Analisis Perbandingan Penggunaan Memori dan Waktu Eksekusi dari Algoritma Sorting Bidirectional-Conditional-Insertion-Sort dan Counting Sort

Fauzan Firzandy Khifzan – 2106702756

Universitas Indonesia, Faculty of Computer Science, Computer Science, Depok, Indonesia

"Dengan ini saya menyatakan bahwa TE ini adalah hasil pekerjaan saya sendiri"

Abstrak

Kami melakukan analalisis perbandingan antara algoritma *sorting* Bidirecitonal-Conditional-Insertion-Sort (BCIS) dan Counting Sort (CS) untuk mengamati kinerja keduanya dalam hal waktu eksekusi dan penggunaan memori pada dataset dengan variasi ukuran yang berbeda-beda. Dalam hasil uji coba, diketahui bahwa BCIS memiliki kelebihan dalam menggunakan memori yang efisien dan stabil, sehingga menjadi opsi yang bagus untuk dataset yang berukuran kecil atau terurut. Sedangkan CS sangat baik dalam mengoptimalkan waktu eksekusi, sehingga masih menjadi pilihan utama untuk rangkaian data yang tak terurut dan berskala besar. Karakteristik spesifik dari dataset dan batasan memori dalam situasi tertentu menjadi faktor penentu utama dalam performa terbaik kedua algoritma tersebut.

---

1. Bidirectional-Conditional-Insertion-Sort (BCIS)

Algoritma BCIS merupakan sebuah algoritma perbaikan dari algoritma *insertion sort* yang mengoptimalkan proses *sorting* dengan membagi dua bagian pada *array*. Bagian tersebut terdapat pada bagian kiri *array* yang digunakan untuk menyimpan elemen terurut yang lebih kecil dan bagian kanan *array* untuk menyimpan elemen terurut yang lebih besar. Cara kerjanya, bagian kiri mengurutkan secara *ascending* masing-masing elemen dengan penunjuk yang *incremental*, sedangkan bagian kanan mengurutkan secara *descending* masing-masing elemen dengan petunjuk yang *decremental*, sehingga menyisakan elemen-elemen di bagian tengah *array* yang berisikan deret angka yang belum terurut. Selain itu, terdapat optimisasi pula bagi BCIS dengan tetap menjaga panjang *sort trip* (*k*) selalu mendekati akar dari panjang *array* agar meminimalkan jumlah perbandingan yang perlu dilakukan dalam setiap iterasi. Hal ini menghemat waktu dan tidak menghabiskan terlalu banyak memori.

2. Counting Sort (CS)

Algoritma Counting Sort (CS) adalah algoritma pengurutan yang berfokus pada penghitungan frekuensi kemunculan setiap elemen dalam dataset yang akan diurutkan. Algoritma ini efektif dalam mengurutkan data yang memiliki *range* nilai yang terbatas. Cara kerjanya adalah dengan membuat *array* tambahan yang disebut *array* hitung (*counting array*) yang digunakan untuk menghitung berapa kali setiap nilai muncul dalam dataset. Setelah itu, algoritma akan membuat array hasil yang sudah terurut berdasarkan informasi dari *array* hitung. CS bekerja dalam waktu linier, dan

kinerjanya sangat baik ketika *range* nilai dalam dataset relatif kecil. Algoritma ini tidak melakukan perbandingan elemen, sehingga ia sangat efisien dalam hal waktu eksekusi. Namun, algoritma ini membutuhkan lebih banyak memori, terutama saat nilai dalam dataset sangat besar.

3. Perbandingan waktu eksekusi dari kedua algoritma

Untuk membandingkan waktu eksekusi keduanya, kami mengimplementasikan kedua algoritma dengan menggunakan bahasa pemrograman python. Setelah kami buat, kami menggunakan modul time yang digunakan untuk mencari *interval* dari sebelum dan sesudah dilakukannya suatu algoritma shingga didapatkan waktu eksekusi dari algoritma tersebut.

	Dataset Type	Algorithm	Dataset Size	Execution Time (ms)
0	Unsorted	BCIS	500	0.740767
1	Sorted	BCIS	500	0.387192
2	Reversed	BCIS	500	5.259037
3	Unsorted	Counting Sort	500	0.091076
4	Sorted	Counting Sort	500	0.192881
5	Reversed	Counting Sort	500	0.193834
6	Unsorted	BCIS	5000	61.010122
7	Sorted	BCIS	5000	3.199816
8	Reversed	BCIS	5000	546.966076
9	Unsorted	Counting Sort	5000	0.916958
10	Sorted	Counting Sort	5000	1.983881
11	Reversed	Counting Sort	5000	1.989126
12	Unsorted	BCIS	50000	9730.520010
13	Sorted	BCIS	50000	33.200979
14	Reversed	BCIS	50000	55412.424088
15	Unsorted	Counting Sort	50000	9.804964
16	Sorted	Counting Sort	50000	19.845009
17	Reversed	Counting Sort	50000	19.727230

Figure 1: Tabel waktu eksekusi

Perbandingan waktu eksekusi dari kedua algoritma mengungkapkan perbedaan yang sangat signifikan dalam kinerja keduanya, yang erat terkait dengan sifat dasar masing-masing algoritma. Counting sort terbukti jauh lebih cepat dan efisien dalam mengurutkan dataset yang tidak terurut, terurut, maupun yang sudah terbalik. Ini terjadi karena Counting sort memanfaatkan penghitungan frekuensi nilai-nilai dalam dataset, yang memungkinkan pengurutan elemen-elemen tersebut dengan cara yang lebih efisien daripada algoritma perbandingan tradisional.

