Yitzhak Edmund Tio Manalu (22/499769/TK/54763) (S-1 Teknologi Informasi)

**ASD : Enam Algoritma Sorting**

**Objektif Laporan**

Pada penjelasan materi sebelumnya, diberikan enam macam algoritma metode sorting, yaitu Shell, Merge, Binary Tree, Radix, Quick, dan Heap. Masing-masing algoritma memiliki perbedaan dari segi kompleksitas dan efektivitas. Oleh karena itu, pada laporan ini, akan dilakukan analisis terhadap masing-masing algoritma tersebut sekaligus menjabarkannya dalam bentuk implementasi. *Source code* dari masing-masing algoritma diadaptasi dari berbagai *website*, tetapi masing-masing analisis diberikan tambahan kode berupa *counter* waktu dan data set array berukuran 500.000 *integers* untuk melakukan perbandingan efisiensi antara masing-masing algoritma.

1. **Shell Sort Method**
2. **Gambaran Umum**

Metode Shell Sort merupakan algoritma yang melakukan *sorting* dengan cara melakukan perbandingan antara dua data yang berjarak terlebih dahulu. Umumnya, metode ini dilakukan dengan dimulai dari jarak sebesar n/2 sehingga data dari pertama hingga data ke-n/2 akan di-*sorting* dengan data lawannya yang berjarak n/2 dengan data tersebut. Setelah itu, jarak dibagi lagi menjadi setengahnya (jika awalnya n/2, maka jarak berikutnya adalah n/4) hingga perbandingan jarak tersisa menjadi 1 saja. Ketika nilai jarak sudah menjadi 1, maka *sorting* sudah hampir selesai di mana masing-masing data hanya perlu digeser sekali ke posisi yang tepat dengan metode *Insertion Sort*. Ilustrasi dari algoritma tersebut dapat dilihat pada GIF di bawah ini.

|  |
| --- |
| Sumber: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Shell_Sort.gif> |

1. **Kompleksitas**

Diketahui bahwa algoritma ini memiliki kompleksitas *best case* dari algoritma ini adalah O(n log n) dengan *worst case* O(n­­2) dan rata-rata O(n log n). Kompleksitas ini jika berjalan secara rata-rata maka dapat dikatakan sebagai algoritma yang cukup baik, tetapi tampaknya apabila kasus yang terjadi adalah *worst case*, maka algoritma ini cukup memakan biaya waktu.

1. **Source Code dan Implementasi**

#include <iostream>

#include <string>

#include <chrono>

using namespace std;

using namespace std::chrono;

milliseconds deltaInit;

milliseconds deltaVal;

// Shell Sort algorithm from https://www.geeksforgeeks.org/shellsort/

/\* function to sort arr using shellSort \*/

int shellSort(int arr[], int n)

{

    // Start with a big gap, then reduce the gap

    for (int gap = n/2; gap > 0; gap /= 2)

    {

        // Do a gapped insertion sort for this gap size.

        // The first gap elements a[0..gap-1] are already in gapped order

        // keep adding one more element until the entire array is

        // gap sorted

        for (int i = gap; i < n; i += 1)

        {

            // add a[i] to the elements that have been gap sorted

            // save a[i] in temp and make a hole at position i

            int temp = arr[i];

            // shift earlier gap-sorted elements up until the correct

            // location for a[i] is found

            int j;

            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

                arr[j] = arr[j - gap];

            //  put temp (the original a[i]) in its correct location

            arr[j] = temp;

        }

    }

    return 0;

}

void printArray(int arr[], int n)

{

    for (int i=0; i<n; i++)

        cout << arr[i] << " ";

}

void countTime(int cond = 0 /\*0 = set start, 1 = stop count, 2 = print count\*/) {

    if (cond == 0)

        deltaInit = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch());

    else if (cond == 1)

        deltaVal = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch()) - deltaInit;

    else if (cond == 2)

        cout << "Interval : " << to\_string(deltaVal.count()) << "ms" << endl;

}

void shellSortRes(int\* arr={0}, unsigned int n=1, bool print=false) {

    if (print) {

        cout << "Array before sorting: \n";

        printArray(arr, n);

    }

    countTime(0);

    shellSort(arr, n);

    countTime(1);

    if (print) {

        cout << "\nArray after sorting: \n";

        printArray(arr, n);

        cout << "\n";

    }

}

int main() {

    int arr[] = {12, 34, 54, 2, 3};

    int n = sizeof(arr)/sizeof(arr[0]);

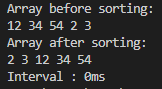
    shellSortRes(arr, n, true);

    countTime(2);

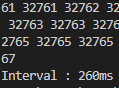
    return 0;

}

Dalam *source code* tersebut, sudah terdapat *unsorted array* berukuran 5 *integers* dengan isi 12, 34, 54, 2, dan 3. Hasil dari operasi algoritma tersebut sesuai pada *output* berikut ini.



Kode ini berjalan dengan masa waktu 0ms yang terhitung sangat cepat. Akan tetapi, untuk mengujinya lebih baik, maka perlu diberikan *data set* yang lebih besar lagi yang sudah di-*generate* menggunakan kode yang terdapat pada lampiran. Berikut adalah hasil uji menggunakan *data set* berukuran 500.000 *integers*.



