Advanced Algorithms 3rd Assignment

Siddhesh Shirsat Viviktesh Agwan Ajinkya Potdar

Task 1:

a)

```
Iter-MM-ijk( Z; X; Y; n )
for i   1 to n do
        for j   1 to n do
                for k   1 to n do
                        Z[i; j ]   Z[i; j ] + X[i; k] Y [k; j]
```

The elements of Z are accessed in row major order so the number of I/O's to access all the elements of Z will be $O(n(1 + n/B))$.

(i.e. $O(1 + n/B)$ to access the contiguous elements in each row. And there are n such rows.)

The elements of X will be accessed in row major order and each row will be accessed n times once it is brought in the cache. So the number of I/O's to access all the elements of X will be $O(n(1 + n/B))$.

The elements of Y will be accessed in column major order and the complete matrix will be accessed n times. So the number of I/O's to access all the elements of Y will be $O(n(n^2)) = O(n^3)$.

Therefore, cache complexity of this version: $O(n(1 + n/B)) + O(n^3) = O(n^3)$

```
Iter-MM-ikj( Z; X; Y; n )
for i   1 to n do
        for k   1 to n do
                for j   1 to n do
                        Z[i; j ]   Z[i; j ] + X[i; k] Y [k; j]
```

The elements of Z will be accessed in row major order and each row will be accessed n times once it is brought in the cache. So the number of I/O's to access all the elements of Z will be $O(n(1 + n/B))$.

The elements of X will be accessed in row major order. So the number of I/O's to access all the elements of X will be $O(n(1 + n/B))$.

The elements of Y will be accessed in row major order and the complete matrix will be accessed n times. So the number of I/O's to access all the elements of Y will be $O(n^2(1 + n/B))$.

Therefore, cache complexity of this version: $O(n(1 + n/B)) + O(n^2(1 + n/B)) = O(n^2 + n^3/B))$

```
Iter-MM-jik( Z; X; Y; n )-

for j   1 to n do

        for i   1 to n do

                for k   1 to n do

                        Z[i; j ]   Z[i; j ] + X[i; k] Y [k; j]
```

The elements of Z will be accessed in column major order. So the number of I/O's to access all the elements of Z will be $O(n^2)$.

The elements of X will be accessed in row major order and the complete matrix will be accessed n times. So the number of I/O's to access all the elements of X will be $O(n^2(1 + n/B))$.

The elements of Y will be accessed in column major order and each element of a given column will be accessed n times for a given value of j. So the number of I/O's to access all the elements of Y will be $O(n^3)$.

Therefore, cache complexity of this version: $O(n^2) + O(n^2(1 + n/B)) + O(n^3)= O(n^3)$

```
Iter-MM-jki( Z; X; Y; n )

for j   1 to n do

        for k   1 to n do

                for i   1 to n do

                        Z[i; j ]   Z[i; j ] + X[i; k] Y [k; j]
```

The elements of Z will be accessed in column major order and each element of a given column will be accessed n times for a given value of j. So the number of I/O's to access all the elements of Z will be $O(n^3)$.

The elements of X will be accessed in column major order and the complete matrix will be accessed n times. So the number of I/O's to access all the elements of X will be $O(n^3)$.

The elements of Y will be accessed in column major order. So the number of I/O's to access all the elements of Y will be $O(n^2)$.

Therefore, cache complexity of this version: $O(n^2) + O(n^3) + O(n^3)= O(n^3)$

```
Iter-MM-kij( Z; X; Y; n )

for k   1 to n do

        for i   1 to n do

                for j   1 to n do

                        Z[i; j ]   Z[i; j ] + X[i; k] Y [k; j]
```

The elements of Z will be accessed in row major order and the matrix will be accessed n times. So the number of I/O’s to access all the elements of Z will be $O(n^2(1 + n/B))$.

The elements of X will be accessed in column major order. So the number of I/O’s to access all the elements of X will be $O(n^2)$.

The elements of Y will be accessed in row major order and each row will be accessed n times once it is brought in the cache. So the number of I/O’s to access all the elements of Y will be $O(n(1 + n/B))$.

Therefore, cache complexity of this version: $O(n^2(1 + n/B)) + O(n^2) + O(n(1 + n/B)) = O(n^2)$

```
Iter-MM-kji( Z; X; Y; n )

for k   1 to n do

        for j   1 to n do

                for i   1 to n do

                        Z[i; j ]   Z[i; j ] + X[i; k] Y [k; j]
```

The elements of Z will be accessed in column major order and the matrix will be accessed n times. So the number of I/O’s to access all the elements of Z will be $O(n^3)$.

The elements of X will be accessed in column major order and each element of a given column will be accessed n times for a given value of k. So the number of I/O’s to access all the elements of X will be $O(n^3)$.

