**Analisis Algoritma**

Tugas 5



­­

**Dibuat oleh:**

Muhammad Iqbal Alif Fadilla

140810180020

**Universitas Padjadjaran**

**Fakultas Matematika dan Ilmu Pengetahuan**

**2020**

**Studi Kasus 5**

1. Program Closest Pair of Points C++
2. /\*
3. Nama        : Muhammad Iqbal Alif Fadilla
4. Kelas       : B
5. NPM         : 140810180020
6. Deskripsi   : Closest Pair of Points
7. \*/
9. #include <bits/stdc++.h>
10. **using** **namespace** std;
12. **class** Point
13. {
14. **public**:
15. **int** x, y;
16. };
18. **int** compareX(**const** **void** \*a, **const** **void** \*b)
19. {
20. Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;
21. **return** (p1->x - p2->x);
22. }
24. **int** compareY(**const** **void** \*a, **const** **void** \*b)
25. {
26. Point \*p1 = (Point \*)a, \*p2 = (Point \*)b;
27. **return** (p1->y - p2->y);
28. }
30. **float** dist(Point p1, Point p2)
31. {
32. **return** sqrt((p1.x - p2.x) \* (p1.x - p2.x) +
33. (p1.y - p2.y) \* (p1.y - p2.y));
34. }
36. **float** bruteForce(Point P[], **int** n)
37. {
38. **float** min = FLT\_MAX;
39. **for** (**int** i = 0; i < n; ++i)
40. **for** (**int** j = i + 1; j < n; ++j)
41. **if** (dist(P[i], P[j]) < min)
42. min = dist(P[i], P[j]);
43. **return** min;
44. }
46. **float** min(**float** x, **float** y)
47. {
48. **return** (x < y) ? x : y;
49. }
51. **float** stripClosest(Point strip[], **int** size, **float** d)
52. {
53. **float** min = d; // Initialize the minimum distance as d
55. qsort(strip, size, **sizeof**(Point), compareY);
57. **for** (**int** i = 0; i < size; ++i)
58. **for** (**int** j = i + 1; j < size && (strip[j].y - strip[i].y) < min; ++j)
59. **if** (dist(strip[i], strip[j]) < min)
60. min = dist(strip[i], strip[j]);
62. **return** min;
63. }
65. **float** closestUtil(Point P[], **int** n)
66. {
67. // If there are 2 or 3 points, then use brute force
68. **if** (n <= 3)
69. **return** bruteForce(P, n);
71. // Find the middle point
72. **int** mid = n / 2;
73. Point midPoint = P[mid];
75. **float** dl = closestUtil(P, mid);
76. **float** dr = closestUtil(P + mid, n - mid);
78. // Find the smaller of two distances
79. **float** d = min(dl, dr);
81. Point strip[n];
82. **int** j = 0;
83. **for** (**int** i = 0; i < n; i++)
84. **if** (abs(P[i].x - midPoint.x) < d)
85. strip[j] = P[i], j++;
87. **return** min(d, stripClosest(strip, j, d));
88. }
90. **float** closest(Point P[], **int** n)
91. {
92. qsort(P, n, **sizeof**(Point), compareX);
94. **return** closestUtil(P, n);
95. }
97. // Driver code
98. **int** main()
99. {
100. Point P[] = {{12, 1}, {33, 21}, {54, 36}};
101. **int** n = **sizeof**(P) / **sizeof**(P[0]);
102. cout << "The smallest distance is " << closest(P, n);
103. **return** 0;
104. }



1. Kompleksitas Waktu

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

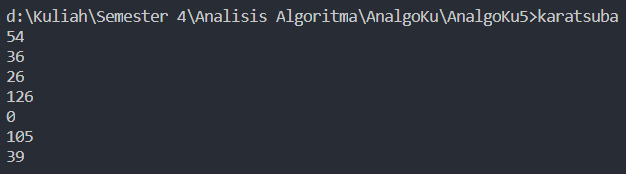
- Kompleksitas waktu dapat ditingkatkan menjadi O (nLogn) dengan mengoptimalkan langkah 5 dari algoritma di atas.

- Kode menemukan jarak terkecil. Dapat dengan mudah dimodifikasi untuk menemukan titik dengan jarak terkecil.

