



Politechnika Poznańska

Informatyka rok I semestr 2

L10, Piątek 0:00 - 0:00

Algorytmy i Struktury Danych

Prowadzący: Dominik Piotr Witczak

Sprawozdanie nr 1

Algorytmy Sortowania

Autor:

Dominik Fischer 164176
Oliwer Miller 163544

Rok akademicki 2024/2025

Wprowadzenie

Tutaj piszemy wprowadzenie

Selection Sort

Zamysłem danego algorytmu jest po kolei wyszukiwanie najmniejszych elementów w tablicy. Na początku pierwszy element jest uznawany za najmniejszy, następnie wyszukiwany jest rzeczywisty najmniejszy element i zamieniany miejscem z początkowym. Wtedy lewa część tablicy staje się częścią posortowaną, a działanie jest kontynuowane do końca tablicy.

Złożoność czasowa algorytmu: $O(n^2)$

Terminal

```
def selection_sort(data):
    n=len(data)
    for j in range(n-1):
        min = j
        for i in range(j+1, n):
            if data[i] < data[min]:
                min = i
        data[j], data[min] = data[min], data[j]
    return data
```

Insertion Sort

Pętla *for* zaczyna iterację od drugiego elementu, ponieważ pierwszy uznaje wstępnie za posortowany. Algorytm porównuje kolejno elementy *key* z elementami w posortowanej części po lewej stronie, następnie, przy użyciu pętli *while*, po kolei zamienia elementy miejscami, dopóki dany element nie znajdzie się na odpowiedniej pozycji.

Złożoność czasowa algorytmu:

Najgorszy przypadek: $O(n^2)$ - lista malejąca

Najlepszy przypadek: $O(n)$ - lista posortowana

Średni przypadek: $O(n^2)$ - losowe dane

Terminal

```
def insertion_sort (data):
    for i in range(1, len(data)):
        key = data[i]
        j = i - 1
        while j >= 0 and data[j] > key:
            data[j + 1] = data[j]
            j -= 1
        data[j + 1] = key
    return data
```

Shell Sort With Sadgewick Gaps

Dwie funkcje są użyte do tego algorytmu. Funkcja *sedgewick gaps* jest odpowiedzialna za stworzenie odstępów używanych do sortowania. Pętla *while* tworzy odstęp w zależności od parzystości zmiennej *k*, które są później zapisywane w liście *gaps*. Funkcja *shell sort* wykorzystuje odstęp zwrócone przez poprzednią funkcję do porównywania i zamiany miejscami elementów ciągu w danych odstępach od siebie.

Złożoność czasowa algorytmu:

Najgorszy przypadek: $O(n^{\frac{3}{2}})$ - lista malejąca

Najlepszy przypadek: $O(n \log n)$ - ciąg prawie posortowany

Średni przypadek: $O(n^{\frac{4}{3}})$ - losowe dane

Terminal

```
def sedgewick_gaps(n):
    gaps = []
    k = 0
    while True:
        if k % 2 == 0:
            gap = 9 * (2 ** k) - 9 * (2 ** (k
                // 2)) + 1
        else:
            gap = 4 ** k + 3 * 2 ** (k - 1) + 1

        if gap >= n:
            break
        gaps.append(gap)
        k += 1

    return gaps[::-1]

def shell_sort(data):
    n = len(data)
    gaps = sedgewick_gaps(n)

    for gap in gaps:
        for i in range(gap, n):
            temp = data[i]
            j = i
            while j >= gap and data[j - gap] >
                temp:
                data[j] = data[j - gap]
                j -= gap
            data[j] = temp
    return data
```

Heap Sort

Algorytm wykorzystuje do działania drzewo max heap, w którym, z zasady, największy element znajduje się na samej górze i rodzice mają większą wartość niż dzieci. Funkcja *heap* jest odpowiedzialna za budowę kopca. Zmienne *left* i *right* to indeksy dzieci.

Elementy kopca są ze sobą porównywane, potem następuje rekurencyjne wywołanie funkcji, aby uporządkować poddrzewa.

