Dynamic Programming

Dynamic programming is a method for solving a complex problem by breaking it down into simpler subproblems, solving each of those subproblems just once, and storing their solutions – in an array(usually).

Now, every time the same sub-problem occurs, instead of recomputing its solution, the previously calculated solutions are used, thereby saving computation time at the expense of storage space.

```
1+1+1+1+1+1+1+1 =8


Ans+1=9

Ans1+2=Ans2
```

```
*writes down "1+1+1+1+1+1+1+1 =" on a sheet of paper*
"What's that equal to?"
*counting* "Eight!"
*writes down another "1+" on the left*
"What about that?"
*quickly* "Nine!"
"How'd you know it was nine so fast?"
"You just added one more."
So you didn't need to recount because you remembered there were eight!
Dynamic Programming is just a fancy way to say 'remembering stuff to save time later'
```

Dynamic programming can be implemented in two ways –

- Memoization
- Tabulation

Memoization – Memoization uses the top-down technique to solve the problem i.e. it begin with original problem then breaks it into sub-problems and solve these sub-

problems in the same
way.

In this approach, you assume that you have already computed all subproblems. You typically perform a recursive call (or some iterative equivalent) from the main problem. You ensure that the recursive call never recomputes a subproblem because you cache the results, and thus
duplicate sub-problems are not recomputed.

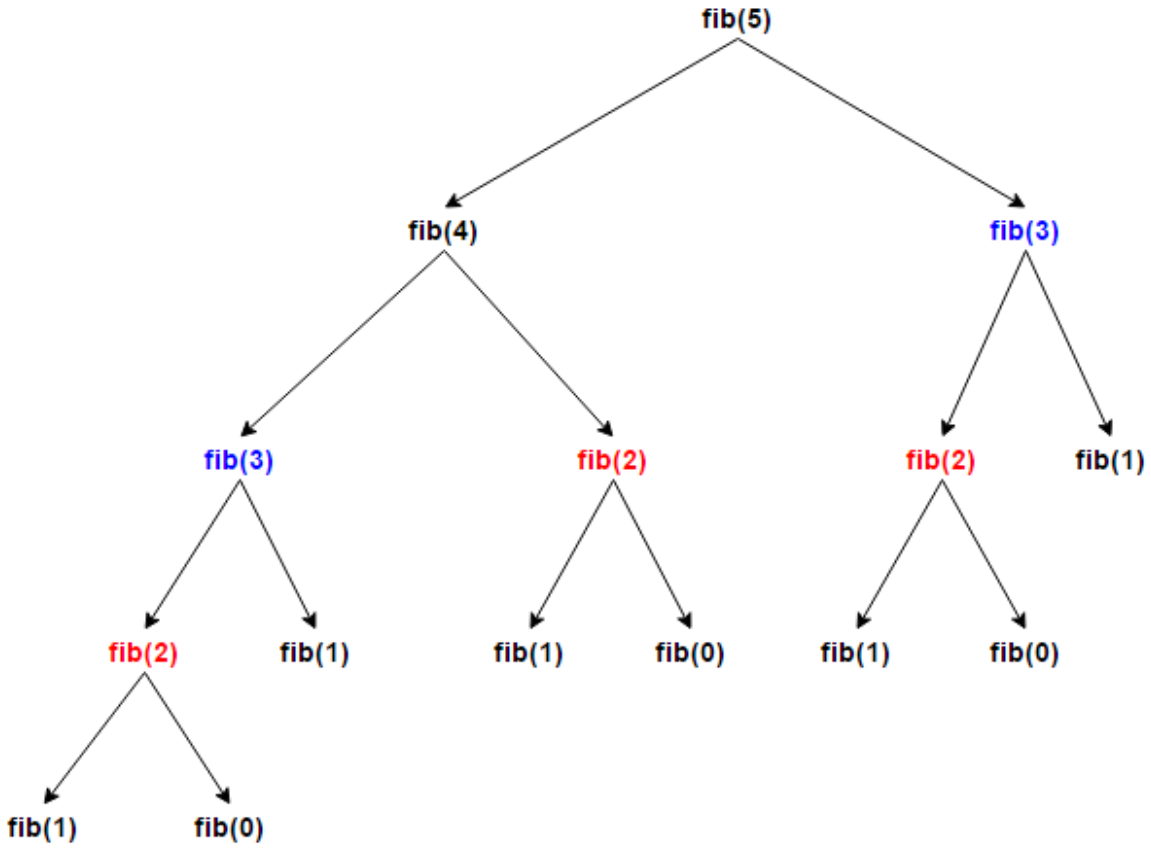
Tabulation – Tabulation is the typical Dynamic Programming approach. Tabulation uses the bottom-up approach to solve the problem, i.e., by solving all related sub-problems first, typically
by storing the results in an array. Based on the results stored in the array, the solution to the “top” / original problem is then computed.

