11.anotherDP

$$Smallest(l) = \begin{cases} 0, if\ l = 0; \\ (A[1])^2, if\ l = 1; \\ \min\limits_{0 \le k < l} \{Smallest(k) + (A[k+1] + \cdots + A[1])^2\}, otherwise; \end{cases}$$

11.50 Dynamic Programming, lifting

a)
We need two specifications. One for when the root is blue, and one for green. Each has different base cases too.

// case blue

$$Bbest(T) = \begin{cases} T.dat, if\ T\ is\ a\ leaf; \\ \sum\limits_{v\ in\ Child[T]} Gbest(v), otherwise; \end{cases}$$

// case green

$$Gbest(T) = \begin{cases} 0, if\ T\ is\ a\ leaf; \\ \sum\limits_{v\ in\ Child[T]} \max\{Gbest(v), Bbest(v)\}, otherwise; \end{cases}$$

// main drive
MaxV(T) = max{Bbest(T), Gbest(T)};

b)
After we compute the Bbest and Gbest value of each vertex for the first time, we store it in the Data[1...n]array so that our next reference could just use that corresponding value and just do the comparison.

c)
Read(n);
Create the empty array of child lists children[1...n];
Let 1 be the root of the tree;
Creat the global array Data[1...n] where each Data[i] has the three fields .dat, .Bbest, and .Gbest;
for i <- 1 to n do Read Data[i].dat endfor;
for i <- 1 to n do Data[i].Bbest <- Nil endfor;

```
for i <- 1 to n do Data[i].Gbest <- Nil endfor;
print(max{Bbest(1), Gbest(1)});

function Bbest(T);
   if T is a leaf then return T.dat;
   else
      if Data[T].Bbest == Nil then
         sum <- 0;
         for v in Child[T] do
            Gtemp <- Data[v].Gbest;
            if Gtemp <- Nil then {We have not calculated Gbest(v) yet}
               Gtemp <- Gbest(v);
               sum <- sum + Gtemp;
               Data[v].Gbest <- Gtemp; {store the answer}
            endif;
         endfor;
      endif;
      Data[T].Bbest <- sum;
      return sum;
   endif;
end_Bbest;

function Gbest(T);
   if T is a leaf then return 0;
   else
      if Data[T].Gbest == Nil then
         sum <- 0;
         for v in Child[T] do
            Gtemp <- Data[v].Gbest;
            Btemp <- Data[v].Bbest;
            if Gtemp <- Nil then {We have not calculated Gbest(v) yet}
               Gtemp <- Gbest(v);
               Data[v].Gbest <- Gtemp; {store the answer}
            endif;
            if Btemp <- Nil then {We have not calculated Bbest(v) yet}
               Btemp <- Bbest(v);
               Data[v].Bbest <- Btemp; {store the answer}
            endif;
            if Gtemp ≥ Btemp then sum <- sum + Gtemp;
            else sum <- sum + Btemp endif;
         endfor;
      endif;
      Data[T].Gbest <- sum;
      return sum;
    endif;
end_Gbest;
```

5.Graph

We can use a simplified version of bucket sort and the Paige-Tarjan lexicographical sort algorithm to sort edges in the Adj[n] array.

First, use the PT algorithm to store all parent-child pairs in the parents Adjacency array.