Funkcja *heap sort* wykorzystuje funkcję *heap* do konstrukcji drzewa, dodatkowo zamienia korzeń drzewa ze skrajnym liściem, tak ustawia największy element w części posortowanej, następnie wywołuje funkcję *heap* ponownie.

Złożoność czasowa algorytmu: $O(n \log n)$

Terminal

```
def heap(data,n,i):
    largest=i
    left = 2*i+1
    right = 2*i+2

    if left<n and data[left]>data[largest]:
        largest = left
    if right<n and data[right]>data[largest]:
        largest=right
    if largest !=i:
        data[i], data[largest]=data[largest],
        data[i]
        heap(data, n, largest)

def heap_sort(data):
    n = len(data)

    for i in range(n // 2 - 1, -1, -1):
        heap(data, n, i)

    for i in range(n - 1, 0, -1):
        data[i], data[0] = data[0], data[i]
        heap(data, i, 0)

    return data
```

Quick Sort Left Pivot

W tym algorytmie pierwszy element jest wybierany jako pivot. Funkcja *partition* dzieli tablicę, dzięki zmiennym *i* oraz *j* dzieli tablicę na część mniejszą i większą od pivota.

Funkcja *quick sort left pivot* rekurencyjnie sortuje i następnie zwraca podtablicę.

Złożoność czasowa algorytmu:

Najgorszy przypadek: $O(n^2)$

Najlepszy przypadek: $O(n \log n)$

Średni przypadek: $O(n \log n)$

Terminal

```
def partition(A, p, r):
    pivot = A[p]
    i = p+1
    j = r

    while True:
        while i <= j and A[i] <= pivot:
            i += 1
        while i <= j and A[j] > pivot:
            j -= 1
        if i <= j:
            A[i], A[j] = A[j], A[i]
        else:
            break

    A[p], A[j] = A[j], A[p]
    return j

def quick_sort_left_pivot(A, p, r):
    if p < r:
        q = partition(A, p, r)
        quick_sort_left_pivot(A, p, q-1)
        quick_sort_left_pivot(A, q+1, r)
    return A
```



Quick Sort Random Pivot

W tym algorytmie pivot jest wybierany losowo. Dany ciąg jest dzielony na trzy podtablice w zależności od wielkości elementów w porównaniu do pivota. Na końcu następuje rekurencyjne wywołanie funkcji na podtablicach.

Złożoność czasowa algorytmu:

Najgorszy przypadek: $O(n^2)$

Najlepszy przypadek: $O(n \log n)$

Średni przypadek: $O(n \log n)$

Terminal

```
def quick_sort_random_pivot(data):
    if len(data) <= 1:
        return data

    pivot_index = random.randint(0, len(data) - 1)
    pivot = data[pivot_index]

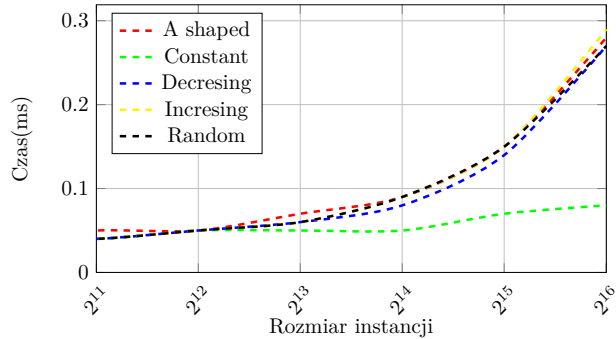
    left = []
    middle = []
    right = []

    for i, x in enumerate(data):
        if i == pivot_index:
            middle.append(x)
        elif x < pivot:
            left.append(x)
        elif x > pivot:
            right.append(x)

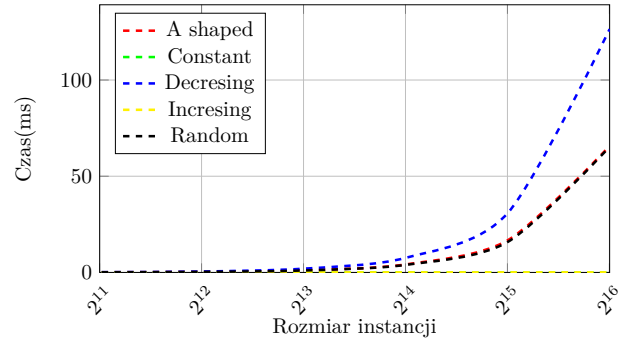
    return quick_sort_random_pivot(left) +
           middle + quick_sort_random_pivot(right)
```

