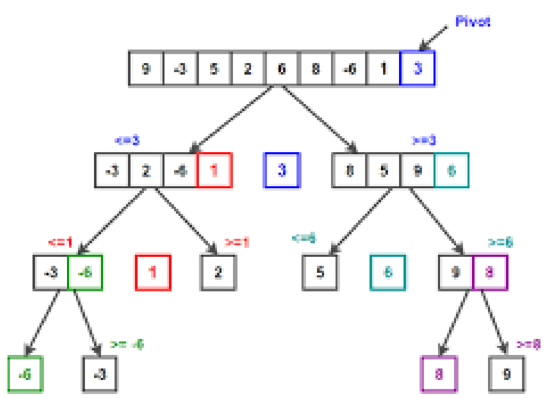
procedure MULTIPLY(x, y)
    $n \leftarrow \max(|x|, |y|)$
    if $n = 1$ **then**
        return $x \cdot y$
    end if
    $m \leftarrow \lceil n/2 \rceil$
    $x_L, x_R \leftarrow$
    $y_L, y_R \leftarrow$
    $p_1 \leftarrow$ MULTIPLY(x_L, y_L)
    $p_2 \leftarrow$ MULTIPLY(x_R, y_R)
    $p_3 \leftarrow$ MULTIPLY$((x_L + x_R), (y_L + y_R))$
    return $p_1 \cdot 2^{2m} + (p_3 - p_1 - p_2) \cdot 2^m + p_2$
end procedure

QuickSort

Algorithm 7 QuickSort - Sortowanie szybkie

```
procedure QUICKSORT(A, low, high)
    if low < high then
        p ← PARTITION(A, low, high)
        QUICKSORT(A, low, p - 1)
        QUICKSORT(A, p + 1, high)
    end if
end procedure

procedure PARTITION(A, low, high)
    pivot ← A[high]
    i ← low − 1
    for j ← low to high − 1 do
        if A[j] ≤ pivot then
            i ← i + 1
            SWAP(A[i], A[j])
        end if
    end for
    SWAP(A[i + 1], A[high])
    return i + 1
end procedure
```



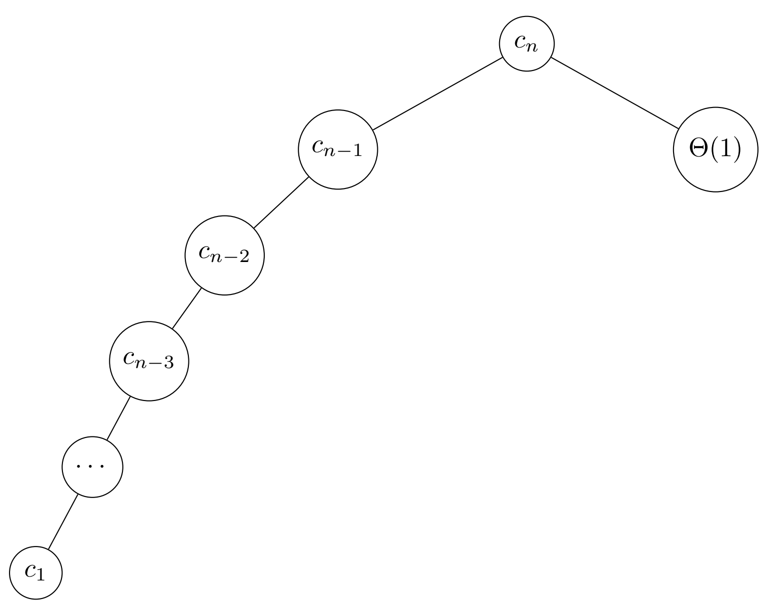
Algorithm 8 Hoare Partition

```
procedure HOARE_PARTITION(A, p, q)
    pivot ← A[⌊(p+q)/2⌋]
    i ← p − 1
    j ← q + 1
    while true do
        repeat
            i ← i + 1
        until A[i] ≥ pivot
        repeat
            j ← j − 1
        until A[j] ≤ pivot
        if i ≥ j then
            return j
        end if
        swap(A[i], A[j])
    end while
end procedure
```

Analiza worst-case QuickSorta

$T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$

Drzewo rekurencji (dla przypadku pesymistycznego, tj. jednostronny podział):



$T(n) \leq \sum_{i=1}^{n} c \cdot i = c \cdot \sum_{i=1}^{n} i = \Theta(n^2)$

Analiza best-case
Jeśli pivot zawsze dzieli tablicę na dwie równe części:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow T(n) = \Theta(n \log n)$$

Analiza average-case
Niech T_n oznacza liczbę porównań dla tablicy długości n.