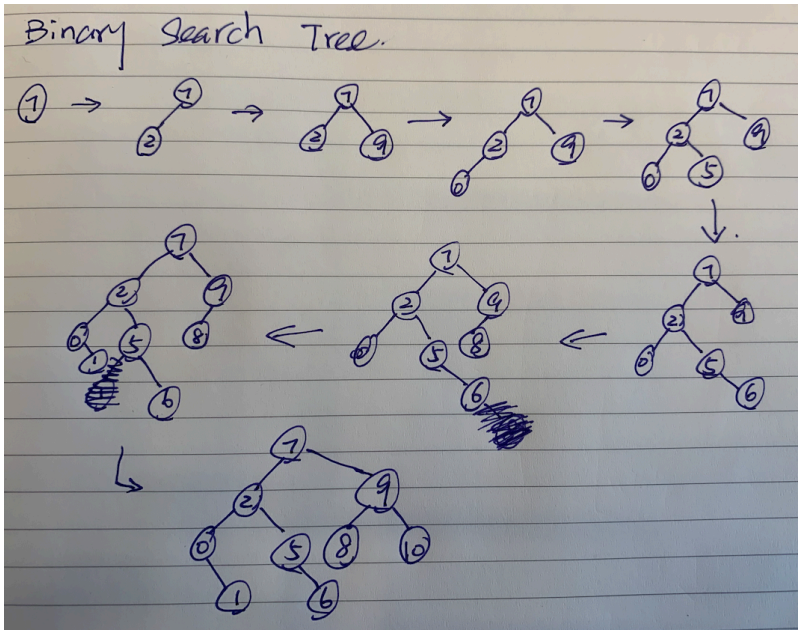
The second pass uses bucket sort, and places (parent, child) pairs into the child bucket.

Lastly, put each pair back to the parents' Adjacency array in order of the buckets. Since the buckets are sorted, the pairs will be sorted.