Sesuai pada gambar di atas, algoritma Shell Sort membutuhkan waktu 260 ms saja untuk memproses *unsorted random array* yang terdiri dari 500.000 *integers*. Akan tetapi, untuk melihat apakah nilai ini tetap, dilakukan 10x operasi yang menghasilkan kecepatan seperti yang terdapat pada tabel di bawah ini.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 260ms | 192ms | 225ms | 191ms | 210ms | 321ms | 206ms | 221ms | 229ms | 198ms |

Maka, rata-rata durasi operasi yang dilakukan oleh algoritma Shell Sort membutuhkan waktu 225,3ms saja untuk menyelesaikan *sorting* data sebanyak 500.000 *integers*.

1. **Merge Sort Method**
2. **Gambaran Umum**

Metode Merge Sort merupakan algoritma yang melakukan sorting dengan cara melakukan pembagian bagian-bagian *array* hingga sekecil mungkin baru kemudian di-*sorting­* sesuai dengan masing-masing bagiannya. Kemudian, bagian-bagian kecil yang sudah ter-*sort* mulai digabungkan kembali sekaligus di-*sorting* hingga membentuk kesatuan utuh yang sudah ter-*sort* secara utuh. Algoritma ini sebenarnya membuat kegiatan *sorting* menjadi lebih terstruktur karena pengurutan menjadi lebih mudah meskipun membutuhkan beberapa iterasi. Berikut ini merupakan gambar dan GIF yang menjelaskan bagaimana algoritma Merge Sort bekerja.

|  |
| --- |
| Sumber: <https://media.geeksforgeeks.org/wp-content/uploads/20220722205737/MergeSortTutorial.png> |

|  |
| --- |
| Sumber: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Merge_Sort.gif> |

1. **Kompleksitas**

Merge Sort memiliki kompleksitas yang sama untuk ketiga kondisinya, yaitu *best case*, *average case*, dan *worst case*. Ketiganya sama-sama memiliki kompleksitas O(n log n) sehingga dapat dikatakan bahwa algoritma ini lebih baik daripada algoritma Shell Sort.

1. **Source Code dan Implementasi**

// C++ program for Merge Sort

#include <iostream>

#include <string>

#include <chrono>

using namespace std;

using namespace std::chrono;

milliseconds deltaInit;

milliseconds deltaVal;

void countTime(int cond = 0 /\*0 = set start, 1 = stop count, 2 = print count\*/) {

    if (cond == 0)

        deltaInit = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch());

    else if (cond == 1)

        deltaVal = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch()) - deltaInit;

    else if (cond == 2)

        cout << "Interval : " << to\_string(deltaVal.count()) << "ms" << endl;

}

// Merges two subarrays of array[].

// First subarray is arr[begin..mid]

// Second subarray is arr[mid+1..end]

void merge(int array[], int const left, int const mid, int const right)

{

    auto const subArrayOne = mid - left + 1;

    auto const subArrayTwo = right - mid;

    // Create temp arrays

    auto \*leftArray = new int[subArrayOne],

        \*rightArray = new int[subArrayTwo];

    // Copy data to temp arrays leftArray[] and rightArray[]

    for (auto i = 0; i < subArrayOne; i++)

        leftArray[i] = array[left + i];

    for (auto j = 0; j < subArrayTwo; j++)

        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne

        = 0, // Initial index of first sub-array

        indexOfSubArrayTwo

        = 0; // Initial index of second sub-array

    int indexOfMergedArray

        = left; // Initial index of merged array

    // Merge the temp arrays back into array[left..right]

    while (indexOfSubArrayOne < subArrayOne

        && indexOfSubArrayTwo < subArrayTwo) {

        if (leftArray[indexOfSubArrayOne]

            <= rightArray[indexOfSubArrayTwo]) {

            array[indexOfMergedArray]

                = leftArray[indexOfSubArrayOne];

            indexOfSubArrayOne++;

        }

        else {

            array[indexOfMergedArray]

                = rightArray[indexOfSubArrayTwo];

            indexOfSubArrayTwo++;

        }

        indexOfMergedArray++;

    }

    // Copy the remaining elements of

    // left[], if there are any

    while (indexOfSubArrayOne < subArrayOne) {

        array[indexOfMergedArray]

            = leftArray[indexOfSubArrayOne];

        indexOfSubArrayOne++;

        indexOfMergedArray++;

    }

    // Copy the remaining elements of

    // right[], if there are any

    while (indexOfSubArrayTwo < subArrayTwo) {

        array[indexOfMergedArray]

            = rightArray[indexOfSubArrayTwo];

        indexOfSubArrayTwo++;

        indexOfMergedArray++;

    }

    delete[] leftArray;

    delete[] rightArray;

}

// begin is for left index and end is

// right index of the sub-array

// of arr to be sorted \*/

void mergeSort(int array[], int const begin, int const end)

{

    if (begin >= end)

        return; // Returns recursively

    auto mid = begin + (end - begin) / 2;

    mergeSort(array, begin, mid);

    mergeSort(array, mid + 1, end);

    merge(array, begin, mid, end);

}

// UTILITY FUNCTIONS

// Function to print an array

void printArray(int A[], int size)

{

    for (auto i = 0; i < size; i++)

        cout << A[i] << " ";

    cout << endl;