Porównanie czasów wykonania

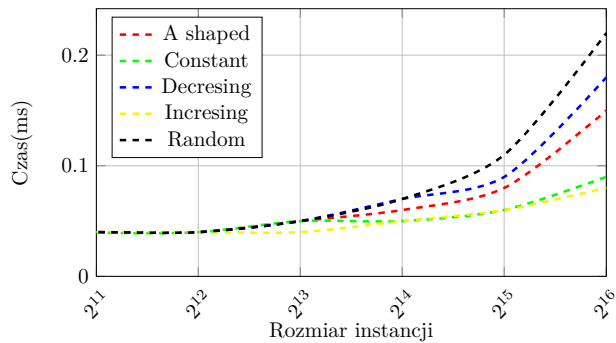
Złożoność Obliczeniowa Algorytmu Heap Sort



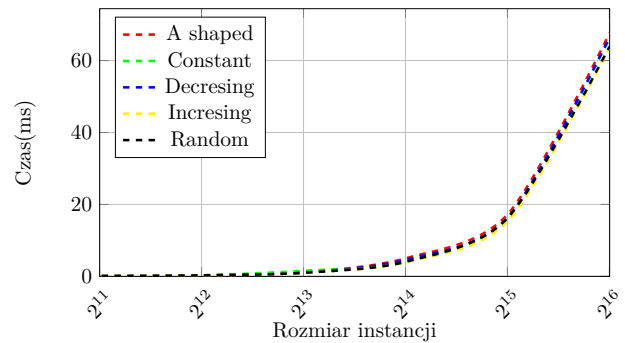
Złożoność Obliczeniowa Algorytmu Insertion Sort



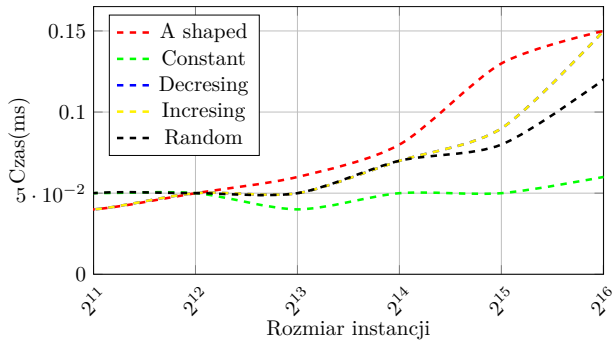
Złożoność Obliczeniowa Algorytmu Shell Sort



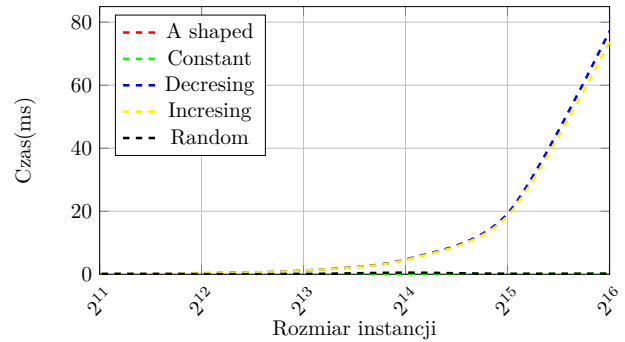
Złożoność Obliczeniowa Algorytmu Selection Sort