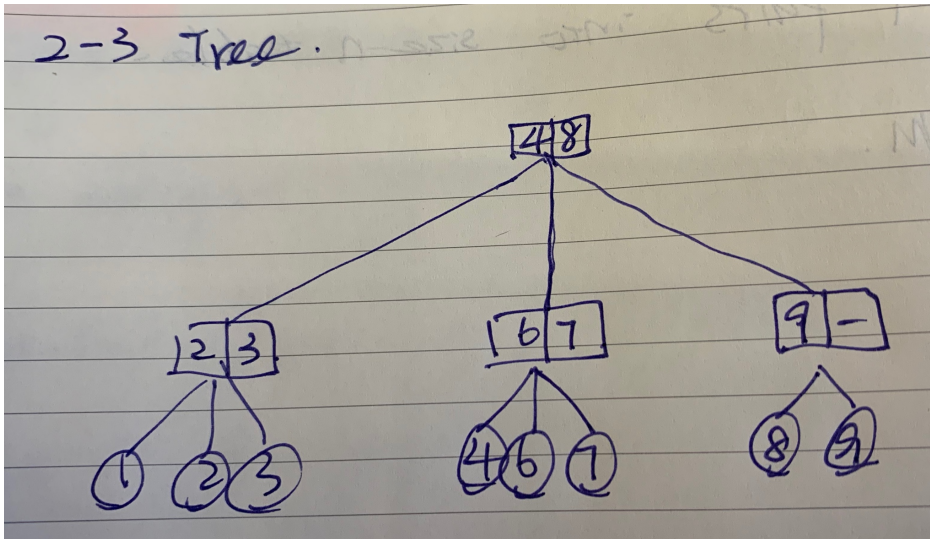
5.40 a) Analyzing Mergesort

$T(n) = l * n + 2 * r + 2^l$

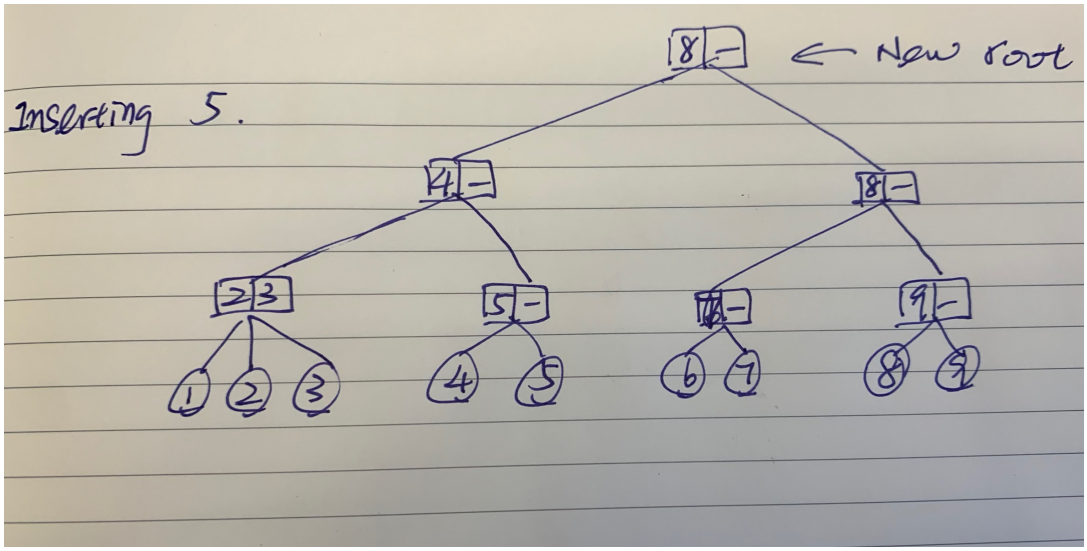
6.5



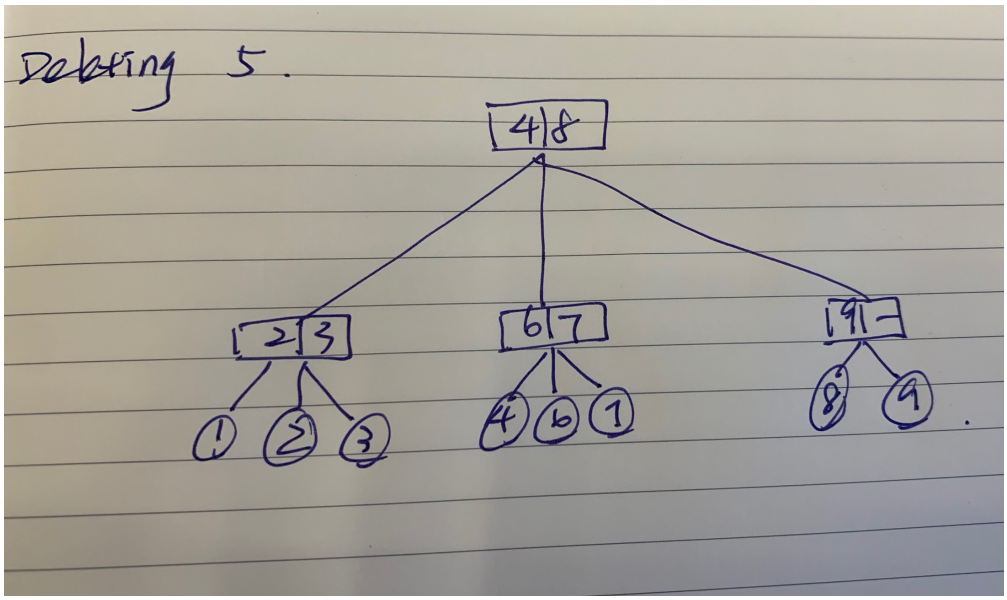
6insertdelete.a



6insertdelete.b



6insertdelete.c



6.10 2-3trees

a)
On the path from the root of the 2-3 tree down to the vertex that would become the parent of the leaf-records x (if no splitting were to occur), each vertex would have exactly 3 child vertices. In this way, when the leaf is added, it would push out its siblings and force each parent level to split up, causing the root to be divided into two, and creating a new root.

6.range

a. Insert(x): insert the key x and increment the count value stored in all vertices on the path from the root to this new vertex by 1.

b. Locate(x): will just compare the value x with the left and right guilds at each vertex along the way. Since the left guide stores the smallest node in the center partition for 3-child structures and in the right partition for 2-child structures, Locate(x) will easily find which partition it should enter, and save time from testing in other partitions.

c. Delete(x): as Delete(x) traverses from root to the vertex, decrement the count value by 1 from all vertices along the way.

d. CountLessThan(x): would simply add the count values in the path, subtracting its right sibling counts at the leaf level.

6.intervals

Under an interval 2-3 tree structure, each interval (a,b) will be stored as pairs. A different pair (r,s) will be considered smaller than (a,b) if either $r < a$ or $r = a$ and $s < b$.

This covers the locate function. As for add and delete, it is similar to the regular 2-3 tree, where each pair is treated as a vertex.