}

// Driver code

int main()

{

    int arr[] = {12, 34, 54, 2, 3};

    auto arr\_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";

    printArray(arr, arr\_size);

    countTime(0);

    mergeSort(arr, 0, arr\_size - 1);

    countTime(1);

    cout << "\nSorted array is \n";

    printArray(arr, arr\_size);

    countTime(2);

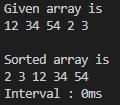
    return 0;

}

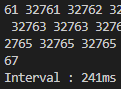
// This code is contributed by Mayank Tyagi

// This code was revised by Joshua Estes

Dalam *source code* tersebut, sudah terdapat *unsorted array* berukuran 5 *integers* dengan isi 12, 34, 54, 2, dan 3. Hasil dari operasi algoritma tersebut sesuai pada *output* berikut ini.



Kode ini berjalan dengan masa waktu 0ms yang terhitung sangat cepat. Akan tetapi, untuk mengujinya lebih baik, maka perlu diberikan *data set* yang lebih besar lagi yang sudah di-*generate* menggunakan kode yang terdapat pada lampiran. Berikut adalah hasil uji menggunakan *data set* berukuran 500.000 *integers*.



Sesuai pada gambar di atas, algoritma Merge Sort membutuhkan waktu 241 ms saja untuk memproses *unsorted random array* yang terdiri dari 500.000 *integers*. Akan tetapi, untuk melihat apakah nilai ini tetap, dilakukan 10x operasi yang menghasilkan kecepatan seperti yang terdapat pada tabel di bawah ini.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 241ms | 225ms | 233ms | 233ms | 260ms | 246ms | 248ms | 244ms | 267ms | 246ms |

Maka, rata-rata durasi operasi yang dilakukan oleh algoritma Merge Sort membutuhkan waktu 244.3ms saja untuk menyelesaikan *sorting* data sebanyak 500.000 *integers*. Hal ini menunjukkan bahwa meskipun secara teori, kompleksitas Merge Sort lebih efektif ketimbang Shell Sort, hasil pada percobaan laporan ini justru menunjukkan bahwa Merge Sort bekerja lebih lambat ketimbang Shell Sort.

1. **Binary Search Tree Method**
2. **Gambaran Umum**

Metode Sorting Binary Search Tree merupakan metode yang memanfaatkan karakteristik BST yang memang memiliki pengurutan dengan bentuk *left child* < *parent* < *right child*. Bentuk BST ini apabila dibaca dengan cara *in-order traversal* akan selalu membaca dari *left child* terlebih dahulu kemudian parent, dan akhirnya *right child*. Sebagai gambaran, ilustrasi dapat dilihat dari GIF berikut ini.

|  |
| --- |
| Sumber: <https://blog.penjee.com/5-gifs-to-understand-binary-search-tree/> |

1. **Kompleksitas**

Kompleksitas dari BST Sort sesungguhnya serupa dengan kompleksitas pembentukan BST itu sendiri karena *sorting* dilakukan secara tidak langsung dengan cara membentuk BST. BST sendiri memiliki kompleksitas sebesar rata-rata O(n log n) dalam pembentukannya dan O(n2) sebagai *worst case*.

1. **Source Code dan Implementasi**

// C++ program to implement Tree Sort

#include <iostream>

#include <string>

#include <chrono>

using namespace std;

using namespace std::chrono;

milliseconds deltaInit;

milliseconds deltaVal;

void countTime(int cond = 0 /\*0 = set start, 1 = stop count, 2 = print count\*/) {

    if (cond == 0)

        deltaInit = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch());

    else if (cond == 1)

        deltaVal = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch()) - deltaInit;

    else if (cond == 2)

        cout << "Interval : " << to\_string(deltaVal.count()) << "ms" << endl;

}

struct Node

{

    int key;

    struct Node \*left, \*right;

};

// A utility function to create a new BST Node

struct Node \*newNode(int item)

{

    struct Node \*temp = new Node;

    temp->key = item;

    temp->left = temp->right = NULL;

    return temp;

}

// Stores inorder traversal of the BST

// in arr[]

void storeSorted(Node \*root, int arr[], int &i)

{

    if (root != NULL)

    {

        storeSorted(root->left, arr, i);

        arr[i++] = root->key;

        storeSorted(root->right, arr, i);

    }

}

/\* A utility function to insert a new

Node with given key in BST \*/

Node\* insert(Node\* node, int key)

{

    /\* If the tree is empty, return a new Node \*/

    if (node == NULL) return newNode(key);

    /\* Otherwise, recur down the tree \*/

    if (key <= node->key)

        node->left = insert(node->left, key);

    else if (key > node->key)

        node->right = insert(node->right, key);

    /\* return the (unchanged) Node pointer \*/

    return node;

}

// This function sorts arr[0..n-1] using Tree Sort

void treeSort(int arr[], int n)

{

    struct Node \*root = NULL;

    // Construct the BST

    root = insert(root, arr[0]);

    for (int i=1; i<n; i++)

        root = insert(root, arr[i]);

    // Store inorder traversal of the BST

    // in arr[]

    int i = 0;

    storeSorted(root, arr, i);

}

// Function to print an array

void printArray(int A[], int size)

