Jakub Rynio | nr albumu: 117323
Informatyka niestacjonarna
semestr: 3 Algorytmy i złożoność:
Grupa laboratoryjna: 2 Porównanie alagorytmów sortowania (Insertion Sort vs Selection Sort)

Opis algorytmu Insertion Sort

Algorytm **Insertion Sort** to prosty algorytm sortowania, który działa na zasadzie wstawiania elementów w odpowiednie miejsce w już posortowanej części tablicy. Działa podobnie do sposobu sortowania kart w ręce.

1. **Podział zbioru:** Przyjmujemy, że pierwszy element tablicy jest już posortowany.
2. **Przechodzenie przez elementy:** Iterujemy przez kolejne elementy tablicy, zaczynając od drugiego.
3. **Porównywanie i przesuwanie:** Każdy nowy element porównujemy z elementami w posortowanej części tablicy i przesuwamy większe elementy w prawo.
4. **Wstawienie elementu:** Wstawiamy bieżący element w odpowiednie miejsce.

Algorytm zaimplementowany w pythonie

```python
import random
import time

def insertionSort(arr):
    n = len(arr)
    if n <= 1:
        return
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

array_size = int(input("Podaj rozmiar tablicy: "))

arr = [random.randint(0, 1000) for _ in range(array_size)]

print("Tablica przed posortowaniem: ")
print(arr)

start_time = time.time()
insertionSort(arr)
end_time = time.time()

print("\n\nTablica po posortowaniu: ")
print(arr)

print(f"Czas wykonania: {end_time - start_time:.6f} sekund")
input()
```

Złożoność Obliczeniowa

- **Najlepszy przypadek (tablica już posortowana):** $O(n)$

- **Średni przypadek:** $O(n^2)$
- **Najgorszy przypadek (tablica posortowana odwrotnie):** $O(n^2)$
- **Złożoność pamięciowa:** $O(1)$ (algorytm sortowania in-place)

Zalety

- Prosty do zaimplementowania.
- Stabilny (nie zmienia kolejności równych elementów).
- Efektywny dla małych zbiorów danych.

Wady

- Nieefektywny dla dużych zbiorów danych ze względu na złożoność $O(n^2)$.

Algorytm Heap Sort

Algorytm *Heap Sort* to jeden z klasycznych algorytmów sortowania, który wykorzystuje strukturę danych zwaną *kopcem* (heap). Jest to algorytm oparty na podejściu *„dziel i zwyciężaj”* (divide and conquer), który pozwala na efektywne posortowanie zbioru elementów. Jego złożoność czasowa to $O(n \log n)$ w przypadku wszystkich przypadków (średni, najlepszy i najgorszy).

Algorytm zaimplementowany w pythonie

```
import random
import time

def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap

        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

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

