“Dengan ini saya menyatakan bahwa TE ini adalah hasil pekerjaan saya sendiri”

Perbandingan Running Time Algoritma Randomized Shell Sort dan Max Heap Sort

1. Deskripsi Singkat dan Penerapan Randomized Shell Sort

Shell sort adalah algoritma sorting hasil modifikasi algoritma insertion sort dimana keduanya membandingkan setiap elemen akan tetapi pada shell sort perbandingan tiap elemen dilakukan pada jarak (*gap*) tertentu. Algoritma Shell sort memiliki beberapa variasi salah satunya adalah randomized shell sort. Pada randomized shell sort terdapat sedikit modifikasi pada penentuan nilai *gap* dimana pada randomized shell sort nilai *gap*-nya berupa nilai yang acak.

Secara umum, prinsip dasar algoritma randomized shell sort adalah membandingkan dan menukar tiap elemen yang memiliki *gap* tertentu. Nilai *gap* ini didapatkan dari fungsi *randomize* sehingga didapatkan nilai acak. Lalu setelah melakukan perbandingan, secara bertahap kita akan menggunakan *gap* hasil *randomize* berikutnya sampai seluruh elemen telah terurut dengan baik. Berikut detail penerapan dari randomized shell sort misalkan dengan *input* = [10 7 9 2 5 1 8 3 ] dengan didapatkan *random* *sequence*-nya yaitu [3 2 1 1]:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 10 | 7 | 9 | 2 | 5 | 1 | 8 | 3 |

Pada iterasi *gap* pertama kita akan memakai *gap* = 3 sehingga kita akan menukar elemen dengan jarak antar elemennya 2 jika elemen kiri lebih besar daripada elemen kanan. Bermula dari mengecek 10 dengan 2. Hal ini berlangsung sekaligus mengecek elemen dengan *gap* sebelumnya misalkan untuk elemen terakhir yaitu 3 akan mengecek 2 elemen karena di belakangnya yang ber-*gap* 3 ada 2 elemen. Hal ini berlaku sama untuk iterasi *gap* berikutnya. Hal ini yang membuat hasil iterasi *gap* pertama sebagai berikut:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 2 | 3 | 1 | 8 | 5 | 9 | 10 | 7 |

Pada iterasi *gap* kedua kita akan memakai *gap* = 2 sehingga kita akan menukar elemen dengan jarak antar elemennya adalah 2. Hal ini yang membuat hasil iterasi *gap* kedua sebagai berikut:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 3 | 2 | 7 | 5 | 8 | 10 | 9 |

Pada iterasi *gap* 3 kita akan memakai *gap* = 1 sehingga kita akan menukar elemen dengan jarak antar elemennya adalah 1. Hal ini yang membuat hasil iterasi *gap* ketiga sebagai berikut:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | 7 | 8 | 9 | 10 |

Pada iterasi *gap* 4 kita akan memakai *gap* = 1 sehingga kita akan menukar elemen dengan jarak antar elemennya adalah 1. Hal ini yang membuat hasil iterasi *gap* keempat sekaligus hasil akhirnya dikarenakan *gap sequence* telah usai sebagai berikut:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 5 | 7 | 8 | 9 | 10 |

1. Pseudocode
2. Pseudocode Randomized Shell Sort

|  |
| --- |
| procedure randomizedShellSort(arr as array):  n = length(arr)  gapSequence = [random.randint(1, n // 2) for \_ in range(n // 2)] // generate collection of random gap    for gap in gapSequence:  for i from gap to n - 1:  temp = arr[i]  j = i    // Shift elements that are greater than temp by gap positions to the right  while j >= gap and arr[j - gap] > temp:  arr[j] = arr[j - gap]  j = j - gap    arr[j] = temp |

1. Pseudocode Max Heap Sort

|  |
| --- |
| Procedure Heapify(A as array, n as int, i as int):  max = i  leftchild = 2i + 1  rightchild = 2i + 2  if (leftchild <= n) and (A[i] < A[leftchild])  max = leftchild  else  max = i  if (rightchild <= n) and (A[max] > A[rightchild])  max = rightchild  if (max != i)  swap(A[i], A[max])  Heapify(A, n, max)  Procedure Heapsort(A as array):  n = length(A)  for i = n/2 downto 1  Heapify(A, n ,i)    for i = n downto 2  exchange A[1] with A[i]  A.heapsize = A.heapsize - 1  Heapify(A, i, 0) |

1. Hasil Eksperimen

Tabel 1: Running time Randomized Shell Sort (ms)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Small Size (29angka) | Medium Size (213angka) | Big Size (216angka) |
|  | 14,9322 | 3842,9081 | 281280,701 |
| Status Sorted | 14,8873 | 3809,4389 | 281948,209 |
|  | 15,0764 | 3836,5707 | 237918,083 |
| **Rata-rata (Sorted)** | **14,9653** | **3829,63923** | **267048,998** |
|  | 14,5855 | 3684,7188 | 240544,236 |
| Status Randomize | 15,3079 | 3724,4348 | 243149,423 |
|  | 14,9250 | 3707,3848 | 238232,328 |
| **Rata-Rata (Randomize)** | **14,9394667** | **3705,5128** | **240641,996** |
|  | 14,2305 | 3725,6799 | 240229,254 |
| Status Reversed | 15,5630 | 3738,3099 | 244546,189 |
|  | 14,0030 | 3749,3048 | 238539,075 |
| **Rata-rata (Reversed)** | **15,3626667** | **3737,76487** | **241104,839** |