The elements of Y will be accessed in row major. So the number of I/O’s to access all the elements of Y will be $O(n(1 + n/B))$.

Therefore, cache complexity of this version: $O(n^3) + O(n(1 + n/B)) = O(n^3)$

c)

R2Z(X(n by n), X_Z[1….n^2], n) n = power of 2

if(n <= 1) X_Z = X

else

R2Z(X11, X_Z[1…$n^2/4$], n/2)

R2Z(X12, X_Z[$n^2/4+1…n^2/2$], n/2)

R2Z(X21, X_Z[$n^2/2+1…3n^2/4$], n/2)

R2Z(X22, X_Z[$3n^2/4+1…n^2$], n/2)

Running time complexity:

T(n) = O(1) if n<=1

= 4*T(n/2) + O(1) otherwise

Therefore, T(n) = $\Theta(n^2)$ Master Theorem case 1.

Cache Complexity:

For $n^2 > M$, (M = size of the memory)

Q(n) = $O(n + n^2/B)$ if $n^2 <= \alpha M$

= 4Q(n/2) + O(1) otherwise

Therefore, Q(n) = $O(n^2/M + n^2/(B*sqrt(M))) = O(n^2/(B*sqrt(M)))$ when $M = \Omega(B^2)$ (i.e. tall cache)

For $n^2 <= M$, Q(n) = $O(1 + n^2/B)$

Therefore, for all n, Q(n) = $O(n^2/(B*sqrt(M)) + n^2/B + 1)$

Z2R(X_Z[1….n^2], X(n by n), n) n = power of 2

if(n <= 1) X = X_Z

else

Z2R(X_Z[1…$n^2/4$], X11, n/2)

Z2R(X_Z[n^2/4+1…n^2/2], X12, n/2)

Z2R(X_Z[n^2/2+1…3n^2/4], X21, n/2)

Z2R(X_Z[3n^2/4+1…n^2], X22, n/2)

Running time complexity:

$T(n) = O(1)$ if $n<=1$

$= 4*T(n/2) + O(1)$ otherwise

Therefore, $T(n) = \Theta(n^2)$ Master Theorem case 1.

Cache Complexity:

For $n^2 > M$, (M = size of the memory)

$Q(n) = O(n + n^2/B)$ if $n^2 <= \alpha M$

$= 4Q(n/2) + O(1)$ otherwise

Therefore, $Q(n) = O(n^2/M + n^2/(B*sqrt(M))) = O(n^2/(B*sqrt(M)))$ when $M = \Omega(B^2)$ (i.e. tall cache)

For $n^2 <= M$, $Q(n) = O(1 + n^2/B)$

Therefore, for all n, $Q(n) = O(n^2/(B*sqrt(M)) + n^2/B + 1)$

e)

We see L1 and L2 cache misses are less in Iterative MM version 2 and version 5 as compared to other versions. This is in accordance with the cache complexities determined in (a).

Also, the cache misses of RecMM are higher than that of Z-Mortan multiplication i.e. RecMM2.

Please find the results in the directory ‘tables’.

Q2.A

Main idea is to use dynamic programming and cache the values in the table to compute new values.

Create and Update table score[i,j]

```
Calculate_score()
{
        for i  ←  1 to n-1
        {
                score [i,n]= 0;
        }
        for j  ← n-1 to 2 do
        {
           for i  ← 1 to j-1 do
              {
                max  = -1;
               for k ← j+2 to n
                  {
                    cur_score ← score_one_fold(i,j,k)+score[j+1,k];
                    if cur_score > max then
                        max= cur_score;
                   score[i, j]= max;
                  }
              }
        }
        max = 0;
        for i ← 2 to n
        {
            if score[1,i ]>max then max = score [1, i];

        }

        return max;

}
```

The 3 for loops account for O(n^3) , score_one_fold is O(n) , Hence total time complexity -> O (n^4)
We Store the score(i,j) table , which takes up the O(n^2) space

Q2B.

We can precompute and store the values of score_one_fold(i,j,k) in a 3-Dimensional table before entering in the nested loops of i,j,k of previous question.
Also, instead of making a call to score_one_fold (i,j,k) each time we use some caching as explained below.
We make score_one_fold (i,j,k) call only once for each of the j loops, for the k loop we calculate value of score_one_fold (i,j,k) from score_one_fold (i,j,k-1)
i.e We uses following constant time update logic

score_one_fol(i,j,k)=

$$\begin{cases} \text{score\_one\_fold}(i, j, k-1) + 1 & \text{if } HP(P[j - k]) \text{ and } HP(P[j + 1 + k]) \\ \text{score\_one\_fold}(i, j, k-1) & \text{if } (k - j - 1) > (i - j \qquad) \end{cases}$$

Thus using $O(n^3)$ storage for score_one_fold , we removed score_one_fold function complexity which was $O(n)$, thus reducing the overall time complexity to $O(n^3)$.

