CSC 401 - Advanced Algorithms Assignment

Arnav Kumar

October 23, 2013

1 Question 1

1.1 $A(n) = 3A(n/4) + n^{(3/4)}$

$A(n) = \Theta(n^{log_3 4})$

Proof. Inspection using Iteration method

$$\begin{aligned}
A(n) &= 3A(\frac{n}{4}) + n^{(3/4)} \\
&= 9A(\frac{n}{16}) + 3(\frac{n}{4})^{3/4} + n^{3/4} \\
&\vdots \\
&= 3^k A(\frac{n}{4^k}) + 3^{k-1}(\frac{n}{4^{k-1}})^{3/4} + \cdots + 3(\frac{n}{4})^{3/4} + n^{3/4} \\
&= 3^k A(1) + \sum_{i=0}^{k-1} 3^{k-1}(\frac{n}{4^{k-1}})^{3/4}
\end{aligned}$$

Given that $A(1)$ is a constant

$$= 3^k C_1 + n^{3/4} \sum_{i=0}^{k-1} \frac{3^i}{4^{3i/4}}$$

$\sum_{i=0}^{k-1} 3^i/(4^{3i/4})$ is a numerical constant, taking it as C_2

$$\begin{aligned}
&= (4^k)^{log_3 4} C_1 + n^{3/4} C_2 \\
&= n^{log_3 4} C_1 + n^{3/4} C_2
\end{aligned}$$

$n^{\log_3 4} = n^{0.79}$, $n^{3/4} = n^{0.75}$

Therefore $n^{\log_3 4}$ is the dominating term and,

$$A(n) = \Theta(n^{log_3 4})$$

Verification using Master Theorem,

$$\begin{aligned}A(n) &= 3A(\frac{n}{4}) + n^{(3/4)}\\ a &= 3,\\ b &= 4, \text{and},\\ f(n) &= n^{3/4}\end{aligned}$$

Is f(n) $= \mathcal{O}(n^{\log_b a - \epsilon})$?

$$\log_b a = \log_3 4 = 0.79$$

Because, $n^{\log_3 4} = n^{0.79}$, therefore,

$$\begin{aligned}f(n) &= n^{3/4} = n^{0.75}\\ &= \mathcal{O}(n^{\log_3 4 - \epsilon})\\ \text{where, } \epsilon &= 0.04\end{aligned}$$

Therefore, by Case 1 of the Master Theorem,

$$A(n) = \Theta(n^{log_3 4})$$

□

1.2 $B(n) = B(n-2) + n \lg n$

$B(n) = \mathcal{O}(n^2 \lg n)$

Proof.

$$\begin{aligned}B(n) &= B(n-2) + n \lg n\\ &= B(n-4) + (n-2)\lg(n-2) + n \lg n\\ &\vdots\\ &= B(n-2k) + (n-2k+2)\lg(n-2k+2) + \cdots + (n-2)\lg(n-2) + n \lg n\end{aligned}$$

Choosing k such that $n - 2k = 1$ or $n - 2k = 0$

Therefore, $B(n-2k) = constant (= c$, say)

$$B(n) = c + (n-2k+2)\lg(n-2k+2) + \cdots + (n-2)\lg(n-2) + n \lg n$$

$f(n) = n \lg n$, being the product of two increasing functions, $f_1(n) = n$ and $f_2 = \lg n$, is a strictly increasing function

We can therefore say that

$$\begin{aligned}
(n-2)\lg(n-2) &< (n-2)\lg n,\\
(n-4)\lg(n-4) &< (n-4)\lg n,\\
&\vdots\\
(n-2k+2)\lg(n-2k+2) &< (n-2k+2)\lg n
\end{aligned}$$

Therefore,

$$\begin{aligned}
B(n) &= c + (n-2k+2)\lg(n-2k+2) + \cdots + (n-2)\lg(n-2) + n\lg n\\
&< c + (n-2k+2)\lg n + \cdots + (n-2)\lg n + n\lg n\\
&< c + \lg n((n-2k+2) + \cdots + (n-4) + (n-2) + n)\\
&< c + \lg n(\frac{n(n-1)}{4})\\
&< c + (\frac{n^2}{4})\lg n - (\frac{n}{4})\lg n
\end{aligned}$$