Sementara itu, BCIS mungkin menunjukkan performa yang relatif baik pada dataset yang sudah diurutkan, namun mengalami kinerja yang sangat buruk pada dataset yang sudah terbalik, bahkan pada dataset yang lebih besar. Ini sesuai dengan sifat BCIS yang lebih efisien dalam mengelola dataset yang hampir terurut, karena menghindari perbandingan yang tidak perlu. Namun, saat datanya sudah terbalik, BCIS tidak mampu memanfaatkan keunggulan ini, sehingga waktu eksekusinya menjadi sangat lambat.

Dalam kesimpulannya, hasil eksperimen ini memberikan gambaran jelas tentang keunggulan Counting sort pada segi waktu eksekusi dalam mengatasi berbagai jenis dataset. BCIS mungkin memiliki keunggulan pada dataset yang sudah diurutkan, tetapi kinerjanya yang lambat pada dataset lainnya membuatnya kurang efisien secara keseluruhan. Counting sort, dengan pendekatan berbasis penghitungan, tetap stabil dalam kinerjanya dan efisien pada dataset dengan kisaran nilai terbatas.

4. Perbandingan penggunaan memori dari kedua algoritma

Kami menggunakan modul tracemalloc pada python yang sangat berguna untuk memantau dan menganalisis penggunaan memori dalam program. Ini memungkinkan kami untuk mengidentifikasi seberapa banyak memori yang dialokasikan selama eksekusi program kami.

	Dataset Type	Algorithm	Dataset Size	Memory Consumption (bytes)
0	Unsorted	BCIS	500	224
1	Sorted	BCIS	500	224
2	Reversed	BCIS	500	328
3	Unsorted	Counting Sort	500	16704
4	Sorted	Counting Sort	500	28576
5	Reversed	Counting Sort	500	28576
6	Unsorted	BCIS	5000	224
7	Sorted	BCIS	5000	256
8	Reversed	BCIS	5000	328
9	Unsorted	Counting Sort	5000	235392
10	Sorted	Counting Sort	5000	355264
11	Reversed	Counting Sort	5000	355264
12	Unsorted	BCIS	50000	256
13	Sorted	BCIS	50000	256
14	Reversed	BCIS	50000	296
15	Unsorted	Counting Sort	50000	2480224
16	Sorted	Counting Sort	50000	3680096
17	Reversed	Counting Sort	50000	3680096

Figure 2: Tabel penggunaan memori

Dalam analisis perbandingan penggunaan memori antara dua algoritma *sorting*, terlihat perbedaan yang signifikan. BCIS menunjukkan konsumsi memori yang sangat rendah dan stabil pada berbagai jenis dataset dan ukuran dataset, berkisar antara 224 hingga 328 byte dalam semua kasus yang diuji. Hal ini menggambarkan sifat algoritma BCIS yang sangat efisien dalam penggunaan memori, terlepas dari ukuran dataset atau jenis data yang diurutkan. Sebaliknya, CS menunjukkan konsumsi memori yang lebih besar, terutama terkait dengan rentang nilai data yang diurutkan. Semakin besar nilai data atau semakin tidak teratur urutan data, semakin besar alokasi memori yang dibutuhkan oleh CS. Ini dapat dijelaskan oleh kebutuhan CS untuk *array* tambahan yang ukurannya sebanding dengan nilai maksimum dalam dataset. Oleh karena itu, jika masalah penggunaan memori adalah prioritas, BCIS menjadi pilihan yang lebih efisien.

5. Analisis Kompleksitas

Pada pencarian kompleksitas, kita dapat mencarinya melalui perbandingan antara besaran dataset dengan waktu eksekusinya yang terdapat pada Figure 1. Untuk BCIS dengan *average case*, kita membandingkan untuk ukuran dataset 500:5000:50000 dengan waktu eksekusi 0.74:61:9730. Dapat dilihat penambahan 10 kali lipat pada ukuran dataset akan mengakibatkan waktu eksekusi berlipat 100 kali, artinya kompleksitasnya melebihi O(n^2). Selanjutnya untuk BCIS dengan *best case,* membandingkan ukuran dataset 500:5000:50000 dengan waktu eksekusi 0.38:3.19:33.2 menghasilkan kelipatan yang sama sehingga kompleksitasnya adalah O(n). Dengan menggunakan metode yang sama, kita mendapatkan kompleksitas BCIS *worst case* sebesar O(n^2) dan kompleksitas CS untuk *average case, best case,* dan *worst case* secara berturut-turut adalah O(n), O(n), dan O(n).