Memoization and tabulation are both storage techniques applied to avoid recomputation of a subproblem

Consider a program to generate Nth fibonacci number
Fib(n)=Fib(n-1)+Fib(n-2)

```
//using top-down approach without Dynamic Programming
int Fib(int n)
{
     if(n<=1)
     {
          return n;
     }
     else
     {
          return (Fib(n-1)+Fib(n-2));
     }
}
```

Notice that if we call, say, `fib(5)`, we produce a call tree that calls the function on the same value many times:



In particular, `fib(3)` was calculated twice, and `fib(2)` was calculated three times from scratch. In larger examples, many more subproblems are recalculated, leading to an exponential-time algorithm.

Now, suppose we have a simple map object, `lookup`, which maps each value of `fib` that has already been calculated to its result, and we modify our function to use it and update it. The resulting function runs in O(n) time instead of exponential time (but requires O(n) space):

```
//using top-down approach with Memoization (Dynamic Programming)
int memoize[];
    //method to initialize memoize array to -1
    void initialize()
    {
      ...
    }
    int Fib(int n)
    {
      if(memoize[n]==-1)
      {
        //means the solution is not yet calculated
        if(n<=1)
        {
```

```
          memoize[n]=1;
        }
        else
        {
          memoize[n]=Fib[n-1]+Fib[n-2];
        }
      }
      return memoize[n];
    }
```

```
import java.util.HashMap;
import java.util.Map;

class fibonacci
{
    // Function to find the n'th Fibonacci number
    public static int fib(int n, Map<Integer, Integer> lookup)
    {
        if (n <= 1) {
            return n;
        }

        // if the subproblem is seen for the first time
        if (!lookup.containsKey(n))
        {
            int val = fib(n - 1, lookup) + fib(n - 2, lookup);
            lookup.put(n, val);
        }

        return lookup.get(n);
    }

    public static void main(String[] args)
    {
        int n = 8;
        Map<Integer, Integer> lookup = new HashMap<>();

        System.out.println(fib(n, lookup));
    }
}
```

n the bottom-up approach, we calculate the smaller values of `fib` first, then build larger values from them. This method also uses O(n) time since it contains a loop that repeats `n-1` times, but it only takes constant O(1) space, in contrast to the top-down approach, which requires O(n) space to store the map.

```
//Bottom up Dynamic Programming
int table[N];
    void setup_fib()
    {
        table[0] = 1;
```

```
        table[1] = 1;
        for (int i = 2; i < N; i++)
           table[i] = table[i-1] + table[i-2];
    }

    int Fib(int x)
    {
      return table[x];
    }
```

```
class fibonacci
{
    // Function to find n'th Fibonacci number
    public static int fib(int n)
    {
        if (n <= 1) {
            return n;
        }

        int previousFib = 0, currentFib = 1;
        for (int i = 0; i < n - 1; i++)
        {
            int newFib = previousFib + currentFib;
            previousFib = currentFib;
            currentFib = newFib;
        }

        return currentFib;
    }

    public static void main(String[] args) {
        System.out.print(fib(8));
    }
}
```

The idea behind dynamic programming, In general, is to solve a given problem, by solving different parts of the problem (subproblems), then using the cached solutions of the subproblems to reach an overall solution.