Therefore,

$$B(n) = \mathcal{O}(n^2\lg n)$$

□

1.3 $C(n) = C(\lceil n/3\rceil) + \lceil n/2\rceil$

$C(n) = \Theta(n)$

Proof. Transforming using Akra-Bazzi Theorem,

$$\begin{aligned}
C(n) &= C(\lceil n/3\rceil) + \lceil n/2\rceil\\
C(n) &= \lceil n/2\rceil + C(\lceil n/3\rceil) && \ldots(1)\\
C(n) &= \lceil n/2\rceil + C(n/3 + h(n)) \text{ where } h(n) = \lceil n/3\rceil - n/3
\end{aligned}$$

Since $0 < \lceil n/3\rceil - n/3 < 1$

$$h(n) = \mathcal{O}(1)$$

By Akra-Bazzi Theorem, the above at (1) has the same order as,

$$\begin{aligned}
C'(n) &= \lceil n/2\rceil + C'(n/3)\\
&= n/2 + (\lceil n/2\rceil - n/2) + C'(n/3)
\end{aligned}$$

Let's define a function $f(x) = \lceil n \rceil - n$,

$$\begin{aligned}
C'(n) &= n/2 + f(n/2) + C'(n/3) \\
&= n/2 + n/6 + f(n/2) + f(n/6) + C(n/9) \\
&\vdots \\
&= \frac{n}{2}(1 + \frac{1}{3} + \frac{1}{9} + \cdots + \frac{1}{3^k}) \\
&+ f(\frac{n}{2}) + f(\frac{n}{6}) + \cdots + f(\frac{n}{2 * 3^k}) \\
&+ C'(n/3^k) \ldots \text{where k is such that } \frac{n}{3^k} \text{ is a small fraction}
\end{aligned}$$

Now let,

$$\begin{aligned}
a &= 1 + \frac{1}{3} + \frac{1}{9} + \cdots + \frac{1}{3^k} \text{ and,} \\
b &= f(\frac{n}{2}) + f(\frac{n}{6}) + \cdots + f(\frac{n}{2 * 3^k})
\end{aligned}$$

Also, we know that for small constants,

$C'(n)$ will have constant(say, $= X_3$), O(1) values,

Therefore,

$$C'(n) = \frac{n}{2}a + b + X_3$$

Examining the term a,

We know from the theory of geometric progressions that,

$$\begin{aligned}
\sum_{i=0}^{n} cr^i &= \frac{c(1 - r^{n+1})}{1 - r} \\
a &= 1 + \frac{1}{3} + \frac{1}{9} + \cdots + \frac{1}{3^k}
\end{aligned}$$

Therefore, for us, $c = 1$, $r = \frac{1}{3}$ and $n = k - 1$

$$\begin{aligned}
a &= 1 * (\frac{1 - \frac{1}{3}^k}{1 - \frac{1}{3}}) \\
&= \text{a numerical constant } (= X_1, \text{ say})
\end{aligned}$$

Now, examining the term b,

Since $0 \leq f(x) < 1$ for any x,

$$\begin{aligned} b &= f(\frac{n}{2}) + f(\frac{n}{6}) + \cdots + f(\frac{n}{2 * 3^k}) \\ max(b) &< 1 + 1 + \cdots + 1 \\ &= \log_3 n \\ &= \frac{\lg n}{\lg 3} \text{ and,} \\ min(b) &= 0 + 0 + \cdots + 0 \\ &= 0 \end{aligned}$$

Therefore,

$$\begin{aligned} C'(n) &= \frac{n}{2}X_1 + b + X_3 \\ &< \frac{n}{2}X_1 + \frac{\lg n}{\lg 3} + X_3 \end{aligned}$$

