-1

آ) برای حل این سوال از Binary Search استفاده می‌کنیم. در هر مرحله، فرض می‌کنیم در حال بررسی اندیس‌های i تا j آرایه A هستیم. می‌دانیم که در مرحله اول، i و j به ترتیب برابر با 0 و n-1 هستند. ابتدا اندیس وسط آرایه را با استفاده از رابطه $mid = \frac{i+j}{2}$ پیدا می‌کنیم. حالا باید A[mid] را با mid مقایسه کنیم. 3 حالت ممکن است رخ دهد:

1) A[mid] = mid

2) A[mid] > mid

3) A[mid] < mid

حالت اول که مطلوب مسئله است و mid را return می‌کنیم. در حالت دوم، با توجه به اینکه آرایه A شامل اعداد صحیح است، می‌دانیم که عنصر مورد نظر در سمت راست آرایه نخواهد بود، زیرا اندیس مورد نظر یک واحد یک واحد افزایش پیدا می‌کند و عناصر آرایه هم حداقل یک واحد با هم تفاوت دارند. در نتیجه اختلاف A[mid] و mid اگر بیشتر نشود، کمتر نمی‌شود. به همین دلیل می‌توانیم جست و جو را در سمت چپ آرایه انجام دهیم. به دلیل مشابه، در حالت سوم، جست و جو را در سمت راست آرایه ادامه می‌دهیم. با توجه به اینکه هر بار نصف آرایه را کنار می‌گذاریم، هزینه الگوریتم $O(\log n)$ خواهد بود.

```
function FindFixPoint(A, i, j) do
    if i > j do
        return null
    end

    set mid = (i + j) / 2
    if A[mid] == mid do
        return mid
    end
    if A[mid] > mid do
        return FindFixPoint(A, i, mid - 1)
    end
    return FindFixPoint(A, mid + 1, j)
end
```

ب) در حل این سوال هم از Binary Search استفاده می‌کنیم. ابتدا اندیس عنصر وسط آرایه را پیدا می-کنیم و ماکسیمال بودن این عنصر را بررسی می‌کنیم. اگر ماکسیمال بود، آن را return می‌کنیم. اما اگر ماکسیمال نباشد، حداقل یکی از همسایه‌های عنصر وسط، از این عنصر بزرگ‌تر است و در این حالت، قطعا در سمت عنصر بزرگ‌تر، یک عنصر ماکسیمال وجود دارد و می‌توانیم تابع را روی آن سمت صدا بزنیم و بخش دیگر را کنار بگذاریم. با توجه به اینکه هر بار نصف آرایه کنار گذاشته می‌شود و بررسی نمی‌شود، زمان انجام این الگوریتم $O(\log n)$ است.

```
function FindPeak(A, i, j) do
    set mid = (i + j) / 2

    if (mid == 0 or A[mid - 1] ≤ A[mid]) and
       (mid == A.length - 1 or A[mid + 1] ≤ A[mid]) do
        return mid
    end

    if mid - 1 ≥ 0 and A[mid - 1] > A[mid] do
        return FindPeak(A, i, mid - 1)
    end

    return FindPeak(A, mid + 1, j)
end
```

-2

آ) ابتدا آرایه را از وسط نصف می‌کنیم. بخش سمت راست و بخش سمت چپ را به صورت جداگانه حل می‌کنیم و بزرگ‌ترین زیرآرایه مطلوب را در هر بخش پیدا می‌کنیم. در زمان merge کردن 2 بخش، 3 حالت ممکن است برای بزرگ‌ترین زیرآرایه مطلوب پیش آید:

1) زیرآرایه مطلوب کاملا در بخش سمت راست باشد که در این حالت، همان زیرآرایه‌ای است که از حل جداگانه بخش سمت راست محاسبه شده است.

2) زیرآرایه مطلوب کاملا در بخش سمت چپ قرار گیرد که مشابه مورد قبل، همان زیرآرایه‌ای است که از حل جداگانه بخش سمت چپ محاسبه شده است.

3) زیرآرایه مطلوب شامل قسمتی از بخش سمت چپ و قسمتی از بخش سمت راست آرایه باشد. در این حالت، شروع زیرآرایه در سمت چپ و پایان آن در سمت راست قرار می‌گیرد.