{

    for (auto i = 0; i < size; i++)

        cout << A[i] << " ";

    cout << endl;

}

// Driver Program to test above functions

int main()

{

    //create input array

    int arr[] = {12, 34, 54, 2, 3};

    int n = sizeof(arr)/sizeof(arr[0]);

    cout << "Given array is \n";

    printArray(arr, n);

    countTime(0);

    treeSort(arr, n);

    countTime(1);

    cout << "\nSorted array is \n";

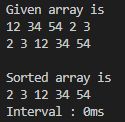
    printArray(arr, n);

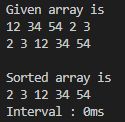
    countTime(2);

    return 0;

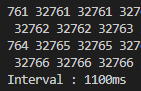
}

Dalam *source code* tersebut, sudah terdapat *unsorted array* berukuran 5 *integers* dengan isi 12, 34, 54, 2, dan 3. Hasil dari operasi algoritma tersebut sesuai pada *output* berikut ini.

****

****

Kode ini berjalan dengan masa waktu 0ms yang terhitung sangat cepat. Akan tetapi, untuk mengujinya lebih baik, maka perlu diberikan *data set* yang lebih besar lagi yang sudah di-*generate* menggunakan kode yang terdapat pada lampiran. Berikut adalah hasil uji menggunakan *data set* berukuran 500.000 *integers*.



Sesuai pada gambar di atas, algoritma BST Sort membutuhkan waktu 1.1s untuk memproses *unsorted random array* yang terdiri dari 500.000 *integers*. Waktu ini terbilang jauh lebih lama ketimbang waktu yang dibutuhkan algoritma-algoritma lain sebelumnya. Seperti yang telah dilakukan pada analisis algoritma sebelumnya, berikut merupakan data hasil 10x operasi yang sama yang dilakukan oleh algoritma ini yang ditunjukkan pada tabel di bawah.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 1100ms | 1240ms | 1296ms | 1278ms | 1203ms | 1518ms | 1548ms | 1301ms | 1540ms | 1271ms |

Maka, rata-rata durasi operasi yang dilakukan oleh algoritma BST Sort membutuhkan waktu 1329.5ms untuk menyelesaikan *sorting* data sebanyak 500.000 *integers*. Durasi ini dapat dikatakan jauh lebih lama ketimbang rata-rata durasi operasi algoritma sebelumnya. Hal ini dapat terjadi karena BST melakukan dua kali operasi terhadap data raksasa yang di-*sorting*, yaitu proses *insertion* ke dalam BST sekaligus pembacaan *in-order traversal* dari BST yang dibentuk. Hal ini menunjukkan bahwa algoritma ini memang nampaknya mudah karena hanya memerlukan pembacaan sederhana dari BST yang dibentuk. Namun, waktu operasi justru dihabiskan pada pembentukan BST.

1. **Radix Sort Method**
2. **Gambaran Umum**

Metode Radix Sort merupakan metode yang mengacu kepada radix atau basis dari suatu data berupa bilangan. Pada bilangan desimal, radix dapat didefinisikan sebagai tingkatan digit sesuai dengan basis bilangan desimal, yaitu 10. Maka, jenis-jenis radix yang ada dalam bilangan desimal adalah satuan, puluhan, ratusan, ribuan, puluh ribuan, dan seterusnya. Metode Radix Sort melakukan *sorting* dengan mengacu pada masing-masing radix dari radix terendah terlebih dahulu. Konsepnya adalah, setiap data akan di-*sorting* menurut ukuran satuannya terlebih dahulu dengan satuan yang terkecil berada di paling kiri dan satuan terbesar di paling kanan untuk *ascending sorting*. Setelah itu, tingkatan ditambah ke puluhan dan proses yang sama dilakukan hingga tidak ada lagi bilangan yang memiliki radix berlebih. Ilustrasi dari algoritma ini dapat dilihat dari GIF di bawah ini.

|  |
| --- |
| Sumber: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Radix_Sort.gif> |

1. **Kompleksitas**

Karena berupa algoritma yang bergantung pada radix, maka kompleksitas dari algoritma ini juga bergantung pada sifat radix dari data set yang di-*sorting* menggunakan algoritma ini. Khusus algoritma ini, kompleksitasnya adalah berupa O(d \* (n + b)) dengan d adalah jumlah digit, n adalah jumlah elemen, dan b adalah basis bilangan yang digunakan.

1. **Source Code dan Implementasi**

// C++ implementation of Radix Sort

#include <iostream>

#include <string>

#include <chrono>

using namespace std;

using namespace std::chrono;

milliseconds deltaInit;

milliseconds deltaVal;

void countTime(int cond = 0 /\*0 = set start, 1 = stop count, 2 = print count\*/) {

    if (cond == 0)

        deltaInit = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch());

    else if (cond == 1)

        deltaVal = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch()) - deltaInit;

    else if (cond == 2)

        cout << "Interval : " << to\_string(deltaVal.count()) << "ms" << endl;

}

// A utility function to get maximum value in arr[]

int getMax(int arr[], int n)

{

    int mx = arr[0];

    for (int i = 1; i < n; i++)

        if (arr[i] > mx)

            mx = arr[i];

    return mx;

}

// A function to do counting sort of arr[] according to

// the digit represented by exp.

