演算法Homework 20190919

山下夏輝（Yamashita Natsuki）

R08922160

Problem5

（1）

(a): False

Counterexample

When f(n) = n2, g(n) = n

n2 + n ≠ O(min(f(n), g(n))) = O(n)

(b): True

(c): True

(d): False

Counterexample

When f(n) = 2n

2n ≠ 2n/2

(e): False

Counterexample

Assume that log2(n!) ≥ c\*n2

When n = 1

0 ≥ c\*1

Then, there is not any positive constant c.

(2)

(a)

At first, c = 108

T(n) =6T(n/3)+cn

=62T(n/32)+3cn

=63T(n/33)+5cn

…

=6kT(n/3k)+(2k-1)cn

When kth expansion, T(n/3k) = T(1).

So, 3k=n

Then,

T(n) =n log[3]2 + c n log3n

=Θ(n log n)

(b)

At first, c = 24

T(n) = T(n/3)+T(n/4)+T(n/12)

The recursion-tree of it is below.

Then the complexities of each layers are below.

1st: cn

2nd: 8/12\*cn

3rd: 82/122\*cn

4th: 83/123\*cn

…

Then,

T(n) =(1 + 8/12 + 82/122 + 83/123 + … )cn

= 1/(1 - 8/12)cn

= 3cn

= Θ(n)

(c)

T(n) = √nT(√n) + 2nlogn

T(n)/n= (√nT(√n) + 2nlogn)/n

= T(√n)/n + 2logn

Assume that S(n) = T(√n)/n

S(n) = S(√n) + 2logn

Change variable as k = logn

S(2k) = S(2k/2) +2k

Assume that SS(k) = S(2k)

SS(k) = SS(k/2) + 2k

Use master theorem

SS(k) = SS(k/2) + 2k

nlog[2]1=0 < O(n) and also c\*2k for c<1

So,

SS(k) = Θ(k)

S(2k) = Θ(logk)

S(n) = Θ(loglogn)

T(n) = Θ(n loglogn)

(d)

At first, 4n/logn<4n

Eg. n=1 0<4

n=4 8<16

So that O(n)

And also,

-4n/logn = Ω(nlog[2]2+ε) with ε =0.00001.

-4n/2logn/2 < c 4n/logn with c =0.9.

So,

T(n) = Θ(nlogn)

Problem6

(1)

Pseudo Code and Complexities

Merge(B[], l, m, h)

inv\_cnt=0

i=low

j=mid+1

k=low

while i to m && j to h //inversion count

O(n)

if B[i] <= B[l]

temp[k++] = B[i++]

else

temp[k++] = B[j++]

inv\_cnt += (mid- i + 1)

while i to m //sort left side. right side is already sorted

O(n)

n

temp[k++] = B[i++]

i = low //sub sorted elements for B[]

for i to h

O(n)

n

B[i] = temp[i]

return inv\_cnt

Inversion(n, B[])

if i==j // base case

return 0

else // recursive case

O(nlogn)

m = (i+j)/2

O(nlogn)

cnt += Mergesort(B[], i, m)

cnt += Mergesort(B[], m+1, j)

cnt += Merge(B[], i, m+1, j)

return cnt

(2)

As (1) show, the function of Inversion(n, B[]) runs in O(nlogn) and the function of Merge(B[], l, m, h) which is called in Inversion(n, B[]) runs in O(n). So the sum of the complexities T(n) is O(n log n).

T(n) = O(n log n) + O(n)

= O(n log n)

(3)

About bubble sort, given a sequence of unique numbers A = a1, a2,…,an. A[k] indicates the kth element in A. c[k] indicates the number of elements which is bigger than A[k] in the left side of A[k]. Then, you can get the number of exchanges, calculating Σc[k].

About inversion count with mergesort, given a sequence of unique numbers B = b1, b2,…,bn. B[k] indicates the kth element in B. c[k] indicates the number of elements which is bigger than B[k]. By definition of inversion, c[k] is the number of inversion as well. Then Σc[k] is the number of inversion in B.

So, the number of exchanges when performing bubble sort on the sequence S is equal to the number of inversions in S.