برای محاسبه حالت سوم، دو اندیس را در وسط آرایه در نظر می‌گیریم، به طوری که یکی از آن‌ها در قسمت چپ آرایه و دیگری در قسمت راست آرایه باشد. سپس اندیس چپی را به سمت چپ و اندیس راستی را به سمت راست حرکت می‌دهیم. این کار را تا زمانی ادامه می‌دهیم که شرط مسئله نقض نشود و بزرگ‌ترین زیرآرایه مطلوب ساخته شود. سپس، طول زیرآرایه بدست آمده را با زیرآرایه‌های بخش چپ و راست مقایسه کرده و بزرگ‌ترین آن‌ها را return می‌کنیم. با توجه به این که این کار را می‌توان در $\mathcal{O}(n)$ انجام داد (هر عضو آرایه حداکثر یک بار بررسی می‌شود)، و با توجه به رابطه $\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(n)$ می‌توان گفت زمان انجام الگوریتم برابر با $\mathcal{O}(n \log n)$ خواهد بود.

```
function FindLongestSubArray(A, i, j, k) do
    if i ⩾ j do
        return 0, null, null // Length, Start, Stop
    end

    set mid = (i + j) / 2

    set leftLength, leftStart, leftStop = FindLongestSubArray(A, i, mid, k)
    set rightLength, rightStart, rightStop = FindLongestSubArray(A, mid + 1, j, k)

    set length, start, stop = leftLength > rightLength ?
                              leftLength, leftStart, leftStop :
                              rightLength, rightStart, rightStop

    if abs(A[mid] - A[mid + 1]) ⩽ k do
        set l, r = mid, mid + 1

        while l ⩾ i and abs(A[l] - A[l + 1]) ⩽ k do
            l -= 1
        end

        while r ⩽ j and abs(A[r] - A[r - 1]) ⩽ k do
            r += 1
        end

        if r - l - 1 > length do
            length = r - l - 1
            start = l + 1
            stop = r - 1
        end
    end

    return length, start, stop
end
```

ب) برای حل این مسئله در زمان $O(n)$ می‌توانیم از یک حلقه ساده بر روی آرایه استفاده کنیم. از اول آرایه شروع به حرکت کرده و اولین زیرآرایه مطلوب را پیدا می‌کنیم و طول، شروع و پایان آن را ذخیره می‌کنیم. سپس به عنصر بعدی رفته و زیرآرایه بعدی را شکل می‌دهیم. پس از شکل گرفتن کامل هر زیرآرایه، طول آن را با بزرگ‌ترین زیرآرایه ذخیره شده مقایسه می‌کنیم و در صورتی که زیرآرایه جدید بزرگ‌تر بود، آن را جایگزین زیرآرایه قبلی می‌کنیم. در نهایت بزرگ‌ترین زیرآرایه بدست آمده را return می‌کنیم. با توجه به اینکه در این روش آرایه حداکثر یک بار پیمایش می‌شود، زمان انجام این الگوریتم $O(n)$ خواهد بود.

```
function FindLongestSubArray(A, k) do
    set maxLength, start, stop = 0, null, null
    set i = 0
    while i < A.length do
        set j = i
        while i + 1 < A.length and
              abs(A[i] - A[i + 1]) ⩽ k do
            i += 1
        end
        set len = i - j + 1
        if len > maxLength do
            maxLength = len
            start = j
            stop = i
        end
    end
    return maxLength, start, stop
end
```

3- آرایه A شامل سه‌تایی‌های مرتب را در نظر می‌گیریم. ابتدا این آرایه را بر اساس تاریخ تولد افراد و به صورت صعودی مرتب می‌کنیم که هزینه این کار، $O(n\log n)$ است. سپس آرایه را به دو قسمت مساوی تقسیم می‌کنیم. هر قسمت را به صورت جداگانه حل کرده و بزرگ‌ترین بازه مطلوب را در هر قسمت بدست می‌آوریم. برای بازه نهایی، سه حالت ممکن است اتفاق افتد که باید بزرگ‌ترین حالت را return کنیم:

1) بازه نهایی، همان بازه بدست آمده از حل قسمت راست باشد.
2) بازه نهایی، همان بازه بدست آمده از حل قسمت چپ باشد.
3) بازه نهایی حاصل از همپوشانی دو بازه باشد که یکی از آن‌ها در قسمت چپ و دیگری در قسمت راست باشد.