void countSort(int arr[], const int n, int exp)

{

    int output[n]; // output array

    int i, count[10] = { 0 };

    // Store count of occurrences in count[]

    for (i = 0; i < n; i++)

        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual

    // position of this digit in output[]

    for (i = 1; i < 10; i++)

        count[i] += count[i - 1];

    // Build the output array

    for (i = n - 1; i >= 0; i--) {

        output[count[(arr[i] / exp) % 10] - 1] = arr[i];

        count[(arr[i] / exp) % 10]--;

    }

    // Copy the output array to arr[], so that arr[] now

    // contains sorted numbers according to current digit

    for (i = 0; i < n; i++)

        arr[i] = output[i];

}

// The main function to that sorts arr[] of size n using

// Radix Sort

void radixsort(int arr[], const int n)

{

    // Find the maximum number to know number of digits

    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead

    // of passing digit number, exp is passed. exp is 10^i

    // where i is current digit number

    for (int exp = 1; m / exp > 0; exp \*= 10)

        countSort(arr, n, exp);

}

// A utility function to print an array

void print(int arr[], int n)

{

    for (int i = 0; i < n; i++)

        cout << arr[i] << " ";

    cout << endl;

}

// Driver Code

int main()

{

    int arr[] = {12, 34, 54, 2, 3};

    const int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";

    print(arr, n);

    // Function Call

    countTime(0);

    radixsort(arr, n);

    countTime(1);

    cout << "\nSorted array is \n";

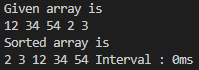
    print(arr, n);

    countTime(2);

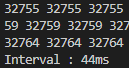
    return 0;

}

Dalam *source code* tersebut, sudah terdapat *unsorted array* berukuran 5 *integers* dengan isi 12, 34, 54, 2, dan 3. Hasil dari operasi algoritma tersebut sesuai pada *output* berikut ini.



Kode ini berjalan dengan masa waktu 0ms yang terhitung sangat cepat. Akan tetapi, untuk mengujinya lebih baik, maka perlu diberikan *data set* yang lebih besar lagi yang sudah di-*generate* menggunakan kode yang terdapat pada lampiran. Namun, setelah diuji, algoritma ini tidak mampu menyelesaikan permasalahan *data set* yang berukuran 500.000, 400.000, dan 300.000 *integers*. Oleh karena itu, data set yang digunakan khusus untuk algoritma ini diganti dengan data set yang berukuran 200.000 *integers*.



Sesuai pada gambar di atas, algoritma Radix Sort hanya membutuhkan waktu 44ms untuk memproses *unsorted random array* yang terdiri dari 200.000 *integers*. Waktu ini memang sangat singkat meskipun algoritma ini tetap terlalu berat untuk data yang terlalu besar. Berikut adalah perhitungan rata-rata kecepatan proses Radix Sort.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 44ms | 29ms | 29ms | 28ms | 28ms | 28ms | 29ms | 30ms | 31ms | 28ms |

Maka, rata-rata durasi operasi yang dilakukan oleh algoritma Radix Sort membutuhkan waktu 30.4ms untuk menyelesaikan *sorting* data sebanyak 200.000 *integers*. Meskipun memiliki kecepatan yang tinggi, algoritma ini terlalu berat untuk diterapkan pada data yang lebih besar. Hal ini kemungkinan terjadi karena Radix Sort membutuhkan ruang yang besar dalam penerapannya karena prosesnya yang banyak menggunakan *array*.

1. **Quick Sort Method**
2. **Gambaran Umum**

Metode Quick Sort merupakan metode *sorting* yang pada dasarnya memiliki kemiripan dengan Merge Sort. Hanya saja, metode ini tidak melakukan partisi tepat di tengah *array*, melainkan dengan mengambil suatu nilai acak yang terdapat dalam *array* tersebut dan kemudian membagi *array* tersebut menjadi dua dengan partisi masing-masing berupa satu bagian yang berisi data dengan nilai yang lebih kecil dari nilai yang dipilih dan bagian lainnya berisi data dengan nilai yang lebih besar atau sama dengan nilai yang dipilih. Hal ini terus menerus dilakukan hingga data tidak bisa dibagi lagi dan kemudian barulah digabungkan. Hasil penggabungannya sudah pasti ter­-*sorting* karena setiap partisi benar-benar memiliki hubungan lebih besar/lebih kecil secara langsung tanpa ada data lain di tengah-tengahnya yang memiliki kemungkinan lebih besar/lebih kecil pula terhadap nilai yang ada dari masing-masing partisi tersebut. Ilustrasi dari Quick Sort dapat dilihat pada gambar dan GIF di bawah ini.

|  |
| --- |
| Sumber: <https://www.geeksforgeeks.org/quick-sort/> |

|  |
| --- |
| Sumber: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Quick_Sort.gif> |

1. **Kompleksitas**

Kompleksitas dari algoritma ini cenderung standar, yaitu dengan *worst case* O(n2), *average case* O(n log n), dan *best case* O(n log n). Adapun kompleksitas dari algoritma ini sangat bergantung pada pemilihan pivot pada awal *sorting*. Pivot dapat diambil dari yang paling menguntungkan untuk algoritma ini agar tidak bekerja terlalu lama. Oleh karena itu, algoritma ini masuk ke dalam kategori algoritma yang kurang stabil karena memang tidak dapat selalu ditentukan kecepatan kerjanya meskipun memang pada umumnya, algoritma ini dapat bekerja dengan cepat sesuai pada namanya.