Hence,

$$C'(n) = \mathcal{O}(n) \qquad \ldots(2)$$

Also,

$$\begin{aligned} C'(n) &= \frac{n}{2}X_1 + b + X_3 \\ &\geq \frac{n}{2}X_1 + 0 + X_3 \end{aligned}$$

Hence,

$$C'(n) = \Omega(n) \qquad \ldots(3)$$

Thus, from (2) and (3) we can conclude,

$$C'(n) = \Theta(n)$$

□

2 Question 2

2.1 Farthermost pair of vertices of a convex polygon

We shall use a slight modification of the rotating-callipers algorithm to find our answer. We will iteratively find all the anti-podal points of the polygon. A pair of anti-podal points is defined a pair of points through which two parallel tangents can be drawn to the polygon which does not intersect any other edge of the polygon. We can generate all the antipodal points by first determining one pair and then rotating the tangents minimally along the sides of the polygon till they pass another set of antipodal points. In our algorithm, we can find our first anti-podal pair as the pair of points, first of which has the minimum y co-ordinate and the other the maximum y co-ordinate amongst the set of points. Let's assume two methods $angle(line_a, line_b)$ that returns the angle between lines $line_a$ and $line_b$ and $distance(point_a, point_b)$ that returns the distance between points $point_a$ and $point_b$. The anti-podal pair with the maximum distance between them is our required pair. If there are many such pairs, we return the first one to be found.
Here is the pseudo-code:

```
$n \leftarrow$number of vertices
$p_1 \leftarrow$ point with minimum y-co-ordinate
$p_2 \dots p_n \leftarrow$ points sorted CCW according to polar angle from $p_1 \dots$ (1)
$p_a \leftarrow p_1$
$p_b \leftarrow$point with minimum y-co-ordinate
$angleRotated \leftarrow 0$
$maxDistance \leftarrow distance(p_a, p_b)$
$furthermostPair \leftarrow (p_a, p_b)$
$supportingLine_a \leftarrow$ horizontal vector passing thro' $p_a$ towards +ve X axis
$supportingLine_b \leftarrow$ horizontal vector passing thro' $p_b$ towards -ve X axis
**while** $angleRotated < \pi$ **do**
    $edge_a \leftarrow edge(p_a, p_{a+1})$
    $edge_b \leftarrow edge(p_b, p_{b+1})$
    $angle_a \leftarrow angle(supportingLine_a, edge_a)$
    $angle_b \leftarrow angle(supportingLine_b, edge_b)$
    **if** $angle_a < angle_b$ **then**
        **if** $distance(p_{a+1}, p_b) > maxDistance$ **then**
            $maxDistance \leftarrow distance(p_{a+1}, p_b)$
            $furthermostPair \leftarrow (p_{a+1}, p_b)$
        **end if**
        $p_a \leftarrow p_{a+1}$ (This index wraps around. $p_{n+1} = p_1$)
    **else if** $angle_b < angle_a$ **then**
        **if** $distance(p_a, p_{b+1}) > maxDistance$ **then**
            $maxDistance \leftarrow distance(p_a, p_{b+1})$
            $furthermostPair \leftarrow (p_a, p_{b+1})$
        **end if**
```

```
            $p_b \leftarrow p_{b+1}$ (This index also wraps around. $p_{n+1} = p_1$)
        **else** (i.e. $angle_a = angle_b$)
            **if** $distance(p_{a+1}, p_b) > maxDistance$ **then**
                $maxDistance \leftarrow distance(p_{a+1}, p_b)$
                $furthermostPair \leftarrow (p_{a+1}, p_b)$
            **else if** $distance(p_a, p_{b+1}) > maxDistance$ **then**
                $maxDistance \leftarrow distance(p_a, p_{b+1})$
                $furthermostPair \leftarrow (p_a, p_{b+1})$
            **else if** $distance(p_{a+1}, p_{b+1}) > maxDistance$ **then**
                $maxDistance \leftarrow distance(p_{a+1}, p_{b+1})$
                $furthermostPair \leftarrow (p_{a+1}, p_{b+1})$
            **end if**
            $p_a \leftarrow p_{a+1}$
            $p_b \leftarrow p_{b+1}$ (This indices wrap around. $p_{n+1} = p_1$)
        **end if**
        $angleRotated \leftarrow angleRotated + min(angle_a, angle_b)$
    **end while**
    **return** $furthermostPair$
```