6. Kesimpulan

Dalam perbandingan antara Bidirectional-Conditional-Insertion-Sort (BCIS) dan Counting Sort (CS), dapat ditarik kesimpulan berikut. BCIS memiliki kinerja baik pada dataset hampir terurut, dengan kompleksitas waktu berkisar antara O(n) hingga O(n^2), namun kinerjanya buruk pada dataset yang sudah terbalik atau besar. Di sisi lain, CS sangat efisien dalam mengurutkan dataset dengan rentang nilai terbatas, dengan kompleksitas waktu O(n), tetapi membutuhkan lebih banyak memori.

Dalam hal penggunaan memori, BCIS lebih efisien dan stabil, sementara CS membutuhkan lebih banyak memori, terutama pada dataset dengan nilai yang besar. Pilihan antara kedua algoritma tergantung pada jenis dataset dan prioritas penggunaan memori. BCIS cocok untuk dataset hampir terurut dengan penggunaan memori yang efisien, sementara CS lebih baik digunakan pada dataset dengan rentang nilai terbatas, meskipun membutuhkan lebih banyak memori.

Lampiran

Algorithm BCIS Part 1 (Main Body)

```
1:  SL ← left
2:  SR ← right
3:  while SL < SR do
4:      SWAP(array, SR, SL + (SR−SL)/2)
5:      if array[SL] = array[SR] then
6:          if ISEQUAL(array, SL, SR)=-1 then
7:              return
8:          end if
9:      end if
10:     if array[SL] > array[SR] then
11:         SWAP (array, SL , SR)
12:     end if
13:     if (SR − SL) ≥ 100 then
14:         for i ← SL + 1 to (SR − SL)^0.5 do
15:             if array[SR] < array[i] then
16:                 SWAP (array, SR, i)
17:             else if array[SL] > array[i] then
18:                 SWAP (array, SL, i)
19:             end if
20:         end for
21:     else
22:         i ← SL + 1
23:     end if
24:     LC ← array[SL]
25:     RC ← array[SR]
26:     while i < SR do
27:         CurrItem ← array[i]
28:         if CurrItem ≥ RC then
29:             array[i] ← array[SR − 1]
30:             INSRIGHT(array, CurrItem, SR, right)
31:             SR ← SR − 1
32:         else if CurrItem ≤ LC then
33:             array[i] ← array[SL + 1]
34:             INSLEFT(array, CurrItem, SL, left)
35:             SL ← SL + 1
36:             i ← i + 1
37:         else
38:             i ← i + 1
39:         end if
40:     end while
41:     SL ← SL + 1
42:     SR ← SR − 1
43: end while
```

Pseudocode untuk BCIS part 1

Algorithm BCIS Part 2 (Functions)

```
44: function ISEQUAL(array, SL, SR)
45:     for k ← SL + 1 to SR − 1 do
46:         if array[k]! = array[SL] then
47:             SWAP(array, k, SL)
48:             return k
49:         end if
50:     end for
51:     return − 1
52:   ▷ End the algorithm because all scanned items are equa
53: end function
54: function INSRIGHT(array, CurrItem, SR, right)
55:     j ← SR
56:     while j ≤ right and CurrItem > array[j] do
57:         array[j − 1] ← array[j]
58:         j ← j + 1
59:     end while
60:     Array[j − 1] ← CurrItem
61: end function
62: function INSLEFT(array, CurrItem, SL, left)
63:     j ← SL
64:     while j ≥ left and CurrItem < array[j] do
65:         array[j + 1] ← array[j]
66:         j ← j − 1
67:     end while
68:     Array[j + 1] ← CurrItem
69: end function
70: function SWAP(array,i,j)
71:     Temp ← array[i]
72:     array[i] ← array[j]
73:     array[j] ← Temp
74: end function
```

Pseudocode untuk BCIS part 2

```
COUNTING-SORT( A, B, k )          // n = length[A]
1.  for i ← 0 to k
2.      do C[i] ← 0
3.  for j ← 1 to n
4.      do C[ A[j] ] ← C[ A[j] ] + 1
5.  //C[i] now contains the number of elements equal to i.
6.  for i ← 1 to k
7.      do C[i] ← C[i] + C[i - 1]
8.  //C[i] now contains the number of elements less than or equal to i.
9.  for j ← n downto 1     //Place A[j] in its correct sorted position.
10.     do B[ C[ A[j] ] ] ← A[j]
11.        C[ A[j] ] ← C[ A[j] ] - 1     //Taking care of equal elements.
```

Pseudocode untuk CS

Tautan Github pengerjaan:

https://github.com/fauzanfir/BCIS-CS-Experiment/

References

[1] Adnan Saher Mohammed, S¸ahin Emrah Amrahov, and Fatih V C¸ elebi. Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort. Future Generation Computer Systems, 71:102–112, 2017.

[2] https://github.com/fauzanfir/BCIS-CS-Experiment/

[3] https://scele.cs.ui.ac.id/pluginfile.php/195808/mod_resource/content/1/daa7.pdf