1. **Source Code dan Implementasi**

// C code to implement quicksort

#include <iostream>

#include <string>

#include <chrono>

using namespace std;

using namespace std::chrono;

milliseconds deltaInit;

milliseconds deltaVal;

void countTime(int cond = 0 /\*0 = set start, 1 = stop count, 2 = print count\*/) {

    if (cond == 0)

        deltaInit = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch());

    else if (cond == 1)

        deltaVal = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch()) - deltaInit;

    else if (cond == 2)

        cout << "Interval : " << to\_string(deltaVal.count()) << "ms" << endl;

}

// Function to swap two elements

void swap(int\* a, int\* b)

{

    int t = \*a;

    \*a = \*b;

    \*b = t;

}

// Partition the array using the last element as the pivot

int partition(int arr[], int low, int high)

{

    // Choosing the pivot

    int pivot = arr[high];

    // Index of smaller element and indicates

    // the right position of pivot found so far

    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {

        // If current element is smaller than the pivot

        if (arr[j] < pivot) {

            // Increment index of smaller element

            i++;

            swap(&arr[i], &arr[j]);

        }

    }

    swap(&arr[i + 1], &arr[high]);

    return (i + 1);

}

// The main function that implements QuickSort

// arr[] --> Array to be sorted,

// low --> Starting index,

// high --> Ending index

void quickSort(int arr[], int low, int high)

{

    if (low < high) {

        // pi is partitioning index, arr[p]

        // is now at right place

        int pi = partition(arr, low, high);

        // Separately sort elements before

        // partition and after partition

        quickSort(arr, low, pi - 1);

        quickSort(arr, pi + 1, high);

    }

}

void printArray(int arr[], const long long int n)

{

    for (int i = 0; i < n; i++)

        cout << arr[i] << " ";

    cout << endl;

}

// Driver code

int main()

{

    int arr[] = {12, 34, 54, 2, 3};

    int N = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";

    printArray(arr, N);

    // Function call

    countTime(0);

    quickSort(arr, 0, N - 1);

    countTime(1);

    cout << "\nSorted array is \n";

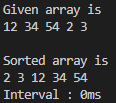
    printArray(arr, N);

    countTime(2);

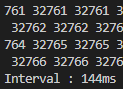
    return 0;

}

Dalam *source code* tersebut, sudah terdapat *unsorted array* berukuran 5 *integers* dengan isi 12, 34, 54, 2, dan 3. Hasil dari operasi algoritma tersebut sesuai pada *output* berikut ini.



Kode ini berjalan dengan masa waktu 0ms yang terhitung sangat cepat. Akan tetapi, untuk mengujinya lebih baik, maka perlu diberikan *data set* yang lebih besar lagi yang sudah di-*generate* menggunakan kode yang terdapat pada lampiran. Berikut adalah hasil uji menggunakan *data set* berukuran 500.000 *integers*.



Sesuai pada gambar di atas, algoritma Quick Sort membutuhkan waktu 144ms untuk memproses *unsorted random array* yang terdiri dari 500.000 *integers*. Selanjutnya, untuk mengonfirmasi kestabilan algoritma ini, dilakukan perhitungan 10x uji coba seperti yang terlihat pada tabel di bawah ini.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 144ms | 125ms | 133ms | 128ms | 147ms | 160ms | 131ms | 156ms | 153ms | 162ms |

Maka, rata-rata durasi operasi yang dilakukan oleh algoritma Quick Sort membutuhkan waktu 143.9ms untuk menyelesaikan *sorting* data sebanyak 500.000 *integers*.

1. **Heap Sort Method**
2. **Gambaran Umum**

Metode Heap Sort merupakan metode *sorting* yang memanfaatkan struktur data *binary tree* sebagai media untuk melakukan *sorting*. *Binary tree* yang dibentuk tentunya bukanlah BST dengan pembentukan yang hanya mempertimbangkan hasil berupa *complete binary tree* saja. Sehingga, pembentukan *tree* menjadi jauh lebih cepat ketimbang BST dan dapat disesuaikan dengan cara kerja dari Heap Sort. Heap Sort pada dasarnya bekerja dengan memastikan bahwa *root* dari *tree* selalu diisi oleh data dengan nilai paling tinggi. Setelah itu, akan dilakukan proses *heapify*, yaitu proses di mana *root* diputar langsung dengan *leaf* paling ujung kanan dan kemudian nilai tersebut dipotong relasinya terhadap *tree* dengan kondisi nilai tersebut sudah berada pada indeks *sorting* yang tepat. Berikut adalah GIF yang dapat menampilkan ilustrasi dari proses tersebut.

|  |
| --- |
| Sumber: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Heap_Sort.gif> |

1. **Kompleksitas**

Kompleksitas yang dimiliki Heap Sort adalah sama untuk ketiga kasus, yaitu *worst case*, *average case*, dan *best case* dengan nilai O(n log n). Akan tetapi, algoritma ini termasuk algoritma yang tidak stabil karena bergantung pada urutan yang terbentuk pada *binary tree* yang juga bergantung kepada urutan awal dari data set yang di-*sorting*. Meskipun demikian, secara memori, algoritma ini termasuk algoritma yang sangat hemat memori karena penggunaan memorinya hanya berbasis pada tambahan *tree* saja tanpa perlu menambah *array-array* lainnya sebagai memori tambahan.