This algorithm works in $\mathcal{O}(n \lg n) time$. Let's analyze why. There are two main tasks in this algorithm:

- Sorting vertices in CCW order from p_1
- Iterating through the vertices optimally to determine the pair furthest apart

We know that sorting vertices in CCW order from p_1 takes $\mathcal{O}(n \lg n)$ time.
Inspecting the above algorithm, we see that we by since *angleRotated* starts from 0 and ends at π, we can conclude that the two point variables, p_a and p_b iterate over every point at most once. Therefore, the number of iterations is $\mathcal{O}(n)$.
Therefore the total running time of the algorithm $= \mathcal{O}(n \lg n) + \mathcal{O}(n) = \mathcal{O}(n \lg n)$

2.2 Checking if two convex hulls intersect

Two convex hulls intersect if any one of the vertices of the first one lies strictly inside the other one. To do this, we first need to find which of the polygons is on the right of the other. We can do this by calculating their centroid (avg. of the x co-ordinates, avg. of the y co-ordinates). For the purposes of the pseudocode, let's assume the hull with m vertices is on the right. For each vertex for the left hull, we see if it lies inside the right hull. To see if a point lies inside a polygon, we draw a horizontal line from the vertex to just outside the right hull. We count the number of edges of the right hull this horizontal intersects. If it is even, then the vertex is outside the right hull, otherwise it's inside. Here is the pseudo code.

```
**function** CHECKPOLYGONINTERSECTION($ConvexHull\ A$, $ConvexHull\ B$)
    $P \leftarrow$ convex hull to the left.
    $Q \leftarrow$ convex hull to the right.
    $p_{1...n} \leftarrow$ points on $P$
    $q_{1...m} \leftarrow$ points on $Q$
    $rightmostX \leftarrow$the X co-ordinate of the point on $Q$ with the maximum X
co-ordinate
    for all $p$ in $p_{1...n}$ do
        if $isPointInPolygon(Q, p, rightmostX)$ then
            return $true$
        end if
    end for
    return $false$                    ▷ None of the points in P are inside Q
end function
function isPointInPolygon($Polygon\ Q$, $Point\ p$, $double\ maxCoordiante$)
$maxCoordiante$ is the X co-ordinate of the point in Q
which has the max X Co-ordinate (right most point)

    $horizontal = LineSegment(p, Point(maxCoordiante + 1, p_y)$    ▷ making
sure that the end of $horizontal$ is to the right of Q
    $q_{1...m} \leftarrow$ points on $Q$
    $count \leftarrow 0$
    for all $q_i$ in $q_{1...n}$ do
        $edge = LineSegment(q_i, q_{i+1})$        ▷ indices wrap around, $q_{m+1} = q_1$
        if $isIntersecting(horizontal, edge)$ then
            $count \leftarrow count + 1$
        end if
    end for
    if $isEven(count)$ then
        return $false$
    else
        return $true$
    end if
end function
function isPointInPolygon($LineSegment\ L_1$, $LineSegment\ L_2$)
    $p_1 = (x_1, y_1) \leftarrow$ starting point of $LineSegment\ L_1$
    $p_2 = (x_2, y_3) \leftarrow$ ending point of $LineSegment\ L_1$
    $p_3 = (x_3, y_3) \leftarrow$ starting point of $LineSegment\ L_2$
    $p_4 = (x_4, y_4) \leftarrow$ ending point of $LineSegment\ L_1$
    if $\neg[(x_2 \geq x_3) \wedge (x_4 \geq x_1)] \wedge [(y_2 \geq y_3) \wedge (y4 \geq y1)]$ then
        return $false$    ▷ If bounding boxes don't intersect, lines can't either
    end if
    $Vector\ v1 = (p_2 - p_1)$
    $Vector\ v2 = (p_3 - p_1)$
    $Vector\ v3 = (p_4 - p_1)$
    $product_1 \leftarrow v2$ x $v1$
    $product_2 \leftarrow v3$ x $v1$
    if $product_1$ x $product_2 \geq 0$ then
    ▷ If $product_1$ x $product_2 = 0$, one (or both) $p_3$ and $p_4$ is collinear with $p_1$ and $p2$
    ▷ If $product_1$ x $product_2 > 0$, both $product_1$ and $product_2$ have the same sign, and thus the points fail the straddle test
        return $false$
    end if
changing the reference points (swapping roles) and doing the test again

    $v1 \leftarrow (p_3 - p_4)$
    $v2 \leftarrow (p_1 - p_3)$
    $v3 \leftarrow (p_2 - p_3)$
    $product_1 \leftarrow v2$ x $v1$
    $product_2 \leftarrow v3$ x $v1$
    if $product_1$ x $product_2 \geq 0$ then
    ▷ If $product_1$ x $product_2 = 0$, one (or both) $p_3$ and $p_4$ is collinear with $p_1$ and $p2$
    ▷ If $product_1$ x $product_2 > 0$, both $product_1$ and $product_2$ have the same sign, and thus the points fail the straddle test
        return $false$
    end if
    return $true$    ▷ the points passed the rejection test and the two straddle tests, therefore they are strictly intersecting
end function
```