APPLICABILITY OF DYNAMIC PROGRAMMING-

The problems that can be solved by using Dynamic Programming has the following two main properties-

1. Overlapping sub-problems
2. Optimal Substructure

1**) Overlapping Subproblems:**

Overlapping subproblems is a property in which a problem can be broken down into subproblems which are used multiple times.

Dynamic Programming is mainly used when solutions of same subproblems are needed again and again. In dynamic programming, computed solutions to subproblems are stored in a array so that these don't have to recomputed. So Dynamic Programming is not useful when there are no overlapping subproblems because there is no point storing the solutions if they are not needed again.

A problem is said to have overlapping subproblems if the problem can be broken down into subproblems and each subproblem is repeated several times, or a recursive algorithm for the problem solves the same subproblem repeatedly rather than always generating new subproblems.

For example, the problem of computing the Fibonacci sequence exhibits overlapping subproblems. The problem of computing the `n'th` Fibonacci number `F(n)` can be broken down into the subproblems of computing `F(n-1)` and `F(n-2)`, and then adding the two. The subproblem of computing `F(n-1)` can itself be broken down into a subproblem that involves computing `F(n-2)`. Therefore, the computation of `F(n-2)` is reused, and the Fibonacci sequence thus exhibits overlapping subproblems. Dynamic programming takes account of this fact and solves each subproblem only once. This can be achieved in either of two ways:

1. **Top-down approach (Memoization):** This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems overlap, one can easily memoize or store the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, first check the table to see if it is already solved. If the subproblem is already solved, use its solution directly; otherwise, solve the subproblem and add its solution to the table.
2. **Bottom-up approach (Tabulation):** Once we formulate the solution to a problem recursively as in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger subproblems by using the solutions to small subproblems. For example, if we already know the values of `F(i-1)` and `F(i-2)`, we can directly calculate the value of `F(i)`.

2) Optimal substructure

Optimal substructure is a property in which an optimal solution of the original problem can be constructed efficiently from the optimal solutions of its sub-problems.

Dynamic programming simplifies a complicated problem by breaking it down into simpler subproblems in a recursive manner. A problem that can be solved optimally by breaking it into subproblems and then recursively finding the optimal solutions to the subproblems is said to have an optimal substructure. In other words, the solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems.

For example, the shortest path `p` from a vertex `u` to a vertex `v` in a given graph exhibits optimal substructure: take any intermediate vertex `w` on this shortest path `p`. If `p` is truly the shortest path, then it can be split into subpaths `p1` from `u` to `w` and `p2` from `w` to `v` such that these, in turn, are indeed the shortest paths between the corresponding vertices.

If a given problem obey both these properties, then the problem can be solved by using Dynamic Programming.

Steps to follow for solving a DP problem –

1. Express a solution mathematically
2. Express a solution recursively
3. Either develop a bottom up algorithm or top-down memoized algorithm

Find all n-digit binary numbers without any consecutive 1's

Given a positive integer `n`, count all n–digit binary numbers without any consecutive `1's`.

For example, for `n = 5`, the binary numbers that satisfy the given constraints are: `[00000, 00001, 00010, 00100, 00101, 01000, 01001, 01010, 10000, 10001, 10010, 10100, 10101]`.

```
class Ndigit1
{
    // count all n–digit binary numbers without any consecutive 1's
```

```
    public static int countStrings(int n, int lastDigit)
    {
        // base case
        if (n == 0) {
            return 0;
        }

        // if only one digit is left
        if (n == 1) {
            return (lastDigit == 1) ? 1: 2;
        }

        // if the last digit is 0, we can have both 0 and 1 at the current position
        if (lastDigit == 0) {
            return countStrings(n - 1, 0) + countStrings(n - 1, 1);
        }
        // if the last digit is 1, we can have only 0 at the current position
        else {
            return countStrings(n - 1, 0);
        }
    }

    public static void main(String[] args)
    {
        // total number of digits
        int n = 5;

        System.out.println("The total number of " + n + "–digit binary numbers " +
                "without any consecutive 1's are " + countStrings(n, 0));
    }
}
```

The time complexity of the above solution is exponential and occupies space in the call stack.