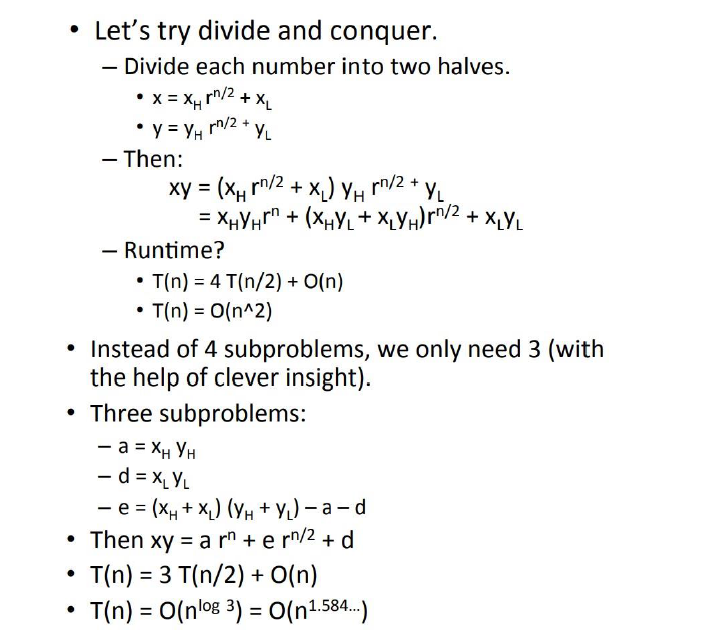
- Kode ini menggunakan pengurutan cepat yang bisa O (n ^ 2) dalam kasus terburuk. Untuk memiliki batas atas sebagai O (n (Logn) ^ 2), algoritma pengurutan O (nLogn) seperti pengurutan gabungan atau pengurutan tumpukan dapat digunakan