This algorithm works in $\Theta(mn) time$. Let's analyze why. These are main tasks in this algorithm:

- Finding which hull is to the right. ($\Theta(n + m)$)
- Finding the right-most point of the right hull. ($\Theta(m)$)
- Iterating through the vertices of the left hull and seeing if the vertex lies inside the the right hull

Inspecting the above algorithm, we see that to find if a point is lying inside a polygon of m sides, takes $\Theta(m)$ time because we iterate over all it's n edges once. We do this for each of the n vertices of the left hull, taking $\Theta(m)$ each time, totally taking $n * \Theta(m) = \Theta(mn)$ time.
Therefore the total running time of the algorithm $= \Theta(n+m) + \Theta(m) + \Theta(mn) = \Theta(mn)$

2.3 Nearest pair of vertices of two convex hulls

We shall use a slight modification of the rotating-callipers algorithm to find our answer. We will iteratively find all the anti-podal points between the polygons. A pair of anti-podal points between two convex polygons is defined a pair of points (one from each polygon) through which a pair of anti-parallel tangents can be drawn to the respective polygons which does not intersect any other edge of the polygon. We can generate all the antipodal points by first determining one pair and then rotating the tangents minimally along the sides of the polygon till they pass another set of antipodal points. In our algorithm, we can find our first anti-podal pair as the pair of points, first of which has the minimum y co-ordinate amongst the points of the first hull and the other the maximum y co-ordinate amongst the set of points in the other hull. Let's assume two methods $angle(line_a, line_b)$ that returns the angle between lines $line_a$ and $line_b$ and $distance(point_a, point_b)$ that returns the distance between points $point_a$ and $point_b$. The anti-podal pair with the minimum distance between them is our required pair. If there are many such pairs, we return the first one to be found. Here is the pseudo-code:

```
$n \leftarrow$number of vertices in $P$
$m \leftarrow$number of vertices in $Q$
$p_1 \leftarrow$ point in $P$ with minimum y-co-ordinate
$p_2 \dots p_n \leftarrow$ points sorted CCW according to polar angle from $p_1 \dots (1)$
$q_1 \leftarrow$ point in $Q$ with maximum y-co-ordinate
$q_2 \dots q_n \leftarrow$ points sorted CCW according to polar angle from $q_1 \dots (1)$
$p_a \leftarrow p_1$
$q_b \leftarrow q_1$
$angleRotated \leftarrow 0$
$minDistance \leftarrow distance(p_a, q_b)$
$nearestPair \leftarrow (p_a, q_b)$
$supportingLine_a \leftarrow$ horizontal vector passing thro' $p_a$ towards +ve X axis
$supportingLine_b \leftarrow$ horizontal vector passing thro' $q_b$ towards -ve X axis
**while** $angleRotated < 2\pi$ **do**
    $edge_a \leftarrow edge(p_a, p_{a+1})$
    $edge_b \leftarrow edge(q_b, q_{b+1})$