برای بدست آوردن بزرگ‌ترین بازه در حالت سوم، ابتدا در گروه سمت چپ، فرد با بزرگ‌ترین زمان مرگ را پیدا می‌کنیم. می‌دانیم قطعا بزرگ‌ترین همپوشانی افراد در گروه راست، با این فرد رخ می‌دهد. سپس همپوشانی تمام افراد گروه راست را با این فرد محاسبه کرده و بزرگ‌ترین آن را return می‌کنیم. در این حالت، هر عضو آرایه حداکثر یک بار بررسی می‌شود و در نتیجه هزینه این کار $O(n)$ است: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

که نشان می‌دهد هزینه کل الگوریتم تقسیم و حل برابر با $O(n\log n)$ است که با توجه به اینکه الگوریتم مرتب‌سازی هم هزینه‌ای برابر با $O(n\log n)$ داشت، هزینه کل الگوریتم برابر با $O(n\log n)$ خواهد بود.

```
function FindMaximumMiddleOverlap(A, mid) do
    if A.length ≤ 1 do
        return 0, null, null
    end
    set maxDeath = A[0]
    for i from 1 to mid do
        if A[i].death > maxDeath.death do
            maxDeath = A[i]
        end
    end
    set second = A[mid + 1]
    set maxOverlap = overlap(maxDeath, second)
    for i from mid + 2 to A.length - 1 do
        if overlap(A[i], maxDeath) > maxOverlap do
            maxOverlap = overlap(A[i], maxDeath)
            second = A[i]
        end
    end
    return maxOverlap, maxDeath, second
end

function FindMaximumOverlap(A, i, j) do // A is sorted by birth
    if A.length ≤ 1 do
        return 0, null, null
    end
    set mid = (i + j) / 2
    set leftMaxOverlap, leftFirst, leftSecond = FindMaximumOverlap(A, i, mid)
    set rightMaxOverlap, rightFirst, rightSecond = FindMaximumOverlap(A, mid + 1, j)
    set maxOverlap, first, second = leftMaxOverlap > rightMaxOverlap ?
                                    leftMaxOverlap, leftFirst, leftSecond :
                                    rightMaxOverlap, rightFirst, rightSecond

    set midMaxOverlap, midFirst, midSecond = FindMaximumMiddleOverlap(A, mid) // O(n)
    if midMaxOverlap > maxOverlap do
        maxOverlap = midMaxOverlap
        first = midFirst
        second = midSecond
    end
    return maxOverlap, first, second // first.name, second.name
end
```

4- برای پیدا کردن جمع مورد نظر، از Binary Search استفاده می‌کنیم. می‌دانیم که بیشترین جمع عناصر یک زیرآرایه حداقل برابر با بزرگ‌ترین عضو آرایه و حداکثر برابر با جمع کل عناصر آرایه است. برای پیدا کردن کمترین مقدار آن، از باینری سرچ استفاده می‌کنیم، به اینصورت که اگر جمع‌های ممکن را به صورت یک مجموعه در نظر بگیریم، عنصر وسط مجموعه را پیدا کرده و بررسی می‌کنیم که آیا امکان دارد این مقدار، بزرگ-ترین جمع m زیرآرایه باشد یا خیر (نحوه بررسی در ادامه آورده شده است). اگر این امکان وجود داشت، برای پیدا کردن کوچک‌ترین مقدار، نیمه کوچک‌تر مجموعه را بررسی می‌کنیم. اما اگر این امکان وجود نداشت، جهت پیدا کردن یک مقدار ممکن، نیمه بزرگ‌تر را بررسی می‌کنیم.

برای بررسی اینکه یک مقدار می‌تواند برابر با بزرگ‌ترین جمع m زیرآرایه باشد یا نه، به روش زیر عمل می‌کنیم: از ابتدای آرایه شروع کرده و بزرگ‌ترین زیرآرایه‌هایی را تشکیل می‌دهیم که جمع عناصر آن‌ها از مقدار مورد نظر بیشتر نشود. تعداد این زیرآرایه‌ها را می‌شماریم. اگر این تعداد از m بیشتر نشود، مقدار مورد نظر می‌تواند مقدار مطلوب باشد و در غیر اینصورت نمی‌توان به همچین مقداری رسید.

