ADA HW1

b07902126 資工二 謝宗儒

Pb5.

(1)

(a)False, if f(n)=n^2-n and g(n)=n, then f(n)+g(n) =f(n) !=O(min (f(n),g(n))

(b)True,

(c)True,

(d)True,

(e)False, big omega(n^2)>=cn^2 (c belongs to R)

🡪log(n!)/log2=big omega(n^2) >=cn^2

🡪n!>=2^(cn^2)---------(\*)

But when n=1, \* becomes 1>=2^c and there is no positive constant c to let the equation be held. Therefore, we know that the statement of log(n!)/log2=big omega(n^2) is false.

(2)

(a) T(n)=6T(n/3)+108n=6[6T[n/9]+36n]+108n=36T(n/9)+324n

=36[6T(n/27)+12n]+324n=216T(n/27)+756n

=216[T(n/81)+4n]+756n

….

=6^k\*T(1)+(2^k-1)\*n, (6^k=n🡪 k=(logn/log2)/(log6/log2) )

🡪T(n)=nT(1)+[n\*2^(log2/log6)-1]\*n

🡪T(n)=O(n)+O(n^2)

🡪T(n)=O(n^2)

(b)

T(n)=T(n/3)+T(n/4)+T(n/12)+24n=[T(n/9)+T(n/12)+T(n/36)+8n]+

[T(n/12)+T(n/16)+T(n/48)+6n]+[T(n/36)+T(n/48)+T(n/144)+2n]+24n

=T(n/9)+2T(n/12)++T(n/16)+2T(n/36)+2T(n/48)+T(n/144)+40n

…..

=T(4)+T(3)+T(1)+24n/(1-(1/3+1/4+1/12))=T(4)+T(3)+T(1)+36n

=theta(n)

(c)

Let S(n)= T(n)/n 🡪 nS(n)=T(n)

nS(n) = sqrt(n)S(sqrt(n))\*sqrt(n)+2n\*lgn

🡪S(n)=S( sqrt(n) ) +2lgn

Let k=lgn🡪 n=2^k

🡪S(2^k)=S(2^(k/2))+2k

Let G(k)=S(2^k)

🡪G(k)=G(k/2)+2k=[G(k/4)+2k]+2k=G(k/4)+4k

………

=G(1)+k^2🡪G(k)=theta(k^2)🡪S(2^k)=theta(k^2)🡪S(n)=theta((lgn)^2)

🡪T(n)=n\*theta((lgn)^2)=theta(n(lgn)^2)

(d)

k=lgn🡪2^k=n

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

🡪T(2^k)=2T(2^k/2)+4\*2^k/k=2[2T(2^k/4)+2\*2^k/k]+4\*2^k/k=4T(2^k/4)+8\*2^k/k

=4[2T(2^k/8)+2^k/k]+8\*2^k/k=8T(2^k/8)+16\*2^k/k

……

=2^k\*T(1)+2^(k+1)\*2^k/k=n\*T(1)+2n\*n/lgn=theta(n^2/lgn)

Pb6.

(1)Set a variable “count”=0 to record the number of inversion the sequence. Recursively divide the sequence until the number of elements in a subsequence being 1 and then do merge-sort(ascending). While doing merge-sort, if leftsub[i] is larger than rightsub[j], count+= j. After doing so, “count” will be the number of inversion of the sequence.

(2)It is almost the same with merge-sort whose time complexity is O( nlogn).

My algorithm just do addition operation that costs O(1) while some of comparisons in merge-sort occur. O(nlogn)\*O(1) remains O(nlogn).

(3)Since bubble-sort(ascending) keeps comparing the elements from left to right and does exchanges if the left one is larger than the right one, every number will do exchange with any smaller numbers at its right side. That result is equal to the definition of the number of inversion which counts the pairs of bi and bj for any i < j and bi > bj in the sequence.

(4) Set an array destination[N] and record the constraints for corresponding points in destination[]. Then, do merge-sort(ascending) with destination[] by the method using in (1) to count the inversions and that result will be the answer, costing O(NlogN).

The idea of the method is that count horizontal lines from the result to the origin.  
I record the constraints in destination[] as final position and do merge-sort to trace all players’ path from their final position to their original position by updating the exchanging player position. Each inversion is equal to one position-exchange and that means one horizontal line is needed. Therefore, after all players reach their beginning position, the amount of horizontal lines is found.