The problem has an optimal substructure and exhibits overlapping subproblems. If we draw the solution's recursion tree, we can see that the same subproblems are getting computed repeatedly. We know that problems with optimal substructure and overlapping subproblems can be solved using dynamic programming, in which subproblem solutions are memoized rather than computed repeatedly.

```
class Ndigit2
{
    // Function to count all n–digit binary numbers without any consecutive 1's
    public static int countStrings(int n)
    {
        int[][] T = new int[n+1][2];

        // if only one digit is left
        T[1][0] = 2;
        T[1][1] = 1;
```

```
        for (int i = 2; i <= n; i++)
        {
            // if the last digit is 0, we can have both 0 and 1 at the current position
            T[i][0] = T[i-1][0] + T[i-1][1];

            // if the last digit is 1, we can have only 0 at the current position
            T[i][1] = T[i-1][0];
        }

        return T[n][0];
    }

    public static void main(String[] args)
    {
        // total number of digits
        int n = 5;

        System.out.print("The total number of " + n + "-digit binary numbers " +
                "without any consecutive 1's are " + countStrings(n));
    }
}
```

The time complexity of the above solution is O(n) and requires O(n) extra space, where `n` is the total number of digits

```
class Ndigit3
{
    // Function to print all n-digit binary numbers without any consecutive 1's
    public static void countStrings(int n, String out, int last_digit)
    {
        // if the number becomes n-digit, print it
        if (n == 0)
        {
            System.out.println(out);
            return;
        }

        // append 0 to the result and recur with one less digit
        countStrings(n - 1, out + '0', 0);

        // append 1 to the result and recur with one less digit
        // only if the last digit is 0
        if (last_digit == 0) {
            countStrings(n - 1, out + '1', 1);
        }
    }

    public static void main(String[] args)
    {
        // total number of digits
        int n = 5;

        String out = "";
        countStrings(n, out, 0);
```

```
    }
}
```

Count all paths in a matrix from the first cell to the last cell

Given an `M × N` rectangular grid, efficiently count all paths starting from the first cell `(0, 0)` to the last cell `(M-1, N-1)`. We can either move down or move towards right from a cell.

```
Input:  3 × 3 matrix

Output: Total number of paths are 6

(0, 0) —> (0, 1) —> (0, 2) —> (1, 2) —> (2, 2)
(0, 0) —> (0, 1) —> (1, 1) —> (1, 2) —> (2, 2)
(0, 0) —> (0, 1) —> (1, 1) —> (2, 1) —> (2, 2)
(0, 0) —> (1, 0) —> (2, 0) —> (2, 1) —> (2, 2)
(0, 0) —> (1, 0) —> (1, 1) —> (1, 2) —> (2, 2)
(0, 0) —> (1, 0) —> (1, 1) —> (2, 1) —> (2, 2)
```

```
class CountAllPath1
{
    // Top-down recursive function to count all paths from cell (m, n)
    // to the last cell (M-1, N-1) in a given `M × N` rectangular grid
    public static int countPaths(int m, int n, int M, int N)
    {
        // there is only one way to reach the last cell
        // when we are at the last row or the last column
        if (m == M - 1 || n == N - 1) {
            return 1;
        }

        return countPaths(m + 1, n, M, N)     // move down
            + countPaths(m, n + 1, M, N);     // move right
    }

    public static void main(String[] args)
    {
        // `M × N` matrix
        int M = 3;
        int N = 3;

        int k = countPaths(0, 0, M, N);
        System.out.println("The total number of paths is " + k);
    }
}
```