$$x_k = \begin{cases} 1, & \text{jeśli partition dzieli tablicę na } (k,\ n-k-1) \\ 0, & \text{w przeciwnym wypadku} \end{cases}$$

$$T_n = \sum_{k=0}^{n-1} x_k \cdot (T_k + T_{n-k-1}) + (n-1)$$

Liczymy wartość oczekiwaną:

$$E(T_n) = \sum_{k=0}^{n-1} \mathbb{E}(x_k) \cdot (\mathbb{E}(T_k) + \mathbb{E}(T_{n-k-1})) + (n-1)$$

$$\mathbb{E}(x_k) = \frac{1}{n} \quad \text{(bo pivot jest losowy)}$$

$$E(T_n) = \frac{1}{n} \sum_{k=0}^{n-1} (E(T_k) + E(T_{n-k-1})) + (n-1)$$

$$= \frac{2}{n} \sum_{k=0}^{n-1} E(T_k) + (n-1)$$

$$\Rightarrow E(T_n) = \Theta(n \log n)$$

Analiza avg Case'a $T_n \rightarrow$ Liczba porównań elementów sortowanej tablicy:
|A| = n

$$x_k = \begin{cases} 1, & \text{jeśli partition dzieli tablicę na } (k, n-k-1) \\ 0, & \text{w przeciwnym wypadku} \end{cases}$$

$$T_n = \begin{cases} T_0 + T_{n-1} + n - 1, gdy(0, n-1) - split \\ T_1 + T_{n-2} + n - 1, gdy(1, n-2) - split \\ ... \\ T_k + T_{n-1-k} + n - 1, gdy(k, n-k-1) - split \\ ... \\ T_{n-1} + T_0 + n - 1, gdy(n-1, o) - split \end{cases}$$

$T_n = \sum_{k=0}^{n-1} x_k(T_k + T_{n-k-1}) + n - 1$

liczymy wartosć oczekiwaną:

$E(T_n) = E(\sum k = 0^{n-1} X_k(T_k + T_{n-k-1} + n - 1))$
$E(T_n) = \sum_{k=0}^{n-1} E(X_k \cdot (T_k + T_{n-k-1}) + n - 1)$
$E(T_n) = \sum_{k=0}^{n-1} E(X_k) - E(T_k + T_{n-k-1} - n - 1)$
$E(T_n) = \frac{1}{n} \cdot \sum_{k=0}^{n-1} E(T_k) + \sum(E(T_{n-k-1}))$

Dual pivot quicksort

< LP	LP	LP ≤ & ≤ RP	RP	RP <

Wartość oczekiwana:
E(liczba porównań w dual pivot partition) $\approx \frac{16}{9}n$
E(liczba porównań w dual pivot qs sedwick) $\approx \frac{32}{15}nlogn$

Yaroslavsky dual pivot qs
E(liczba porównań w partition) $\approx \frac{19}{12}n$
E(liczba porównań w Dual Pivot qs Yaroslavsky) $\approx 1.9nlogn$

Strategia count
E(liczba porównań w Count Partition) $\approx \frac{3}{2}n$
E(liczba porównań w Dual Pivot qs z count) $\approx 1.8nlogn$

Comparsion Model
Dolne ograniczenie na liczbę porównań w problemie sortowania
w Comparsion Modelwynosi $\Omega(nlogn)$

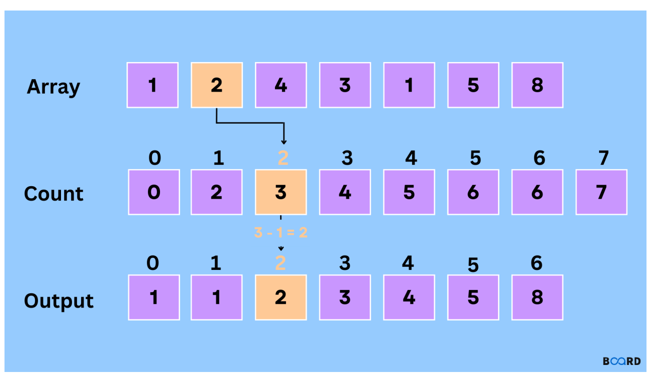
D-d:

- dla dowolnego algorytmu sortującego możemy znaleźć odpowiadające mu drzewo decyzyjne
- n! liści w binarnym drzewie decyzyjnym
- drzewo binarne pełne o wysokości h ma co najmniej 2^h liści
- ale liści w drzewie decyzyjnym powinno być co najmniej n!, zatem:
  $2^h \leq n!$ / lg
  $h \leq \log_2 n!$
  $lgn! = lg(\sqrt{s\pi n}(\frac{n}{e})^n(1 + o(1)))$
  $lg(\frac{n}{e})^n + lg(\sqrt{(}2\pi n)(1 + o(1)))$
  $nlogn - nlge + lg(\sqrt{2\pi n}(2 + o(1))) = \Omega(nlogn)$

