**ASSIGNMENT 1**

**ANALISIS ALGORITMA**

****

**DISUSUN OLEH**

**ILHAM MUHARAM**

**140810170046**

**KELAS B**

**PROGRAM STUDI S-1 TEKNIK INFORMATIKA**

**UNIVERSITAS PADJADJARAN**

**JATINANGOR**

**2019**

**QUESTIONS**

Case 1:

Gale and Shapley published their paper on the Stable Matching Problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.

Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that an assignment of students to hospitals is stable if neither of the following situations arises.

* First type of instability: There are students s and s’, and a hospital h, so that
* s is assigned to h, and
* s’ is assigned to no hospital, and
* h prefers s’ to s.
* Second type of instability: There are students s and s’, and hospitals t’ and h’, so that
* s is assigned to h, and
* s’ is assigned to h’, and
* h prefers s’ to s, and - s’ prefers h to h’.

So, we basically have the Stable Matching Problem, except that

1. hospitals generally want more than one resident, and
2. there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an algorithm to find one.

Case 2:

Take the following list of functions and arrange them in ascending order of growth rate. That is, if function g(n) immediately follows function f(n) in your list, then it should be the case that f(n) is O(g(n)).

* g1(n) =
* g2(n) = 2n
* g3(n) =
* g4(n) = n (log3) n
* g5(n) = n1 org
* g6(n) =
* g7(n) =

Case 3:

Consider the following basic problem. You’re given an array A consisting of n integers A[1], A[2], … A[n]. You’d like to output a two-dimensional n-by-n array B in which B[i,j] (for i <j) contains the sum of array entries A[i] through A[j]--that is, the sum A[i] +A[i + 1] + .. + A[j]. (The value of array entry B[i,j] is left unspecified whenever i ≥ j, so it doesn’t matter what is output for these values.)

Here’s a simple algorithm to solve this problem.

For i = 1, 2,...,n

For j = i+1, i+2, .. n

Add up array entries A[i] through A[j]

Store the result in B[i,j]

Endfor

Endfor

1. For some function f that you should choose, give a bound of the form O(f(n)) on the running time of this algorithm on an input of size n (i.e., a bound on the number of operations performed by the algorithm).
2. For this same function f, show that the running time of the algorithm on an input of size n is also Ω(f(n)). (This shows an asymptotically tight bound of Θ(f(n)) on the running time.)
3. Although the algorithm you analyzed in parts (a) and (b) is the most natural way to solve the problem--after all, it just iterates through the relevant entries of the array B, filling in a value for each—it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time O(g(n)), where lim →( ) / ( )=0.

Case 4:

Is Is Give evidence (proof) according to the definition of Big-O notation, for answers yes or no!

**ANSWERS**

Case 1:

Pembuktian:

Untuk jenis ketidakstabilan pertama, misalkan ada siswa s dan siswa s’, dan rumah sakit. Jika h lebih suka s’ dari pada s, maka h akan menawarkan posisi ke s’ sebelum menawarkan satu ke s; Posisi s' ke s , maka posisi h di beberapa rumah sakit, tidak akan bebas pada akhirnya (Kontradiksi).

Untuk ketidakstabilan kedua kalinya, anggaplah bahwa (hi, sj) adalah pasangan yang menyebabkan ketidakstabilan. Kemudian h, pasti menawarkan posisi untuk sj, karena jika tidak memiliki p, penduduk yang semuanya lebih suka sj. Selain itu, sj pasti telah menolak h, mendukung beberapa hk yang dia pilih; dan karena itu sj harus berkomitmen untuk beberapa hl (mungkin berbeda dari hk) yang juga dia sukai.