    $angle_a \leftarrow angle(supportingLine_a, edge_a)$
    $angle_b \leftarrow angle(supportingLine_b, edge_b)$
    **if** $angle_a < angle_b$ **then**
        **if** $distance(p_{a+1}, q_b) < minDistance$ **then**
            $minDistance \leftarrow distance(p_{a+1}, q_b)$
            $nearestPair \leftarrow (p_{a+1}, q_b)$
        **end if**
        $p_a \leftarrow p_{a+1}$ (This index wraps around. $p_{n+1} = p_1$)
    **else if** $angle_b < angle_a$ **then**
        **if** $distance(p_a, q_{b+1}) < minDistance$ **then**
            $minDistance \leftarrow distance(p_a, q_{b+1})$
            $nearestPair \leftarrow (p_a, q_{b+1})$
        **end if**
        $p_b \leftarrow q_{b+1}$ (This index also wraps around. $q_{m+1} = q_1$)
    **else** (i.e. $angle_a = angle_b$)
        **if** $distance(p_{a+1}, q_b) < minDistance$ **then**
```

```
            $minDistance \leftarrow distance(p_{a+1}, q_b)$
            $nearestPair \leftarrow (p_{a+1}, q_b)$
        else if $distance(p_a, p_{b+1}) < minDistance$ then
            $minDistance \leftarrow distance(p_a, q_{b+1})$
            $nearestPair \leftarrow (p_a, q_{b+1})$
        else if $distance(p_{a+1}, p_{b+1}) < minDistance$ then
            $minDistance \leftarrow distance(p_{a+1}, q_{b+1})$
            $nearestPair \leftarrow (p_{a+1}, q_{b+1})$
        end if
        $p_a \leftarrow p_{a+1}$
        $q_b \leftarrow q_{b+1}$ (These indices wrap around)
    end if
    $angleRotated \leftarrow angleRotated + min(angle_a, angle_b)$
end while
return $nearestPair$
```

This algorithm works in $\mathcal{O}(n \lg n) time$. Let's analyze why. There are two main tasks in this algorithm:

- Sorting vertices of P in CCW order from p_1. $\mathcal{O}(n \lg n)$
- Sorting vertices of Q in CCW order from q_1. $\mathcal{O}(m \lg m)$
- Iterating through the vertices optimally to determine the nearest

Inspecting the above algorithm, we see that we by since *angleRotated* starts from 0 and ends at 2π, we can conclude that the two point variables, p_a and q_b iterate over each vertex of their respective polygons at most once. Therefore, the number of iterations is $\mathcal{O}(n + m)$.
Therefore the total running time of the algorithm $= \mathcal{O}(n \lg n) + \mathcal{O}(m \lg m) + \mathcal{O}(n + m) = \mathcal{O}(n \lg n)$ (for asymptotic upper bound)

3 Question 3

3.1

```
int fib1(n)
{
        if(n==0||n==1)
                return Soln[n];
        Soln[n−1] = fib1(n−1);
        If (Soln[n−2] == infinity)
                Soln[n−2] = fib(n−2);
        Soln[n]= Soln[n−1]+Soln[n−2];
        return Soln[n];
}
```

Let T(n) be the time required to compute fib1(n).
Observations :
If n$\geq$ 2, the recursive call fib1(n-1) is called. That in turn computes Soln[n-2], so Soln[n-2] will never be ∞ and the statement Soln[n-2] = fib (n-2) is never executed and can be ignored.