```
24            heapify(arr, n, i)
25
26        for i in range(n - 1, 0, -1):
27            arr[i], arr[0] = arr[0], arr[i]  # swap
28            heapify(arr, i, 0)
29
30    def generate_random_array(size, lower_bound=1, upper_bound=1000):
31        return [random.randint(lower_bound, upper_bound) for _ in range(size)]
32
33
34    size = int(input("Podaj wielkosc tablicy: "))
35
36    arr = generate_random_array(size)
37
38    print("Tablica przed posortowaniem:")
39    print(arr)
40
41    start_time = time.time()
42    heapSort(arr)
43
44    end_time = time.time()
45
46    print("\nPosortowana tablica:")
47    print(arr)
48
49    print("\nCzas wykonania algorytmu: {:.6f} sekund".format(end_time - start_time))
50
51    input()
```

Kroki działania algorytmu Heap Sort

1. **Budowa kopca**: Algorytm zaczyna od zbudowania kopca. W przypadku Heap Sort używamy kopca *maksymalnego* (max-heap), czyli struktury, w której dla każdego węzła jego wartość jest większa lub równa wartości jego dzieci.

2. **Heapify**: *Heapify* to operacja, która utrzymuje właściwości kopca, przekształcając poddrzewo, tak by spełniało regułę kopca. Rozpoczynamy od ostatniego węzła, który ma dzieci (czyli od elementu na pozycji $\frac{n}{2}-1$), a następnie wykonujemy operację *heapify* na każdym węźle drzewa, przesuwając większe elementy do góry, aby spełniały regułę kopca maksymalnego.

3. **Sortowanie**: Po zbudowaniu kopca, algorytm rozpoczyna sortowanie. Największy element kopca (korzeń) znajduje się na początku i jest przesuwany na koniec posortowanej tablicy. Następnie wykonuje się operację *heapify* na pozostałej części kopca, by przywrócić jego właściwości. Procedura jest powtarzana, aż pozostała część kopca będzie miała tylko jeden element. Na każdym kroku elementy są przesuwane na koniec tablicy, a kopiec jest naprawiany.

Schemat algorytmu

1. Zbuduj kopiec maksymalny z danych wejściowych.

2. Dla każdego elementu od końca do początku:

    - Zamień korzeń (maksymalny element) z ostatnim elementem kopca.

- Zmniejsz rozmiar kopca.
- Przywróć właściwości kopca (heapify).

Przykład

Dla tablicy wejściowej $[4, 10, 3, 5, 1]$, algorytm działałby następująco:

1. **Budowanie kopca**:
   - Na początku mamy nieposortowaną tablicę: $[4, 10, 3, 5, 1]$.
   - Zbuduj kopiec maksymalny: Po przekształceniu kopiec wygląda jak: $[10, 5, 3, 4, 1]$.
2. **Sortowanie**:
   - Zamień pierwszy (największy) element (10) z ostatnim: $[1, 5, 3, 4, 10]$.
   - Zmniejsz rozmiar kopca o 1, teraz przekształcamy kopiec: $[5, 4, 3, 1, 10]$.
   - Powtarzamy te kroki, aż tablica będzie posortowana: $[1, 3, 4, 5, 10]$.

Złożoność czasowa

- Budowanie kopca: $O(n)$
- Heapify dla każdego elementu: $O(\log n)$
- Zatem cała złożoność algorytmu Heap Sort to $O(n \log n)$, co czyni go wydajnym algorytmem sortującym.

Zastosowania i zalety

- **Stabilność**: Heap Sort *nie jest stabilnym algorytmem*, tzn. nie zachowuje kolejności elementów o równych kluczach.
- **Pamięć**: Heap Sort jest algorytmem *w miejscu* (in-place), co oznacza, że do sortowania nie wymaga dodatkowej pamięci poza tym, co jest potrzebne na przechowywanie samej tablicy.
- Jest wykorzystywany w sytuacjach, gdzie stabilność sortowania nie jest kluczowa, a ważne jest wykorzystanie małej ilości pamięci.

Insertion Sort VS Heap Sort

- **Złożoność czasowa:**
  - Insertion Sort: $O(n^2)$ w najgorszym przypadku, $O(n)$ w najlepszym.
  - Heap Sort: $O(n \log n)$ w każdym przypadku.
- **Stabilność:**
  - Insertion Sort: Stabilny.
  - Heap Sort: Niestabilny.
- **Złożoność przestrzenna:**
  - Insertion Sort: $O(1)$.
  - Heap Sort: $O(1)$.
- **Wydajność dla dużych danych:**
  - Insertion Sort: Powolny dla dużych zbiorów.
  - Heap Sort: Szybszy i bardziej wydajny dla dużych zbiorów.
- **Zastosowanie:**
  - Insertion Sort: Lepszy dla małych lub częściowo posortowanych danych.
  - Heap Sort: Lepszy do sortowania dużych danych, gdy stabilność nie jest wymagana.
- **Implementacja:**
  - Insertion Sort: Prostszy w implementacji.
  - Heap Sort: Bardziej skomplikowany.

Porównanie prędkości dla sortowanie tablicy z losowymi liczbami od 0 do 1000:

liczba elementow tablicy	100	1000	10000	100000
Insertion Sort	0	0.024642	2.938645	242.313858
xHeap Sort	0	0.002026	0.033705	0.532758