Tabel 2: Running time Max Heap Sort (ms)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Small Size (29angka) | Medium Size (213angka) | Big Size (216angka) |
|  | 2,0106 | 44,0512 | 438,4077 |
| Status Sorted | 2,0030 | 42,1977 | 453,7053 |
|  | 2,0270 | 44,1983 | 435,5927 |
| **Rata-rata (Sorted)** | **2,0135** | **43,4824** | **442,568567** |
|  | 1,9016 | 42,1407 | 417,949 |
| Status Randomize | 2,0022 | 40,3576 | 424,0994 |
|  | 2,0013 | 42,1133 | 399,5914 |
| **Rata-Rata (Randomize)** | **1,9683** | **41,5372** | **413,879933** |
|  | 1,9996 | 40,0138 | 392,7622 |
| Status Reversed | 2,0027 | 40,4370 | 380,3189 |
|  | 1,9991 | 41,0011 | 379,8244 |
| **Rata-rata (Reversed)** | **2,0004** | **40,4839** | **384,301833** |

1. Analisis Hasil Eksperimen

Berdasarakan data yang kita dapatkan, kita akan membagi hasil eksperimen ini menjadi 2 bagian yaitu:

1. Perbandingan *Running Time* Berdasarkan Ukuran *Input*

Pada algoritma randomize shell sort terjadi peningkatan *running time* antara *small* dengan *medium size* sekitar 243-256 dan untuk medium dengan *big size* meningkat sekitar 64-70. Lalu untuk *small* dengan *big size* meningkat sekitar 15694-17844. Di lain sisi algoritma max heap sort antara *small* dengan *medium size* *running time*-nya meningkat sekitar 20-21 dan pada *medium* dengan *big size* meningkat sekitar 9-10. Lalu untuk *small* dengan *big size* meningkat sekitar 182-219. Di setiap ukuran *input* juga menunjukkan bahwa max heap sort lebih cepat daripada randomized shell sort.

1. Perbandingan *Running Time* Berdasarkan Keadaan *Input*

Secara umum untuk setiap keadaan *input* juga menunjukkan bahwa max heap sort lebih cepat daripada randomized shell sort. Lalu pada algoritma randomized shell sort saat *input small size*, *running time* terbesar ada pada saat *input reversed* dan *running time* terkecil ada pada saat *input* *random*. Akan tetapi hal ini berubah saat *input* semakin besar dimana saat *input medium size*, *running time* terbesar ada pada saat *input sorted* sedangkan terkecil tetap pada saat *input random*. Hal ini berlaku sama saat memakai input *big size*. Hal ini dapat terjadi pada input *small size* dikarenakan *input* yang berukuran kecil dan sebenarnya selisih waktu antar *input* sangat kecil dimana kurang dari 1 ms sehingga pengaruh keadaan *input* masih tidak terlalu kelihatan. Oleh karena itu, kita mendapatkan *best case* *scenario* saat *input random* dan *worst case* *scenario* saat *input sorted*. Hal ini yang membuat kita mendapatkan *asymptotic notation* dari algoritma randomized shell sort yaitu Ω(saat *input random*) dan O(saat *input sorted*).

Pada algoritma max heap sort saat input *small size*, *running time* terbesar ada pada saat *input sorted* dan *running time* terkecil ada pada saat *input random*. Akan tetapi hal ini berubah saat *input* semakin besar dimana saat *input medium size*, *running time* terbesar tetap pada saat *input sorted* akan tetapi *running time* terkecil menjadi *input reversed*. Hal ini mungkin terjadi dikarenakan pada *small size input* berukuran kecil sehingga tidak terlalu signifikan perbedaan *running time* antar keadaan *input* dan selisih *antara input random* dengan *reversed* sangat keci*l*. Oleh karena itu, kita mendapatkan *best case scenario* saat *input reversed* dan *worst case scenario* saat *input sorted*. Hal ini yang membuat kita mendapatkan *asymptotic notation* dari algoritma max heap sort yaitu Ω(saat *input reversed*) dan O(saat *input sorted*).

1. Kesimpulan

Berdasarkan hasil eksperimen ini dapat kita simpulkan bahwa algoritma max heap sort memiliki kompleksitas waktu lebih kecil di setiap keadaan maupun ukuran input sehingga dapat dikatakan lebih cepat daripada randomized shell sort. Selain itu ukuran *input* sangat mempengaruhi *running time* kedua algoritma tersebut akan tetapi algoritma randomized shell sort lebih sensitif terhadap perubahan ukuran *input* dibandingkan max heap sort dimana lonjakan kenaikannya lebih signifikan dibandingkan max heap sort. Lalu terhadap perubahan keadaan *input* perubahan *running time* cukup signifikan dimana kita mendapatkan *asymptotic notation* untuk randomized shell sort yaitu Ω(saat *input random*) dan O(saat *input sorted*). Kemudian untuk algoritma max heap sort kita mendapatkan *asymptotic notation* dari algoritma max heap sort yaitu Ω(saat *input reversed*) dan O(saat *input sorted*).

1. Referensi

Abishev. (2019, September 14). Heapsort. https://fullyunderstood.com/pseudocodes/heap-sort/

Simplilearn. (2023, Februari 20). Shell Sort Algorithm: Everything You Need to Know. https://www.simplilearn.com/tutorials/data-structure-tutorial/shell-sort

Mittal. (2019, September 14). Shellsort. <https://www.geeksforgeeks.org/shellsort/>

Michael T Goodrich. Randomized Shellsort: A Simple Data-Oblivious Sorting Algorithm. *Journal of the ACM (JACM)*, 58(6):1–26, 2011.

1. Link github

<https://github.com/dapulmh/Tugas-Eksperimen-1-Daffa-MH>