LAPORAN PRAKTIKUM 5

ANALISIS ALGORITMA

Program Studi S-1 Teknik Informatika

Departemen Ilmu Komputer

Fakultas Matematika dan Ilmu Pengetahuan Alam

Universitas Padjadjaran

2020

DISUSUN OLEH:

AITHRA JUNIA BOUTY

140810180054

**Studi Kasus 5: Mencari Pasangan Titik Terdekat (Closest Pair of Points)**

1. Buatlah program untuk menyelesaikan problem closest pair of points menggunakan algoritma divide & conquer yang diberikan. Gunakan bahasa C++

Program:

#include <bits/stdc++.h>

using *namespace* std;

// A structure to represent a Point in 2D plane

*class* Point

{

*public:*

*int* x, y;

};

/\* Following two functions are needed for library function qsort().

Refer: http://www.cplusplus.com/reference/clibrary/cstdlib/qsort/ \*/

// Needed to sort array of points

// according to X coordinate

*int* compareX(const *void*\* *a*, const *void*\* *b*)

{

    Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;

    return (p1->x - p2->x);

}

// Needed to sort array of points according to Y coordinate

*int* compareY(const *void*\* *a*, const *void*\* *b*)

{

    Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;

    return (p1->y - p2->y);

}

// A utility function to find the

// distance between two points

*float* dist(Point *p1*, Point *p2*)

{

    return sqrt( (p1.x - p2.x)\*(p1.x - p2.x) +

                (p1.y - p2.y)\*(p1.y - p2.y)

            );

}

// A Brute Force method to return the

// smallest distance between two points

// in P[] of size n

*float* bruteForce(Point *P*[], *int* *n*)

{

*float* min = FLT\_MAX;

    for (*int* i = 0; i < n; ++i)

        for (*int* j = i+1; j < n; ++j)

            if (dist(P[i], P[j]) < min)

                min = dist(P[i], P[j]);

    return min;

}

// A utility function to find

// minimum of two float values

*float* min(*float* *x*, *float* *y*)

{

    return (x < y)? x : y;

}

// A utility function to find the

// distance beween the closest points of

// strip of given size. All points in

// strip[] are sorted accordint to

// y coordinate. They all have an upper

// bound on minimum distance as d.

// Note that this method seems to be

// a O(n^2) method, but it's a O(n)

// method as the inner loop runs at most 6 times

*float* stripClosest(Point *strip*[], *int* *size*, *float* *d*)

{

*float* min = d; // Initialize the minimum distance as d

    qsort(strip, size, sizeof(Point), compareY);

    // Pick all points one by one and try the next points till the difference

    // between y coordinates is smaller than d.

    // This is a proven fact that this loop runs at most 6 times

    for (*int* i = 0; i < size; ++i)

        for (*int* j = i+1; j < size && (strip[j].y - strip[i].y) < min; ++j)

            if (dist(strip[i],strip[j]) < min)

                min = dist(strip[i], strip[j]);

    return min;

}

// A recursive function to find the

// smallest distance. The array P contains

// all points sorted according to x coordinate

*float* closestUtil(Point *P*[], *int* *n*)

{

    // If there are 2 or 3 points, then use brute force

    if (n <= 3)

        return bruteForce(P, n);

    // Find the middle point

*int* mid = n/2;

    Point midPoint = P[mid];

    // Consider the vertical line passing

    // through the middle point calculate

    // the smallest distance dl on left

    // of middle point and dr on right side

*float* dl = closestUtil(P, mid);

*float* dr = closestUtil(P + mid, n - mid);

    // Find the smaller of two distances

*float* d = min(dl, dr);

    // Build an array strip[] that contains

    // points close (closer than d)

    // to the line passing through the middle point

    Point strip[n];

*int* j = 0;

    for (*int* i = 0; i < n; i++)

        if (abs(P[i].x - midPoint.x) < d)

            strip[j] = P[i], j++;

    // Find the closest points in strip.

    // Return the minimum of d and closest

    // distance is strip[]

    return min(d, stripClosest(strip, j, d) );

}

// The main functin that finds the smallest distance

// This method mainly uses closestUtil()

*float* closest(Point *P*[], *int* *n*)

{

    qsort(P, n, sizeof(Point), compareX);

    // Use recursive function closestUtil()

    // to find the smallest distance

    return closestUtil(P, n);

}

// Driver code

*int* main()

{

    Point P[] = {{4, 2}, {12, 30}, {40, 50}, {5, 5}, {3, 4}};

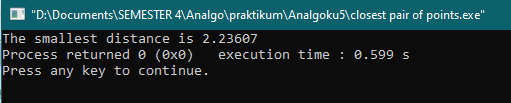
*int* n = sizeof(P) / sizeof(P[0]);

    cout << "The smallest distance is " << closest(P, n);

    return 0;