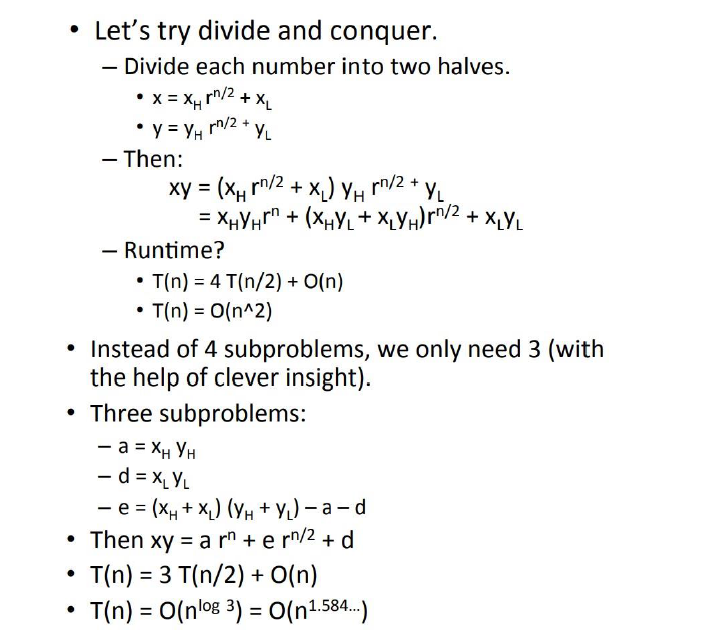
**Studi Kasus 6**

1. Program Karatsuba C++
2. /\*
3. Nama        : Muhammad Iqbal Alif Fadilla
4. Kelas       : B
5. NPM         : 140810180020
6. Deskripsi   : Karatsuba Fast Multiplication Algorithm
7. \*/
8. #include <iostream>
9. #include <stdio.h>
11. **using** **namespace** std;
13. **int** makeEqualLength(string &str1, string &str2)
14. {
15. **int** len1 = str1.size();
16. **int** len2 = str2.size();
17. **if** (len1 < len2)
18. {
19. **for** (**int** i = 0; i < len2 - len1; i++)
20. str1 = '0' + str1;
21. **return** len2;
22. }
23. **else** **if** (len1 > len2)
24. {
25. **for** (**int** i = 0; i < len1 - len2; i++)
26. str2 = '0' + str2;
27. }
28. **return** len1; // If len1 >= len2
29. }
31. // The main function that adds two bit sequences and returns the addition
32. string addBitStrings(string first, string second)
33. {
34. string result; // To store the sum bits
36. // make the lengths same before adding
37. **int** length = makeEqualLength(first, second);
38. **int** carry = 0; // Initialize carry
40. // Add all bits one by one
41. **for** (**int** i = length - 1; i >= 0; i--)
42. {
43. **int** firstBit = first.at(i) - '0';
44. **int** secondBit = second.at(i) - '0';
46. // boolean expression for sum of 3 bits
47. **int** sum = (firstBit ^ secondBit ^ carry) + '0';
49. result = (**char**)sum + result;
51. // boolean expression for 3-bit addition
52. carry = (firstBit & secondBit) | (secondBit & carry) | (firstBit & carry);
53. }
55. // if overflow, then add a leading 1
56. **if** (carry)
57. result = '1' + result;
59. **return** result;
60. }
62. // A utility function to multiply single bits of strings a and b
63. **int** multiplyiSingleBit(string a, string b)
64. {
65. **return** (a[0] - '0') \* (b[0] - '0');
66. }
68. // The main function that multiplies two bit strings X and Y and returns
69. // result as long integer
70. **long** **int** multiply(string X, string Y)
71. {
72. // Find the maximum of lengths of x and Y and make length
73. // of smaller string same as that of larger string
74. **int** n = makeEqualLength(X, Y);
76. // Base cases
77. **if** (n == 0)
78. **return** 0;
79. **if** (n == 1)
80. **return** multiplyiSingleBit(X, Y);
82. **int** fh = n / 2;    // First half of string, floor(n/2)
83. **int** sh = (n - fh); // Second half of string, ceil(n/2)
85. // Find the first half and second half of first string.
86. // Refer http://goo.gl/lLmgn for substr method
87. string Xl = X.substr(0, fh);
88. string Xr = X.substr(fh, sh);
90. // Find the first half and second half of second string
91. string Yl = Y.substr(0, fh);
92. string Yr = Y.substr(fh, sh);
94. // Recursively calculate the three products of inputs of size n/2
95. **long** **int** P1 = multiply(Xl, Yl);
96. **long** **int** P2 = multiply(Xr, Yr);
97. **long** **int** P3 = multiply(addBitStrings(Xl, Xr), addBitStrings(Yl, Yr));
99. // Combine the three products to get the final result.
100. **return** P1 \* (1 << (2 \* sh)) + (P3 - P1 - P2) \* (1 << sh) + P2;
101. }
103. // Driver program to test aboev functions
104. **int** main()
105. {
106. printf("%ld\n", multiply("1001", "0110"));
107. printf("%ld\n", multiply("1100", "0011"));
108. printf("%ld\n", multiply("1101", "0010"));
109. printf("%ld\n", multiply("1001", "1110"));
110. printf("%ld\n", multiply("0000", "1011"));
111. printf("%ld\n", multiply("0111", "1111"));
112. printf("%ld\n", multiply("0011", "1101"));
113. }



1. Komplesitas Waktu





**Studi Kasus 7**

1. Program Tilling C++
2. /\*
3. Nama        : Muhammad Iqbal Alif Fadilla
4. Kelas       : B
5. NPM         : 140810180020
6. Deskripsi   : Tilling Problem
7. \*/
8. #include <bits/stdc++.h>
10. **using** **namespace** std;
12. // function to count the total number of ways
13. **int** countWays(**int** n, **int** m)
14. {
16. // table to store values
17. // of subproblems
18. **int** count[n + 1];
19. count[0] = 0;
21. // Fill the table upto value n
22. **for** (**int** i = 1; i <= n; i++)
23. {
24. // recurrence relation
25. **if** (i > m)
26. count[i] = count[i - 1] + count[i - m];
28. // base cases
29. **else** **if** (i < m)
30. count[i] = 1;
32. // i = = m
33. **else**
34. count[i] = 2;
35. }
37. // required number of ways
38. **return** count[n];
39. }
41. // Driver program to test above
42. **int** main()
43. {
44. **int** n = 9, m = 1;
45. cout << "Number of ways = "
46. << countWays(n, m);
47. **return** 0;
48. }



// n adalah ukuran kotak yang diberikan, p adalah lokasi sel yang hilang

Tile (int n, Point p)

1. Kasus dasar: n = 2, A 2 x 2 persegi dengan satu sel yang hilang tidak ada apa-apanya tapi ubin dan bisa diisi dengan satu ubin.
2. Tempatkan ubin berbentuk L di tengah sehingga tidak menutupi subsquare n / 2 \* n / 2 yang memiliki kuadrat yang hilang. Sekarang keempatnya subskuen ukuran n / 2 x n / 2 memiliki sel yang hilang (sel yang tidak perlu diisi). Lihat gambar 2 di bawah ini.
3. Memecahkan masalah secara rekursif untuk mengikuti empat. Biarkan p1, p2, p3 dan p4 menjadi posisi dari 4 sel yang hilang dalam 4 kotak.

* Ubin (n / 2, p1)
* Ubin (n / 2, p2)
* Ubin (n / 2, p3)
* Ubin (n / 2, p3)

1. 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)

Bagaimana cara kerjanya?

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.