1. **Source Code dan Implementasi**

// C++ program for implementation of Heap Sort

#include <iostream>

#include <string>

#include <chrono>

using namespace std;

using namespace std::chrono;

milliseconds deltaInit;

milliseconds deltaVal;

void countTime(int cond = 0 /\*0 = set start, 1 = stop count, 2 = print count\*/) {

    if (cond == 0)

        deltaInit = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch());

    else if (cond == 1)

        deltaVal = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch()) - deltaInit;

    else if (cond == 2)

        cout << "Interval : " << to\_string(deltaVal.count()) << "ms" << endl;

}

// To heapify a subtree rooted with node i

// which is an index in arr[].

// n is size of heap

void heapify(int arr[], int N, int i)

{

    // Initialize largest as root

    int largest = i;

    // left = 2\*i + 1

    int l = 2 \* i + 1;

    // right = 2\*i + 2

    int r = 2 \* i + 2;

    // If left child is larger than root

    if (l < N && arr[l] > arr[largest])

        largest = l;

    // If right child is larger than largest

    // so far

    if (r < N && arr[r] > arr[largest])

        largest = r;

    // If largest is not root

    if (largest != i) {

        swap(arr[i], arr[largest]);

        // Recursively heapify the affected

        // sub-tree

        heapify(arr, N, largest);

    }

}

// Main function to do heap sort

void heapSort(int arr[], int N)

{

    // Build heap (rearrange array)

    for (int i = N / 2 - 1; i >= 0; i--)

        heapify(arr, N, i);

    // One by one extract an element

    // from heap

    for (int i = N - 1; i > 0; i--) {

        // Move current root to end

        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap

        heapify(arr, i, 0);

    }

}

// A utility function to print array of size n

void printArray(int arr[], int N)

{

    for (int i = 0; i < N; ++i)

        cout << arr[i] << " ";

    cout << "\n";

}

// Driver's code

int main()

{

    int arr[] = { 12, 11, 13, 5, 6, 7 };

    int N = sizeof(arr) / sizeof(arr[0]);

    cout << "Given array is \n";

    printArray(arr, N);

    // Function call

    countTime(0);

    heapSort(arr, N);

    countTime(1);

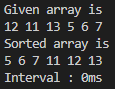
    cout << "Sorted array is \n";

    printArray(arr, N);

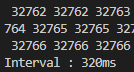
    countTime(2);

}

Dalam *source code* tersebut, sudah terdapat *unsorted array* berukuran 5 *integers* dengan isi 12, 34, 54, 2, dan 3. Hasil dari operasi algoritma tersebut sesuai pada *output* berikut ini.

****

Kode ini berjalan dengan masa waktu 0ms yang terhitung sangat cepat. Akan tetapi, untuk mengujinya lebih baik, maka perlu diberikan *data set* yang lebih besar lagi yang sudah di-*generate* menggunakan kode yang terdapat pada lampiran. Berikut adalah hasil uji menggunakan *data set* berukuran 500.000 *integers*.



Sesuai pada gambar di atas, algoritma Heap Sort membutuhkan waktu 320ms untuk memproses *unsorted random array* yang terdiri dari 500.000 *integers*. Selanjutnya, untuk mengonfirmasi kestabilan algoritma ini, dilakukan perhitungan 10x uji coba seperti yang terlihat pada tabel di bawah ini.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 320ms | 315ms | 395ms | 415ms | 333ms | 391ms | 425ms | 371ms | 356ms | 387ms |

Maka, rata-rata durasi operasi yang dilakukan oleh algoritma Heap Sort membutuhkan waktu 370.8ms untuk menyelesaikan *sorting* data sebanyak 500.000 *integers*.

**Kesimpulan**

Jika hasil pengujian sebelumnya didata dalam sebuah tabel, maka didapatkan akumulasi data seperti berikut.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Rank | Nama Algoritma | Rata-rata Durasi Operasi  (500k *integers*) | Average Time  Complexity | Keterangan |
| 1 | Quick Sort | 143.9ms | O(n log n) | Bekerja sangat cepat |
| 2 | Shell Sort | 225.3ms | O(n log n) | Bekerja dengan cepat |
| 3 | Merge Sort | 244.3ms | O(n log n) | Bekerja dengan cepat |
| 4 | Heap Sort | 370.8ms | O(n log n) | Bekerja relatif cepat |
| 5 | BST Sort | 1329.5ms | O(n log n) | Terhambat pembentukan BST |
| 6 | Radix Sort | 30.4ms (200k *integers*) | O(d\*(n+b)) | Memori terlalu banyak dipakai |

Meskipun demikian, karena pertimbangan yang digunakan dalam laporan ini adalah kompleksitas waktu, diperlukan adanya eksperimen dalam mempertimbangkan kompleksitas ruang juga.