6.intervalstabs

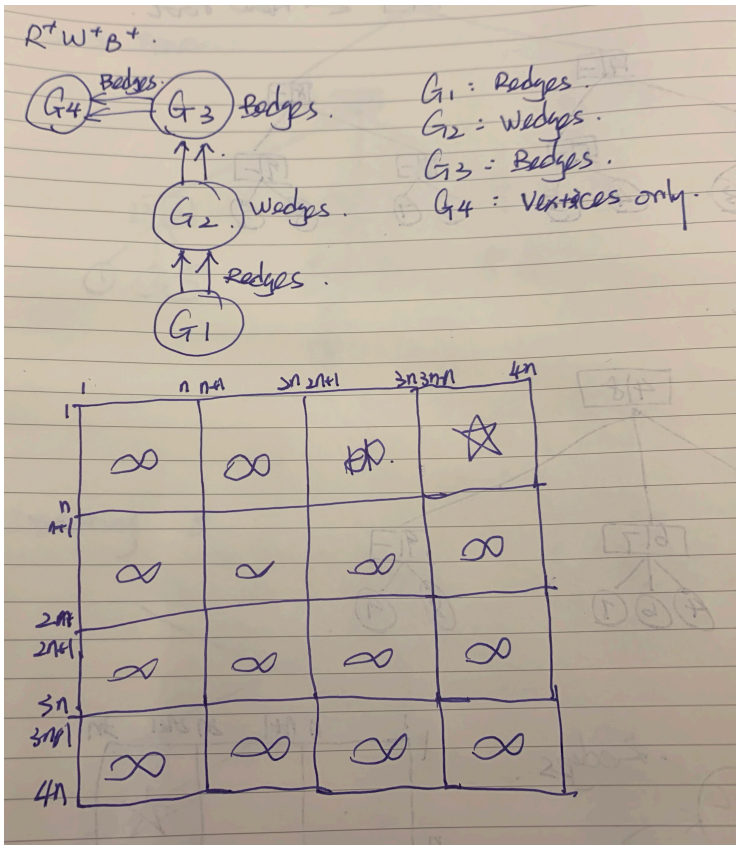
To calculate stab(x), we need to consider two factors. As the hint suggests, the number of intervals that are "alive" are:

1. born before or at x ($a \leq x$)
2. did not die before x ($b > x$)

As such, the precise number is the number of intervals born before or at x, minus those that died before x. Therefore, it is the vertices in the interval 2-3 tree that are to the left of the would-be (a, x) vertex.

8.qqq

a)



We apply the FW algorithm to the supergraph and focus on the upper right quadrant. This entails that all paths have to start from G1, undergo some number of redges, go into G2 from one of the redges (fulfills the R+ requirement), undergo some number of wedges, go into G3 using one of the wedges (W+), undergo some bedges, go into G4 using one bedge (B+), and finally end at G4.