}

Screenshot:



1. Tentukan rekurensi dari algoritma tersebut, dan selesaikan rekurensinya menggunakan metode recursion tree untuk membuktikan bahwa algoritma tersebut memiliki Big-O (n lg n)

**Jawaban:**

Kompleksitas Waktu

Biarkan kompleksitas waktu dari algoritma di atas menjadi T (n). Mari kita asumsikan bahwa kita menggunakan algoritma pengurutan O (nLogn). Algoritma di atas membagi semua titik dalam dua set dan secara rekursif memanggil dua set. Setelah membelah, ia menemukan strip dalam waktu O (n), mengurutkan strip dalam waktu O (nLogn) dan akhirnya menemukan titik terdekat dalam strip dalam waktu O (n). Jadi T (n) dapat dinyatakan sebagai berikut

T (n) = 2T (n / 2) + O (n) + O (nLogn) + O (n)

T (n) = 2T (n / 2) + O (nLogn)

T (n) = T (n x Logn x Logn)

Catatan :

1. Kompleksitas waktu dapat ditingkatkan menjadi O (nLogn) dengan mengoptimalkan langkah 5 dari algoritma di atas.
2. Kode menemukan jarak terkecil. Dapat dengan mudah dimodifikasi untuk menemukan titik dengan jarak terkecil.
3. Kode ini menggunakan pengurutan cepat yang bisa O (n ^ 2) dalam kasus terburuk. Untuk memiliki batas atas sebagai O (n (Logn) ^ 2), algoritma pengurutan O (nLogn) seperti pengurutan gabungan atau pengurutan tumpukan dapat digunakan

**Studi Kasus 6: Algoritma Karatsuba untuk Perkalian Cepat**

1. Buatlah program untuk menyelesaikan problem fast multiplication menggunakan algoritma divide & conquer yang diberikan (Algoritma Karatsuba). Gunakan bahasa C+

Program:

#include<iostream>

#include<stdio.h>

using *namespace* std;

// FOLLOWING TWO FUNCTIONS ARE COPIED FROM http://goo.gl/q0OhZ

// Helper method: given two unequal sized bit strings, converts them to

// same length by adding leading 0s in the smaller string. Returns the

// the new length

*int* makeEqualLength(string &*str1*, string &*str2*)

{

*int* len1 = str1.size();

*int* len2 = str2.size();

    if (len1 < len2)

    {

        for (*int* i = 0 ; i < len2 - len1 ; i++)

            str1 = '0' + str1;

        return len2;

    }

    else if (len1 > len2)

    {

        for (*int* i = 0 ; i < len1 - len2 ; i++)

            str2 = '0' + str2;

    }

    return len1; // If len1 >= len2

}

// The main function that adds two bit sequences and returns the addition

string addBitStrings( string *first*, string *second* )

{

    string result; // To store the sum bits

    // make the lengths same before adding

*int* length = makeEqualLength(first, second);

*int* carry = 0; // Initialize carry

    // Add all bits one by one

    for (*int* i = length-1 ; i >= 0 ; i--)

    {

*int* firstBit = first.at(i) - '0';

*int* secondBit = second.at(i) - '0';

        // boolean expression for sum of 3 bits

*int* sum = (firstBit ^ secondBit ^ carry)+'0';

        result = (*char*)sum + result;

        // boolean expression for 3-bit addition

        carry = (firstBit&secondBit) | (secondBit&carry) | (firstBit&carry);

    }

    // if overflow, then add a leading 1

    if (carry) result = '1' + result;

    return result;

}

// A utility function to multiply single bits of strings a and b

*int* multiplyiSingleBit(string *a*, string *b*)

{ return (a[0] - '0')\*(b[0] - '0'); }

// The main function that multiplies two bit strings X and Y and returns

// result as long integer

*long* *int* multiply(string *X*, string *Y*)

{

    // Find the maximum of lengths of x and Y and make length

    // of smaller string same as that of larger string

*int* n = makeEqualLength(X, Y);

    // Base cases

    if (n == 0) return 0;

    if (n == 1) return multiplyiSingleBit(X, Y);

*int* fh = n/2; // First half of string, floor(n/2)

*int* sh = (n-fh); // Second half of string, ceil(n/2)

    // Find the first half and second half of first string.