Pseudo code for precomputation of score_one_fold

```
for j  ← n-1 to 2 do
{
        for i  ← 1 to j-1 do
        {
                max = -1;
                score_one_f[i][j][j+2] = score_one_fold(i,j,k)
                for k ← j+2 to n
                {
                        update_score_one_fold(I,j,k); // using update logic as mentioned above
                }
        }
}
```

Cache complexity:

Assuming same Compute_score() function in previous question:
Complexity of inner loop: $O(1+n/B)$
So overall cache misses = $O(n^2(1+n/B))$

Q2.c We can modify precomputation storage taken by score_one_fold from O (n^3) to O(n) as shown below. And thus total space complexity reduces to O(n^2) . This space complexity comes because of score matrix which is 2 dimensional.
We see that that value of score_one_fold(i,j,k) is used only in k loop thus , that is to calculate score (x,y) – we need just "k" values of score_one_fold generated in the k loop, there is no need to store O (n^3) values
Thus we can modify the score_one_fold(i,j,k) table into just a single array score_one_fold(k)
The first value of this array will be calculated once for each combination of (j,i) –
For every k – we can generate the required value using values already stored in constant time as follows

```
for i ← 1 to n-1
{
        score [i,n]= 0;
}
for j ← n-1 to 2 do
{
   for i ← 1 to j-1 do
   {
        score_one_f[j+2] = score_one_fold(i,j,k)
        for k ← j+2 to n
        {
                // Compute_score_one_f(k) using logic given below in Equation 1
        }

        max = -1;
        for k ← j+2 to n
        {
              Cur_score ← score_one_f [k] +score[j+1,k];
              if cur_score >max
                 then max = cur_score;
             score[i, j]= max;
           }
    }
}
max = 0;
for i ← 2 to n
{
    if score [1,i]>max
          then max = score [1, i];
}

return max;

}
```
where:

score_one_f(k)= $\begin{cases} \text{score\_f}(k-1)+1 & \text{if } HP(P[j-k]) \text{ and } HP(P[j+1+k] \\ \text{score\_f}(k-1) & \text{if } (k-j-1) > (i-j \qquad) \end{cases}$ ... Equation 1

Cache Complexity:

Assuming same Compute_score() function in previous question:

Cache complexity of score_one_f:

Within the i-for loop, the elements of score_one_f are accessed with O(1+n/B) I/O's. In the wto k-for loops, the already cached elements will be accessed. Hence no additional I/O's. The process repeats for O(n) values of j.

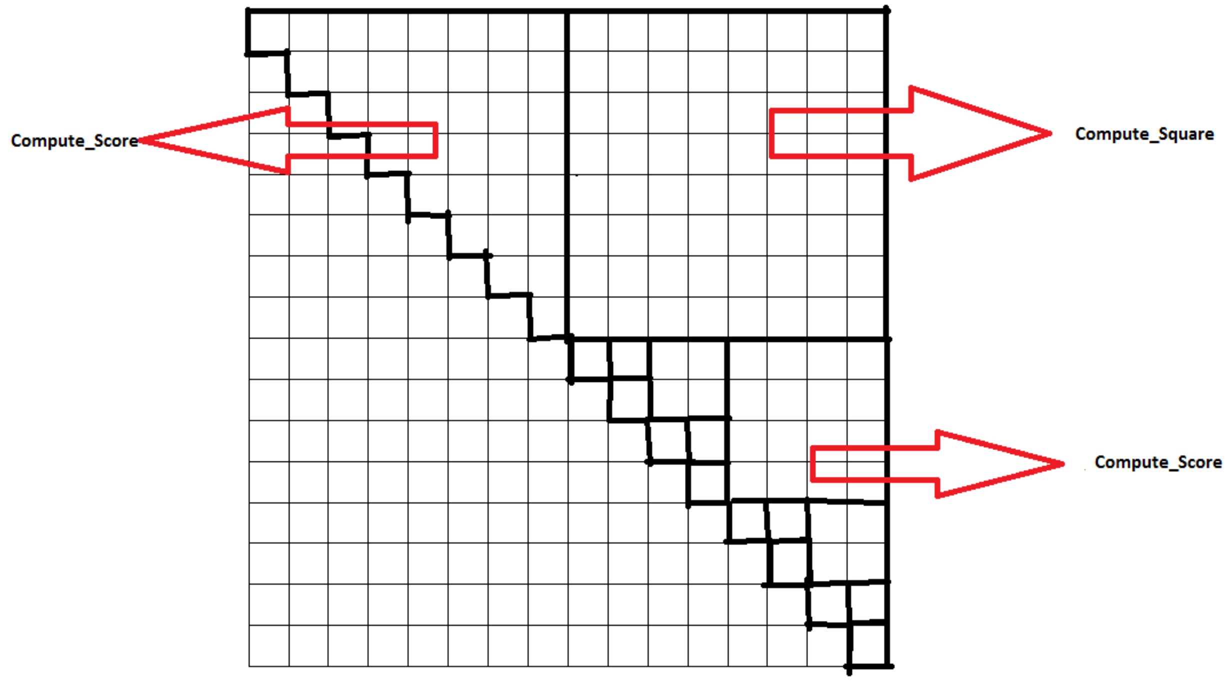
Therefore, overall cache complexity for score_one_f = n*O(1+n/B) = $O(n+n^2/B)$