Akan tetapi, apabila hanya mempertimbangkan *time complexity* saja, maka algoritma yang terbaik adalah Quick Sort dengan performa 36% lebih cepat daripada Shell Sort yang menempati urutan kedua. Selain itu, Shell Sort dan Merge Sort memiliki durasi yang kurang lebih sama meskipun Shell Sort memiliki kecepatan yang sedikit lebih tinggi jika dibandingkan dengan Merge Sort. Selanjutnya, Heap Sort menempati posisi keempat dengan kecepatan yang lebih lambat sebesar 51.8% dari Merge Sort. Hal ini menunjukkan bahwa Heap Sort kurang efektif secara kecepatan jika dibandingkan dengan ketiga algoritma *sorting* sebelumnya. Selanjutnya terdapat BST Sort yang kinerjanya terhambat oleh pembentukan BST. Padahal, pembacaan *sorting* melalui *in-order traversal* bisa saja berjalan dengan kecepatan O(n) karena tidak ada hambatan. Namun, proses pembentukan BST ini yang meningkatkan kompleksitasnya menjadi O(n log n) dan mengakibatkan durasi operasinya meningkat 258% dari durasi Heap Sort. Selanjutnya di posisi terakhir, terdapat Radix Sort yang menerapkan terlalu banyak *array* yang disesuaikan dengan ukuran basis data set. Hal ini mengakibatkan algoritma ini tidak cocok untuk data besar bahkan gagal mengoperasikan data berukuran 300k *integers* ke atas. Meskipun kecepatannya yang sangat tinggi dalam memproses data set berukuran 200k *integers*, tetap saja kualitasnya berkurang akibat ketidakmampuannya dalam memproses data yang sangat besar. Akan tetapi, algoritma ini dapat mengalahkan Quick Sort pada ukuran data yang kecil dengan data tambahan kecepatan Quick Sort dalam memproses data set yang sama seperti yang ditampilkan di bawah tabel ini.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 45ms | 40ms | 42ms | 47ms | 39ms | 55ms | 62ms | 53ms | 61ms | 53ms |

Dapat dilihat bahwa pada tabel di atas, Quick Sort hanya mampu melakukan performa rata-rata sebesar 49.7ms dan nilai ini bernilai 63% lebih lambat jika dibandingkan dengan Radix Sort. Hal ini menunjukkan bahwa salah satu pertimbangan dalam memilih algoritma yang tepat adalah ukuran dari data set yang akan di-*sorting* karena masing-masing algoritma memiliki spesialisasinya sendiri.

**Lampiran**

1. *Source Code Data Set Generator*

#include <iostream>

#include <fstream>

#include <string>

#include <time.h>

#define size 500000

using namespace std;

int main() {

    string fileName = "generated\_dataset.txt";

    ofstream outfile;

    cout << "Opening file..." << endl;

    outfile.open(fileName);

    cout << "File opened." << endl;

    srand((unsigned) time(0));

    cout << "Inserting to " << fileName << "..." << endl;

    for (long long int i = 0; i < size; i++)

        outfile << (unsigned int) rand() << ((i + 1 == size) ? "" : ", ");

    cout << "Insertion finished."  << endl;

    cout << "Closing file..." << endl;

    outfile.close();

    cout << "File closed." << endl;

    return 0;

}

1. *Source Code Timestamp Recorder*

#include <chrono>

using namespace std::chrono;

milliseconds deltaInit;

milliseconds deltaVal;

void countTime(int cond = 0 /\*0 = set start, 1 = stop count, 2 = print count\*/) {

    if (cond == 0)

        deltaInit = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch());

    else if (cond == 1)

        deltaVal = duration\_cast< milliseconds >(system\_clock::now().time\_since\_epoch()) - deltaInit;

    else if (cond == 2)

        cout << "Interval : " << to\_string(deltaVal.count()) << "ms" << endl;}

1. *GitHub Repository*

<https://github.com/iZcy/asd23sortingmethod499769.git>

1. Referensi
2. GIF berbagai algoritma: <https://blogs.cuit.columbia.edu/zp2130/sorting/>
3. Algoritma *Shell Sort*: <https://www.geeksforgeeks.org/shellsort/>
4. GIF *Shell Sort*: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Shell_Sort.gif>
5. Algoritma *Merge Sort*: <https://www.geeksforgeeks.org/merge-sort/>
6. Ilustrasi *Merge Sort*: <https://media.geeksforgeeks.org/wp-content/uploads/20220722205737/MergeSortTutorial.png>
7. GIF *Merge Sort*: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Merge_Sort.gif>
8. Algoritma *Binary Search Tree Sort*:
9. GIF *Binary Search Tree Sort*: <https://blog.penjee.com/5-gifs-to-understand-binary-search-tree/>
10. Algoritma *Radix Sort*: <https://www.geeksforgeeks.org/radix-sort/>
11. GIF *Radix Sort*: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Radix_Sort.gif>
12. Algoritma *Quick Sort*: <https://www.geeksforgeeks.org/quick-sort/>
13. Ilustrasi *Quick Sort*: <https://www.geeksforgeeks.org/quick-sort/>
14. GIF *Quick Sort*: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Quick_Sort.gif>
15. Algoritma *Heap Sort*: <https://www.geeksforgeeks.org/heap-sort/>
16. GIF *Heap Sort*: <http://blogs.cuit.columbia.edu/zp2130/files/2018/12/Heap_Sort.gif>