اگر جمع تمام عناصر آرایه را k در نظر بگیریم، در واقع کار الگوریتم، انجام عمل باینری سرچ روی مقادیر مثبت کوچک‌تر و یا مساوی k است. همچنین در هر مرحله، برای بررسی امکان رسیدن به جمع بدست آمده، یک بار کل آرایه را بررسی می‌کنیم. در نتیجه می‌توان گفت که هزینه کل الگوریتم برابر با $\mathcal{O}(n\,log\,k)$ خواهد بود.

```
function findMinimizedMaximumSum(A, k, m) do
    set start, stop = max(A), k // k is sum of elements in A
    set minSum = 0
    while start ⩽ stop do
        set mid = (start + stop) / 2
        if isSumPossible(A, mid, m) do
            stop = mid - 1
            minSum = mid
        else
            start = mid + 1
        end
    end
    return minSum
end
```

```
function isSumPossible(A, value, m) do
    set count, sum = 0, 0
    for i from 0 to to A.length - 1 do
        if A[i] > value do
            return false
        end
        if sum + A[i] ⩽ value do
            sum += A[i]
        else
            sum = A[i]
            count += 1
        end
    end
    return count ⩽ m
end
```

5- ابتدا آرایه را پیمایش کرده و نقطه با بیشترین y که همان بالاترین نقطه در صفحه است را پیدا می‌کنیم. این نقطه ریشه درخت است. سپس باقی نقاط را برحسب شیب خطی که با ریشه می‌سازند مرتب می‌کنیم. سپس آرایه را از وسط به دو قسمت تقسیم کرده و الگوریتم را به صورت بازگشتی رو هر قسمت اجرا می‌کنیم. این کار را تا زمانی ادامه می‌دهیم که درخت باینری به ارتفاع k به طور کامل ساخته شود. با توجه به اینکه نقاط را بر اساس شیب به دو قسمت تقسیم کرده‌ایم، یال‌های دو زیردرخت همدیگر را قطع نمی‌کنند. در هر مرحله ابتدا یک پیمایش روی آرایه خواهیم داشت که زمان انجام آن $\mathcal{O}(n)$ است. سپس یک مرتب‌سازی با هزینه $\mathcal{O}(n\log n)$ انجام می‌دهیم و در نهایت مسئله را به دو زیرمسئله کوچک‌تر با اندازه $\frac{\mathrm{n}}{2}$ تقسیم می‌کنیم. با توجه به رابطه $\mathcal{T}(n) = 2\mathcal{T}\left(\frac{\mathrm{n}}{2}\right) + \mathcal{O}(n\log n)$ زمان انجام الگوریتم $\mathcal{O}(n\log^2 n)$ خواهد بود.

```
function makeSubTree(A, mainArr, i, depth, k) do
    set root = max(A, (point) → point.y) // find maximum y
    mainArr[root.index].answer = i
    if depth == k do
        return
    end
    A.Remove(root)
    set sorted = sortBySlope(A, root) // O(n log n)
    set mid = sorted.length / 2
    makeSubTree(sorted[..mid], mainArr, i * 2, depth + 1, k) // T(n/2)
    makeSubTree(sorted[mid + 1..], mainArr, i * 2 + 1, depth + 1, k) // T(n/2)
end

function makeBinaryTree(mainArr, k) do
    makeSubTree(A, mainArr, 1, 0, k)
end
```

6- می‌دانیم که M_k را می‌توان به صورت $\begin{bmatrix} M_{k-1} & M_{k-1} \\ M_{k-1} & -M_{k-1} \end{bmatrix}$ نشان داد. برای بدست آوردن حاصل ضرب M_kV ابتدا ماتریس V را به صورت $V = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix}$ نشان می‌دهیم. در این صورت M_kV به شکل زیر قابل محاسبه است:

$$M_kV = \begin{bmatrix} M_{k-1} & M_{k-1} \\ M_{k-1} & -M_{k-1} \end{bmatrix} \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} M_{k-1}V_1 + M_{k-1}V_2 \\ M_{k-1}V_1 - M_{k-1}V_2 \end{bmatrix}$$

در این حالت اگر محاسبه M_kV را $\mathcal{T}(n)$ در نظر بگیریم، با توجه به اینکه $n = 2^k$ است، M_{k-1} یک ماتریس $2^{k-1} \times 2^{k-1}$ خواهد بود که با توجه به اینکه $\frac{2^{k-1}}{2^k} = \frac{1}{2}$ است، محاسبه $M_{k-1}V_i$ معادل $\mathcal{T}\left(\frac{n}{2}\right)$ خواهد بود. همچنین ما فقط به محاسبه $M_{k-1}V_1$ و $M_{k-1}V_2$ نیاز داریم و می‌دانیم که اعمال $M_{k-1}V_1 \pm M_{k-1}V_2$ (جمع و تفریق) در زمان $\mathcal{O}(n)$ انجام می‌شوند. در نتیجه رابطه بازگشتی به صورت زیر خواهد بود:

$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{O}(n) \longrightarrow \mathcal{T}(n) = \mathcal{O}(n \log n)$$