The time complexity of the proposed solution is exponential since it exhibits overlapping subproblems,

```
class CountAllPath2
{
    // Bottom-up function to count all paths from the first cell (0, 0)
    // to the last cell (M-1, N-1) in a given `M × N` rectangular grid
    public static int countPaths(int m, int n)
    {
        // `T[i][j]` stores the number of paths from cell (0, 0) to cell (i, j)
        int[][] T = new int[m][n];

        // There is only one way to reach any cell in the first column, i.e.,
        // to move down
        for (int i = 0; i < m; i++) {
            T[i][0] = 1;
        }

        // There is only one way to reach any cell in the first row, i.e.,
        // to move right
        for (int j = 0; j < n; j++) {
            T[0][j] = 1;
        }

        // fill `T[][]` in a bottom-up manner
        for (int i = 1; i < m; i++)
        {
            for (int j = 1; j < n; j++) {
                T[i][j] = T[i-1][j] + T[i][j-1];
            }
        }

        // last cell of `T[][]` stores the count of paths from cell (0, 0) to
        // cell (i, j)
        return T[m-1][n-1];
    }

    public static void main(String[] args)
    {
        // `M × N` matrix
        int M = 3;
        int N = 3;

        int k = countPaths(M, N);
        System.out.println("The total number of paths is " + k);
    }
}
```

The time complexity of the proposed solution is O(M × N) for an `M × N` matrix. The auxiliary space required by the program is O(M × N). The space complexity of the solution can be improved up to O(N) as we are only reading data of the previous row

for filling the current row. Following is the space-optimized solution using only a single array:

```
class CountAllPath3
{
    // Bottom-up space-efficient function to count all paths from the first
    // cell (0, 0) to the last cell (M-1, N-1) in a given `M × N` rectangular grid
    public static int countPaths(int m, int n)
    {
        int[] T = new int[n];
        T[0] = 1;

        // fill `T[][]` in a bottom-up manner
        for (int i = 0; i < m; i++)
        {
            for (int j = 1; j < n; j++) {
                T[j] += T[j - 1];
            }
        }

        // return the last cell
        return T[n-1];
    }

    public static void main(String[] args)
    {
        // `M × N` matrix
        int M = 3;
        int N = 3;

        int k = countPaths(M, N);
        System.out.print("The total number of paths is " + k);
    }
}
```

We can also find the number of paths from the first cell `(0, 0)` to the last cell `(M-1, N-1)` using a direct formula. To reach the last cell, we have to take `m-1` steps to the right and `n-1` steps down. So, the total steps are `(m-1)+(n-1) = m+n-2`. Out of these `(m+n-2)` steps, any `n-1` steps can be down, and any `m-1` steps can be towards the right.

So, the total number of ways are

$$(m+n-2)C_{(n-1)} \text{ or } (m+n-2)C_{(m-1)}$$

Find maximum profit that can be earned by conditionally selling stocks

Given a list containing future price predictions of two different stocks for the next n–days, find the maximum profit earned by selling the stocks with a constraint that the second stock can be sold, only if no transaction happened on the previous day for any of the stock.

Assume that the given prices are relative to the buying price. Each day, we can either sell a single share of the first stock or a single share of the second stock or sell nothing.

```
Input:

Day    Price(x)  Price(y)
1        5        8
2        3        4
3        4        3
4        6        5
5        3        10

Output: Maximum profit earned is 25


Explanation:

Day 1: Sell stock y at a price of 8
Day 2: Sell stock x at a price of 3
Day 3: Sell stock x at a price of 4
Day 4: Don’t sell anything
Day 5: Sell stock y at a price of 10
```

The idea is to use recursion to solve this problem in a top-down manner. To find the maximum profit till `n'th` day, there are two possible options:

1. Sell the first stock on the `n'th` day, and then recursively calculate the maximum profit till the `(n-1)'th` day.
2. Sell the second stock on the `n'th` day, skip the `(n-1)'th` day, and then recursively calculate the maximum profit till the `(n-2)'th` day.

```
class SellStock1
{
    // Recursive function to find the maximum profit that can be earned by selling
    // stocks. Here, arrays `x[0…n]` and `y[0…n]` contains the two different stocks'
    // future price predictions for the next n–days.
    public static int findMaxProfit(int[] x, int[] y, int n)
    {
```

```
        // base case
        if (n < 0) {
            return 0;
        }

        // store the maximum profit gained
        int profit = 0;

        // sell the first stock on the n'th day, and recur for the (n-1)'th day
        profit = Integer.max(profit, x[n] + findMaxProfit(x, y, n - 1));

        // sell the second stock on the n'th day, and recur for the (n-2)'th day
        // (no transaction can be made on the (n-1)'th day)
        profit = Integer.max(profit, y[n] + findMaxProfit(x, y, n - 2));

        // return the maximum profit
        return profit;
    }

    public static void main(String[] args)
    {
        int[] x = { 5, 3, 4, 6, 3 };
        int[] y = { 8, 4, 3, 5, 10 };

        System.out.println("The maximum profit earned is " +
                findMaxProfit(x, y, x.length - 1));
    }
}
```