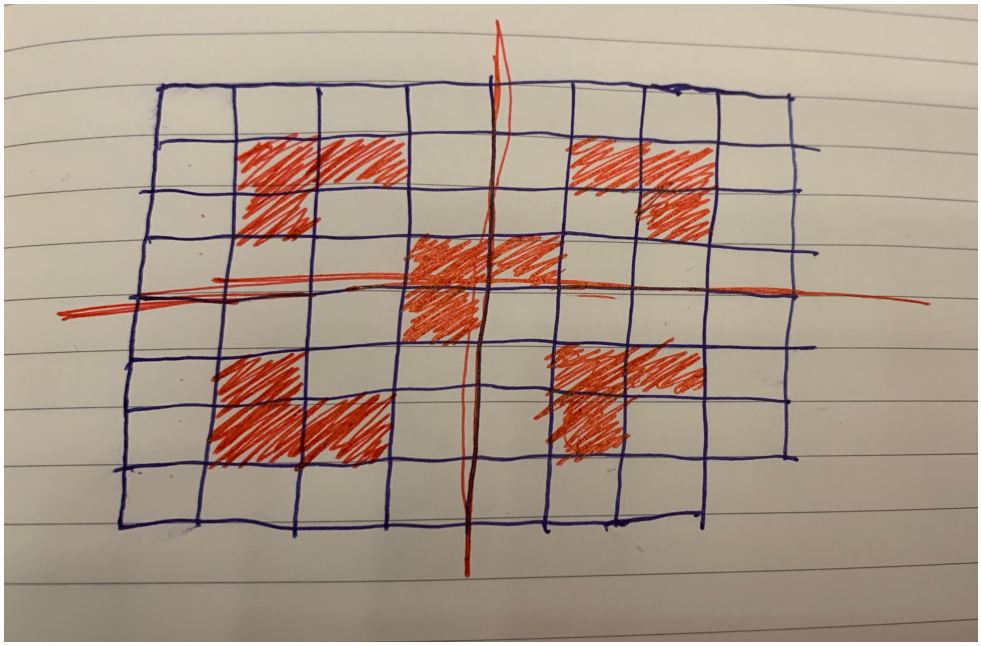
b)
$(4n)^3 + O(4n)^2$

c)
Solve the subproblems of R+W+ first using the 2-step algorithm, and then apply that again to solve the (R+W+)B+ problem.

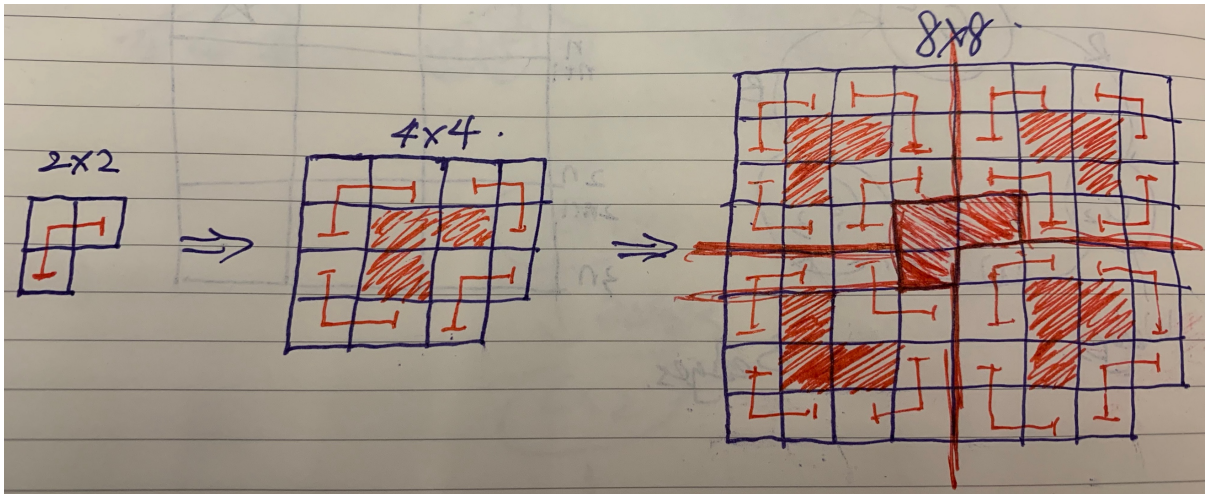
d)
$2n^3 + O(n^2)$

11.14 Divide and conquer

a)
The "elegant" solution looks at the L-shaped pieces as solutions. At each intersection between 4 pieces, each of the three squares in an L-shaped piece falls into a different quadrant. As such, this piece can be seen as the "missing" piece, which results in three 2*2 mini boards with a missing square:



b)
Since the existing piece in a 2*2 board is also L-shaped, we can break larger boards down to this basic unit, and apply recursion:



c)
For the elegant solution, we have seen that where the square is missing does not matter, since a 2*2 basic board can rotate, and the L piece can always fit the other 3 2*2 mini boards as their own "missing" squares.

For the clunky solution, the basic unit has the only missing piece, and we can put that basic unit where ever we want in the general board.

8.onemore

a)
(r)(w)b

b)
(r)(w)(b)

c)
(r)w(b)

8.last

a)
r(w)(b)

b)
(r)(w)(r)(b)

c)
(r+)(w)b