Sortowanie:
Input: $|a| = n, \forall i \in \{1, ..., k\}$
Output: posortowana rosnąco tablica A

Algorithm 9 CountingSort

procedure COUNTING_SORT(A, n, k)
    for $i = 1$ **to** k **do**
        $C[i] \leftarrow 0$
    end for
    for $i = 1$ **to** n **do**
        $C[A[i]] \leftarrow C[A[i]] + 1$
    end for
    for $i = 2$ **to** k **do**
        $C[i] \leftarrow C[i] + C[i-1]$
    end for
    for $i = n$ **downto** 1 **do**
        $B[C[A[i]]] \leftarrow A[i]$
        $C[A[i]] \leftarrow C[A[i]] - 1$
    end for
    return B
end procedure



Złożoność obliczeniowa Counting Sorta:
$\Theta(n + k)$ gdzie $k = O(n)$

Stable Sorting Property
Algorytm zachowuje kolejność równych sobie elementów z tablicy wejściowej

RadixSort

Algorithm 10 RadixSort

---

procedure RADIX_SORT(A, n, d)
    for $i = 1$ **to** d **do**
        $counting\_sort(A, n, 9)$
    end for
    return A
end procedure

Złożoność obliczeniowa RadixSorta

- n liczb b'bitowych
- liczb b bitowych dzielimy na r-bitowe cyfry
- cyfry są z $|0, ..., 2^n - 1| = 2^n$
- Counting Sort sortujący n liczb względem jednej cyfry

Zatem RadixSort będzie miał złożoność oblicznieiową:
$\Theta(\frac{b}{r} \cdot (n + 2^r))$
Co po wykonaniu skomplikowanej analizy daje:
$\Theta(d \cdot n)$

Statystyki pozycyjne

Def: k-tą statystyką pozycyjną nazywam← k-tą najmniejszą wartość z zadanego zbioru
przykład:

- $k = 1 \rightarrow O(n)$
- $k = n \rightarrow O(n)$
- $k = \lfloor \rightarrow$ sortowanie $O(n \log n)$

Algorithm 11 RandomSelect

```
procedure RANDOM_SELECT(A, p, q, i)
    if p = q then
        return A[p]
    end if
    r ← RandPartition(A, p, q)
    k ← r − p + 1
    if i = k then
        return A[r]
    else if i < k then
        return RANDOM_SELECT(A, p, r − 1, i)
    else
        return RANDOM_SELECT(A, r + 1, q, i − k)
    end if
end procedure
```

Select algorithm

- dzielimy A[p..q] na $\frac{n}{\lfloor 5 \rfloor}$ pięcioelementowych częsci oraz ostanią część na $\leq$ 5 elementów
- Sortujemy te grupy i wybieramy z każdej z nich medianę
- Znajdujemy medianę M. Select(M, 1, $\frac{n}{5}$, $\frac{n}{10}$)
- Ustalamy X jako pivot; Partition(A, p, q) i tak samo jak w RandomSelect

Select
Select(A, K) $\rightarrow$ T(n)

- Dziel na 5 elementowe tablice i znajdź ich medianę $\rightarrow \Theta(n)$
- Select (...) $\rightarrow$ znajdź medianę median $\rightarrow T(\lceil \frac{n}{5} \lceil)$
- Użyj mediany median jako pivot w Partition $\rightarrow \Theta(n)$
- Idź do lewej albo prawej podtablicy w zależności od indeksu pivota i szukaj statystyki pozycyjnej

Otrzymujemy: $t(n) = T(\lceil \frac{n}{5} \lceil) + \Theta(?)$

Struktury danych
Set interface:

- build (A) - buduje set z danych zawartych w A
- length - zwraca moc zbioru
- find (k) - zwraca element zbioru o kluczu równym k
- insert (k) - dodaje element o kluczu k do zbioru
- delete (k) - usuwa element o kluczu k ze zbioru
- find_max - zwróc element o największym kluczu
- find_min - zwróć element o najmniejszym kluczu
- find_prev - zwraca element poprzedni od klucza

Binary Search Tree
BST property :

- x $\in$ T - x jest węzłem drzewa T
- Wówczas każdy y *in* x.left ma y.key < x.key
- key y *in* x.right ma y.key > x.key

Inorder Tree Walk

Algorithm 12 Inorder Tree Walk

procedure InorderTreeWalk(x $\in$ T)
    if $x \neq$ null **then**
        InorderTreeWalk(x.left)
        print(x)
        InorderTreeWalk(x.right)
    end if
end procedure

Tree Search

Algorithm 13 TreeSearch

```
procedure TreeSearch(x ∈ T, k)
    if x = null ∨ k = x.key then
        return x
    else if k < x.key then
        return TreeSearch(x.left, k)
    else
        return TreeSearch(x.right, k)
    end if
end procedure
```