(5) Set an array destination[N] and record the constraints for corresponding points in destination[]. After that, put the numbers weren’t used ascendingly to the empty elements in destination from 0 to N. Then, do merge-sort(ascending) with destination[] by the method using in (1) to count the inversions and that result will be the answer, costing O(NlogN).

The idea of the method is that count horizontal lines from the result to the origin.  
I record the constraints in destination[] as final position and set all empty elements’ destination as their origin(so that if there are no horizontal lines in need to meet the constraints, it won’t add any line). Then, do merge-sort to trace all players’ path from their final position to their original position by updating the exchanging player position. Each inversion is equal to one position-exchange and that means one horizontal line is needed. Therefore, after all players reach their beginning position, the amount of horizontal lines is found.

Pb7

(1) Transform N to float. Let (float) a=logN/log2 and (int) b =logN/log2.

If a-b=0, this problem is solvable.

Else, this problem is unsolvable.

(2)Set the length of the board = N, the initial position left bound of block = pl

, the initial position right bound of block = pr and the length of the block = L.

If (pl-1)%2=1

//Unfold to left side

pl-L

L\*2

Print unfold operations

Else if (N-pr)%2=1

//Unfold to right side

pr+L

L\*2

Print unfold operations

While solved=0

If [(pl-1)>0] && (pl-1 < N-pr)

If (log(pl-1)/logL)%2==1

pl-L

L\*2

Print unfold operations

Else

pr+L

L\*2

Print unfold operations

Else if N-pr>0

pr+L

L\*2

Print unfold operations

Else

solved=1

(3)

We can totally unfold log(d1+d2+1)/log2 times. One unfold produce one possibility. Generally, each unfold has at most two options and at least one, so there are no more than 2^( log(d1+d2+1)/log2 ) = (d1+d2+1) possibilities of the status. Therefore, we prove that there are O(d1+d2) possibilities of the status.

(4)

Let B[i] (i=1~n) keep blocks’ left bound position from left to right.

L[i] (i=1~n) keep each blocks’ length

board[j] (j=1~N) =false (if this position is covered by a block’s left bound or right bound set true)

For each B[i], unfold to its limitation(In board and don’t overlap other blocks’ initial position) and record each step.

Ex. B[1]=1, L[1]=1, B[2]=9, L[2]=1, N=9,

Since B[1] can unfold to reach board[2], so set board[2]=true.

And then unfold to reach board[4], so set board[4]=true.

And then unfold to reach board[8], so set board[8]=true.

Unfold to reach board[16] will overlap B[2], so stop.

For block 1, check if it can cover the board’s left bound(board[1]=true), output the operation, else, the problem is unsolvable. Then, keep unfolding and record the path in board until it will overlap B[2].

For block i (1<i<=n),

First, check if board[ B[i] - 1] = false, unfold to left side. Keep unfold to left side until one more unfold will overlap board[ B [i-1] ] (case is unsolvable) or the position of the block’s left bound – 1 is true(find the right possibility ). If find true, output the operation.

Second, do the same thing as B[1] (try to unfold to right side).

For block n, this block should also check if it can cover the board’s right bound (board[N]=true), output the operation, else, the problem is unsolvable.

(5)

I solve the problem by dividing it into blocks’ cases. The problem will be solved only when each block find its best unfold status(connect with other near block and for 1 and N reach the bounds of board).

I use board[] to keep blocks’ all unfold possibilities, but don’t really unfold the blocks to the status right away. An correct operation will only be confirmed when a block’s left-unfold status “connect” with the previous block’s one of right-unfold possibilities. Therefore, I can check whether a block’s certain unfold status is well-connected by inquiring board[] with O(1).

(6)

Since I use board[] to record each block’s all unfold status possibilities, as (3) proved, there are at most di1+di2+1 possibilities for each block need to be check. After go through all blocks, they will at most check 2N-(d11+dN2) times for each block i>1 has to left-unfold until connect with B[i-1] and at least check N times for each block i>1 don’t need to left-unfold because it has already connected with the previous block’s one of possibilities.

Conclusively, this algorithm needs to check at least N times, at most 2N-(d11+dN2) times and each check cost constant time. Therefore, it will be theta(N).