(4)

Pseudo code

Pseudo Code and Complexities

Merge(B[], l, m, h)

inv\_cnt=0

i=low

j=mid+1

k=low

while i to m && j to h //inversion count

O(n)

if B[i] <= B[l]

temp[k++] = B[i++]

else

temp[k++] = B[j++]

inv\_cnt += (mid- i + 1)

while i to m //sort left side. right side is already sorted

O(n)

n

temp[k++] = B[i++]

i = low //sub sorted elements for B[]

for i to h

O(n)

n

B[i] = temp[i]

return inv\_cnt

Calculate\_Inversion(N, B[], constraints)

if i==j // base case

return 0

else // recursive case

arrange B[] according to constraints.

O(n)

m = (i+j)/2

O(nlogn)

cnt += Mergesort(B[], i, m)

cnt += Mergesort(B[], m+1, j)

O(nlogn)

cnt += Merge(B[], i, m+1, j)

O(n)

return cnt

Calculate\_Holizontal\_Line(N, B[], constraints)

//swap the numbers at bottom, based on constraints,

Calculate\_Inversion(N, B[], constraints)

The number of inversion in B is the minimum number of vertical lines because it is the same meaning that you exchange the adjacent numbers at the bottom for the certain times as much as the number of inversion. And also this is the same meaning as that When a horizontal line is encountered, the player transit to another vertical line and then continue going down.

So you get the minimam number of holizontal lines with the pseudo code above which calculates the number of inversions.

(5)

Pseudo code

Pseudo Code and Complexities

Merge(B[], l, m, h)

inv\_cnt=0

i=low

j=mid+1

k=low

while i to m && j to h //inversion count

O(n)

if B[i] <= B[l]

temp[k++] = B[i++]

else

temp[k++] = B[j++]

inv\_cnt += (mid- i + 1)

while i to m //sort left side. right side is already sorted

O(n)

n

temp[k++] = B[i++]

i = low //sub sorted elements for B[]

for i to h

O(n)

n

B[i] = temp[i]

return inv\_cnt

Calculate\_Inversion(N, B[], constraints)

if i==j // base case

return 0

else // recursive case

O(n)

arrange B[] according to constraints

for k to (N – the number of constraints)

O(n)

put not-constraints numbers from smallest to largest into the empty space in B[]

m = (i+j)/2

O(nlogn)

cnt += Mergesort(B[], i, m)

cnt += Mergesort(B[], m+1, j)

O(nlogn)

cnt += Merge(B[], i, m+1, j)

O(n)

return cnt

Calculate\_Holizontal\_Line(N, B[], constraints)

//swap the numbers at bottom, based on constraints,

Calculate\_Inversion(N, B[], constraints)

The number of inversion in B is the minimum number of vertical lines as I mentioned in (4)

Especially in this case, you can reduce the number of inversions as much as you can when you put not-constraints numbers from smallest to largest into the empty space in B[]. In this way, you get the minimam number of holizontal lines with the pseudo code above which calculates the number of inversions.

Problem 7

(1)

N denotes the number of the board.

k is the positive integer.

When N = ｍ\*2k (k is the positive integer), this method below can determine whether the problem is solvable in constant time.

Method()

If N – (ｍ\*2k) = 0

Return ”solvable”

Else

Return “unsolvable”

(2)

N denotes the number of the board.

i denotes the leftmost position of the block or blocks. (Assume i = 0)

j denotes the rightmost position of the block or blocks.

k is the positive integer.

Unfold()

If i%2 !=0

Unfold to left

Print “Unfold to left and block is on” + i-2k  + “-” + j

Else

Unfold to right

Print “Unfold to left and block is on” + i + “-” + j-2k

For k=1, k to log N

if i = (2k, 2k+2, 2 k+4…)

Unfold to left

Print “Unfold to left and block is on” + i-2k  + “-” + j

else j = (2k, 2k+2, 2 k+4…)

Unfold to right

Print “Unfold to left and block is on” + i + “-” + j-2k

return

(3)

Complexities of this problem depends on the position of the initial block. It is the lowest complexity when the initial block is on the leftmost or rightmost, because there is only one direction to unfold the block or blocks. On the other hand, the highest complexity come out when the initial block is on the nearest center.

About O(d1+d2), it is the fixed value in the certain length of the board wherever the position of the block is. Also this equation holds: d1+d2 = N -1 in this task condition.

Here, the number of ways to unfold the block or blocks is below.

First unfold 21

Second unfold 22

Third unfold 23

…

Nth unfold 20

Then the sum of the complexity is 2logN-1. 2logN denotes the length of the board. Therefore, this equation holds: 2logN-1 = d1+d2.

So, there are O(d1+d2) possibilities.

(4)

(5)

(6)