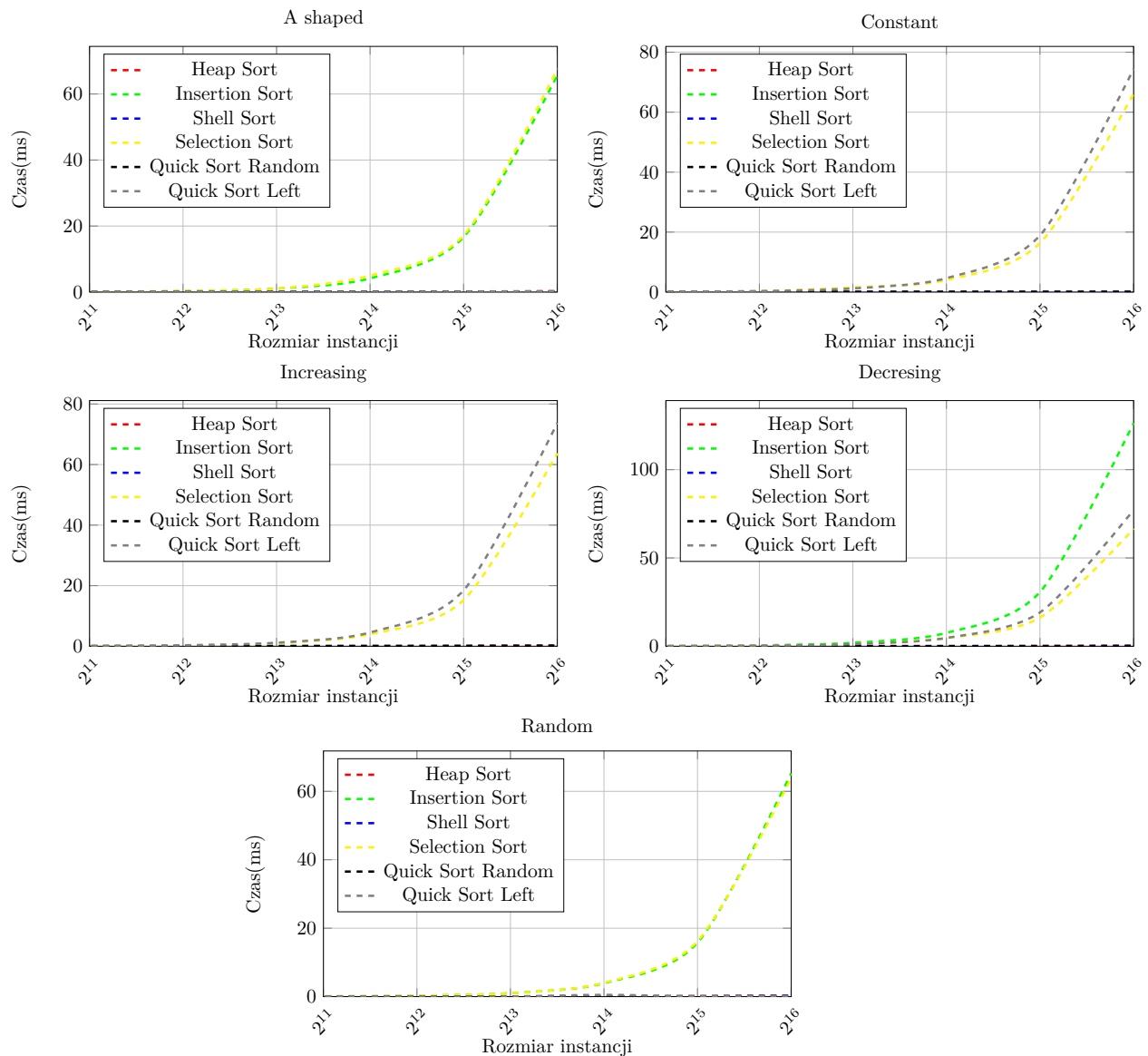
Złożoność Obliczeniowa Algorytmu Quick Sort Random Pivot



Złożoność Obliczeniowa Algorytmu Quick Sort Left Pivot



Porównanie czasów wykonania poszczególnych algorytmów względem danych



Wnioski

Tutaj dajemy wnioski