Cache complexity of score:

The elements of score are accessed in column major order, so its cache complexity is $O(n^2)$.

So overall I/O's = $O(n(1+n/B)) + O(n^2) = O(n^2)$

Q2.D



We compute the score using divide and conquer recursive approach as shown in the figure.

The above matrix is Score (i,j) , for calculating score (i,j) – We divide the computation into 3 parts as shown in the figure.

The bottom compute_score is done first as it can be independently calculated , the compute square whose values depend upon the bottom compute_score - Finally the upper compute score is calculated .

```
Compute_score (i,j,N, S)
{         if( i > j)
          {
                    S[i][j] =0
          }
          Else if ( n== 1)
          {
                    Max_val = 0;
                    S_o_fold[1 - max];
                    S_o_fold[j+2] = Score_one_fold(i,j,j+2);  //calculating for  k= j+2
```

```
            for k=j+2 to max
            {
                        S_o_fold[k]=Calc_sof(); // calc_sof Calculates  the correct value of s_o_fold
in constant time  as  explained in Q2.c

                        Current = s_o_fold[k] + S[j+1][k]
                        If(current > Max_val)
                        {
                                    Max_val = Current;
                        }
            }
            S[i][j] = Max_val
      }
      else {
      Compute_score(i+n/2, j+n/2, n/2,S)
      Compute_square(i, j+n/2, n/2,S);
      Compute_score(i,j, n/2,S);
}

Compute_square(i, j, n, S) {

      if ( n== 1)
      {
            Max_val = 0;
            S_o_fold[1 - max];
            S_o_fold[j+2] = Score_one_fold(i,j,j+2);  //calculating for  k= j+2
            for k=j+2 to max
            {
                        S_o_fold[k]=Calc_sof(); // calc_sof Calculates  the correct value of s_o_fold
in constant time  as  explained in Q2.c

                        Current = s_o_fold[k] + S[j+1][k]
                        If(current > Max_val)
                        {
                                    Max_val = Current;
                        }
            }
            S[i][j] = Max_val;
      }

      Compute_square (i+n/2, j+n/2, n/2,S) // bottom right
      Compute_square (i+n/2, j+n/2, n/2,S) // bottom left
      Compute_square (i,j+n/2, n/2,S)  // top right
      Compute_square (i,j, n/2,S)  // top left
}
```

Time Complexity

$T_{overall}(n) = 2^* T_{overall}(n/2) + T_{square}(n/2)$ - 1

$T_{square}(n) = O(n)$ if n ==1 // We need to calculate max_val calculation

= $4T(n/2)$ otherwise

Thus $T_{square}(n) = 4^{\log_2 n} * T_{square}(1) = O(n^2)*O(n) = O(n^3)$

Substituting in Eq 1 – $T_{overall}(n) = 2^* T_{overall}(n/2) + O(n^3) = O(n^3)$ - By masters Theorem case 3

Space complexity is $O(n^2)$ as we are storing all the values of S – which is 2 D Array

Cache Complexity : -

$Q_{overall}(n) = 2(Q_{overall}(n/2)) + Q_{square}(n/2)$

$Q_{square}(n) = O(n/B * n(1+n/B))$ -if($n^2 < \alpha M$)

$= 4 Q_{square}(n/2) + O(1)$ otherwise

So $Q_{square}(n) = 4^k Q_{square}(n/2^k) + O(1)$

Solving for k

$K = \log_2 n/\sqrt{\alpha M}$

So $Q_{square}(n) = n^2/m * (n/B * n(1+n/B))$

$= n^3/MB * (\sqrt{M} + M/B)$ As $n^2 = \alpha M$ for base case

$= O(n^3/B\sqrt{M})$

As $Q_{square}(n)$ dominates over $2 Q_{overall}(n/2)$

Thus $Q_{overall}(n) = Q_{square}(n) = O(n^3/B\sqrt{M})$

The algorithm in (2B) cannot be converted to divide and conquer as it requires $O(n^3)$ storage whereas we have the restriction of $O(n^2)$ space which is achieved by modifying (2C).