    // Refer http://goo.gl/lLmgn for substr method

    string Xl = X.substr(0, fh);

    string Xr = X.substr(fh, sh);

    // Find the first half and second half of second string

    string Yl = Y.substr(0, fh);

    string Yr = Y.substr(fh, sh);

    // Recursively calculate the three products of inputs of size n/2

*long* *int* P1 = multiply(Xl, Yl);

*long* *int* P2 = multiply(Xr, Yr);

*long* *int* P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr));

    // Combine the three products to get the final result.

    return P1\*(1<<(2\*sh)) + (P3 - P1 - P2)\*(1<<sh) + P2;

}

// Driver program to test aboev functions

*int* main()

{

    printf ("%ld\n", multiply("1100", "1010"));

    printf ("%ld\n", multiply("110", "1010"));

    printf ("%ld\n", multiply("11", "1010"));

    printf ("%ld\n", multiply("1", "1010"));

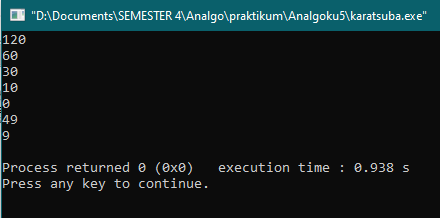
    printf ("%ld\n", multiply("0", "1010"));

    printf ("%ld\n", multiply("111", "111"));

    printf ("%ld\n", multiply("11", "11"));

}

Screenshot:



1. Rekurensi dari algoritma tersebut adalah T (n) = 3T (n / 2) + O (n), dan selesaikan rekurensinya menggunakan metode substitusi untuk membuktikan bahwa algoritma tersebut memiliki Big-O (n lg n)

**Jawaban:**

* Let’s try divide and conquer
* Divide each number into two halves
* x = xH rn/2 + xL
* y = yH rn/2 + yL
* then:

xy = (xH rn/2 + xL) yH rn/2 + yL

= xHyHrn + (xHyL) rn/2 + xL yL

* runtime
* T(n) = 4 T(n/2) + O(n)
* T(n) = O(n2)
* Instead of 4 subproblems, we only need 3 (with the help of clever insight)
* Three subproblems:
* a = xH yH
* d = xL yL
* e = (xH + xL) (yH + yL) – a – d
* Then xy = a rn + e rn/2 + d
* T(n) = 3 T(n/2) + O(n)
* T(n) = O(nlog 3) = O(n1.584…)

**Studi Kasus 7: Permasalahan Tata Letak Keramik Lantai (Tilling Problem)**

1. Buatlah program untuk menyelesaikan problem tilling menggunakan algoritma divide & conquer yang diberikan. Gunakan bahasa C++

Program:

#include <bits/stdc++.h>

using *namespace* std;

// function to count the total number of ways

*int* countWays(*int* *n*, *int* *m*)

{

    // table to store values

    // of subproblems

*int* count[n + 1];

    count[0] = 0;

    // Fill the table upto value n

    for (*int* i = 1; i <= n; i++) {

        // recurrence relation

        if (i > m)

            count[i] = count[i - 1] + count[i - m];

        // base cases

        else if (i < m)

            count[i] = 1;

        // i = = m

        else

            count[i] = 2;

    }

    // required number of ways

    return count[n];

}

// Driver program to test above

*int* main()

{

*int* n = 7, m = 3;

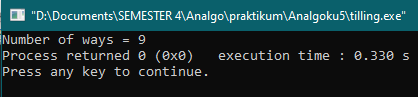
    cout << "Number of ways = "

         << countWays(n, m);

    return 0;

}

Screenshot:



1. Relasi rekurensi untuk algoritma rekursif di atas dapat ditulis seperti di bawah ini. C adalah konstanta. T (n) = 4T (n / 2) + C. Selesaikan rekurensi tersebut dengan Metode Master

**Jawaban:**

Kompleksitas Waktu:

Relasi perulangan untuk algoritma rekursif di atas dapat ditulis seperti di bawah ini. C adalah konstanta.

T (n) = 4T (n / 2) + C

Rekursi di atas dapat diselesaikan dengan menggunakan Metode Master dan kompleksitas waktu adalah O (n2)

Pengerjaan algoritma Divide and Conquer dapat dibuktikan menggunakan Mathematical Induction. Biarkan kuadrat input berukuran 2k x 2k di mana k> = 1.

Kasus Dasar: Kita tahu bahwa masalahnya dapat diselesaikan untuk k = 1. Kami memiliki 2 x 2 persegi dengan satu sel hilang.

Hipotesis Induksi: Biarkan masalah dapat diselesaikan untuk k-1.

Sekarang perlu dibuktikan untuk membuktikan bahwa masalah dapat diselesaikan untuk k jika dapat diselesaikan untuk k-1. Untuk k, ditempatkan ubin berbentuk L di tengah dan memiliki empat subsqure dengan dimensi 2k-1 x 2k-1 seperti yang ditunjukkan pada gambar 2 di atas. Jadi jika dapat menyelesaikan 4 subskuares, dapat menyelesaikan kuadrat lengkap.