Therefore,

$$\begin{aligned}
T(0) &= \Theta(1) \\
T(1) &= \Theta(1) \\
T(n) &= T(n-1) + \Theta(1) \\
&= T(n-2) + \Theta(1) + \Theta(1) \\
&\vdots \\
&= T(n-k) + k\Theta(1)
\end{aligned}$$

taking n = k,

$$\begin{aligned}
&= T(0) + n\Theta(1) \\
&= \Theta(n) \\
T(n) &= \Theta(n)
\end{aligned}$$

Space complexity is $\mathcal{O}(n)$ because intermediate values are stored in an array of size n.

3.2

```
int fib2(n)
{
        if(n==0||n==1)
                return Soln[n];
        Soln[n-2] = fib2(n-2);
        Soln[n-1] = fib2(n-1);
        return (Soln[n-1]+Soln[n-2]);
}
```

Let T(n) be the time required to compute fib2(n).
Observations :

$$
\begin{aligned}
T(0) &= \Theta(1)\\
T(1) &= \Theta(1)\\
T(n) &= T(n-1) + T(n-2) + \Theta(1)\\
T(n-1) &> T(n-2)\\
T(n) &= T(n-1) + T(n-2) + \Theta(1)\\
&< 2T(n-1) + \Theta(1)\\
&< 2(2T(n-2)) + 2\Theta(1) + \Theta(1)\\
&\vdots\\
&< 2^k T(n-k) + 2^{k-1}\Theta(1) + \cdots + 2\Theta(1) + \Theta(1)\\
&< 2^k T(n-k) + \Theta(1)\sum_{i=0}^{k-1} 2^i\\
&< 2^k T(n-k) + \Theta(1).(2^k)\\
&< 2^k T(n-k) + 2^k\Theta(1)
\end{aligned}
$$

taking n = k,

$$
\begin{aligned}
&< 2^n[T(0) + \Theta(1)]\\
&< 2^n[\Theta(1) + \Theta(1)]\\
&< 2.2^n\Theta(1)\\
T(n) &= \mathcal{O}(2^n)
\end{aligned}
$$

Space complexity is $\mathcal{O}(n)$ because intermediate values are stored in an array of size n.

3.3

```
int fib3(n)
{
        if (n==0||n==1)
                return Soln[n];
        Soln[n-1] = fib3(n-1);
        Soln[n] = Soln[n-1]+Soln[n-2];
        return Soln[n];
}
```

Let T(n) be the time required to compute fib3(n).
Observations :

$$\begin{aligned} T(0) &= \Theta(1) \\ T(1) &= \Theta(1) \\ T(n) &= T(n-1) + \Theta(1) \\ &= T(n-2) + \Theta(1) + \Theta(1) \\ &\vdots \\ &= T(n-k) + k\Theta(1) \end{aligned}$$

taking n = k,

$$\begin{aligned} &= T(0) + n\Theta(1) \\ &= \Theta(n) \\ T(n) &= \Theta(n) \end{aligned}$$

Space complexity is $\mathcal{O}(n)$ because intermediate values are stored in an array of size n.

3.4

```
int fib4(n)
{
        if(n==0||n==1)
                return Soln[n];
        Soln[n-1] = fib4(n-1);
        return Soln[n-1] + Soln[n-2];
}
```

Let T(n) be the time required to compute fib4(n).
Observations :

$$\begin{aligned}
T(0) &= \Theta(1)\\
T(1) &= \Theta(1)\\
T(n) &= T(n-1) + \Theta(1)\\
&= T(n-2) + \Theta(1) + \Theta(1)\\
&\vdots\\
&= T(n-k) + k\Theta(1)
\end{aligned}$$

taking n = k,

$$\begin{aligned}
&= T(0) + n\Theta(1)\\
&= \Theta(n)\\
T(n) &= \Theta(n)
\end{aligned}$$

Space complexity is $\mathcal{O}(n)$ because intermediate values are stored in an array of size n.