Algoritma:

While some hospital hi has available positions

hi offersa position to the next student sj on its preference list

if sj is free then

sj accept the offer

else (sj is already committed to a hospital hk)

if sj prefers hk to hi then

sj remains committed to hk

else sj becomes committed to hi

the number of available positions at hk increases by one

the number of available positions at hi increases by one

Case 2:

* g1 muncul sebelum g5. Ini adalah soal yang diselesaikan dimana kita lihat . Jika diambil logaritmanya, kita membandingkan √log n dengan log n + log (log n) log n; ubah variabel melalui z = log n, ini adalah √z = versus z + log z z.
* g5 muncul sebelum g3, karena (log n)3 tumbuh lebih cepat dari log n. (Keduanya polinomial dalam log n, tetapi (log n)3 memiliki tingkat yang lebih besar.)
* g3 muncul sebelum g4: Membagi keduanya dengan n, bandingkan (log n)3 dengan n1/3, atau (ambil akar pangkat tiga), log n dengan n1/9. Sekarang kita menggunakan fakta bahwa logaritma tumbuh lebih lambat daripada eksponensial.
* g4 muncul sebelum g2, karena polinomial tumbuh lebih lambat daripada eksponensial.
* g2 datang sebelum g7: Bandingkan n ke n2, dan n2 adalah polinomial tingkat yang lebih besar.
* g7 hadir sebelum g6: Bandingkan n2 hingga 2n, dan polinomial tumbuh lebih lambat daripada eksponensial.

Case 3:

1. Untuk f(n) = n3, loop luar dari algoritma tersebut berjalan selama n iterasi, loop dalam berjalan selama n iterasi tiap eksekusi. Jadi, baris kode yang menambah entri array A[i] lewat A[j] (untuk sembarang tiap i dan j) dieksekusi sebanyak n2. Menambah entri A[i] lewat A[j] mengambil operasi O(j – i + 1), yang selalu sebanyak O(n). Hasil di R[i, j] butuh waktu konstan. Jadi, running time dari seluruh Algoritma sebanyak (n2 . O(n)), sehingga menjadi O(n3).
2. Misal waktu eksekusi Algoritma kapan i n/4 dan j 3n/4. Dalam kasus ini, j - i + 1 3n/4 – n/4 + 1 > n/2. Jadi, menambah entri array A[t] lewat A[j] akan membutuhkan setidaknya operasi n/2, sejak n/2 bertambah. Berapa banyak waktu untuk mengeksekusi dalam kasus tersebut? Ada (n/4)2 pasangan (i, j) dengan i n/4 dan j3n/4. Algoritma tersebut menghitung semuanya, dan melakukan operasi n/2 tiap pasang. Jadi, Algoritma melakukan operasi setidaknya operasi (n/2 . (n/4)2 = n3/32. (ini adalah Ω(n3)).
3. Dari Algoritma:

For i = 1, 2, …, n

Set B[i, i + 1] to A[i] + A[i + 1]

For k = 2, 3, …, n-1

For i = 1, 2, …, n – k

Set j = i + k

Set B[i, j] to be B[i, j - 1] + A[j]

Algoritma ini berfungsi apabila B[i, j - 1] telah dihitung dalam iterasi sebelumnya di loop luar, dimana k = j – 1 – i, sejak j – 1 – i < j – i. Itu menghitung B[i, i + 1] untuk semua i dengan menjumlahkan A[i] dengan A[i + 1]. Ini membutuhkan operasi O(n). Untuk setiap k, itu menghitung semua B[i, j] untuk j – i = k dengan set B[i, j - 1] + A[j]. Untuk tiap k, Algoritma ini melakukan operasi O(n) sejak kebanyakan n B[i, j] seperti j – i = k. Kurang dari nilai n dari k saling iterasi, jadi Algoritma ini memiliki running time O(n2).

Case 4:

karena

Misalkan = O () Kemudian ada konstanta c sedemikian rupa sehingga untuk n melampaui beberapa n0, <= c . Membagi kedua sisi dengan , didapat <c. Tidak ada nilai untuk c dan n0 yang dapat membuat ini benar, jadi hipotesisnya salah dan! = *O* ().