The time complexity of the above solution is exponential since the code is doing a lot of redundant work. For example, for solving the problem `T(n)`, a recursive call is made to find the maximum profit for both `T(n-1)` and `T(n-2)`. However, finding the maximum profit for `T(n-1)` also requires finding the maximum profit for `T(n-2)`. As the recursion grows deeper, more and more of this type of unnecessary repetition occurs.

his repetition can be avoided with dynamic programming. The idea is to cache each time any subproblem `T(i)` is computed. If we are ever asked to compute it again, return the cached result and not recompute it.

The bottom-up approach computes the maximum earnings of all days, using the already computed profit of past days. The maximum earnings `T[i]` possible till the `i'th` day can be calculated by taking the maximum of the below actions:

1. Sell the first stock on the `i'th` day, and add the profit to the maximum earnings till the `(i-1)'th` day.
2. Sell the second stock on the `i'th` day, and add the profit to the maximum earnings till the `(i-2)'th` day.

```
class SellStock2
{
    // Function to find the maximum earnings that can be earned by selling the stocks.
    // Here, arrays `x[0…n-1]` and `y[0…n-1]` contains the two different stocks'
    // future price predictions for the next n–days.
    public static int findMaxProfit(int[] x, int[] y)
    {
        // base case
        if (x.length == 0) {
            return 0;
        }

        // create an auxiliary array `T[]` to save solutions to the subproblems.
        // Here, `T[i]` stores the maximum earnings till day `i`.
        int[] T = new int[x.length + 1];

        // Base cases
        T[0] = 0;
        T[1] = Integer.max(x[0], y[0]);

        // Fill the auxiliary array `T[]` in a bottom-up manner
        for (int i = 2; i <= x.length; i++) {
            T[i] = Integer.max(x[i - 1] + T[i - 1], y[i - 1] + T[i - 2]);
        }

        // `T[n]` stores the maximum earnings till day `n`
        return T[x.length];
    }

    public static void main(String[] args)
    {
        int[] x = { 5, 3, 4, 6, 3 };
        int[] y = { 8, 4, 3, 5, 10 };

        System.out.println("The maximum profit earned is " + findMaxProfit(x, y));
    }
}
```

The time complexity of the above solution is O(n), where `n` is the total number of given days. The auxiliary space required by the program is O(n) for the auxiliary array and requires no recursion.

Note that the value of `T(n)` is calculated in constant time using the values of `T(n-1)` and `T(n-2)`. We can eliminate the need for an auxiliary array by keeping track of only the last two subproblems' solutions.

```
class SellStock3
{
    // Function to find the maximum profit that can be earned by selling the stocks.
    // Here, arrays `x[0…n-1]` and `y[0…n-1]` contains the two different stocks'
```

```
    // future price predictions for the next n-days.
    public static int findMaxProfit(int[] x, int[] y)
    {
        // base case
        if (x.length == 0) {
            return 0;
        }

        int prev_of_prev = 0;
        int prev = Integer.max(x[0], y[0]);

        // Find the maximum profit in a bottom-up manner
        for (int i = 1; i < x.length; i++)
        {
            int curr = Integer.max(x[i] + prev, y[i] + prev_of_prev);
            prev_of_prev = prev;
            prev = curr;
        }

        // `prev` stores the maximum profit gained till day `n`
        return prev;
    }

    public static void main(String[] args)
    {
        int[] x = { 5, 3, 4, 6, 3 };
        int[] y = { 8, 4, 3, 5, 10 };

        System.out.println("The maximum profit earned is " + findMaxProfit(x, y));
    }
}
```

The time complexity of the above solution is O(n) and requires O(n) extra space, where `n` is the total number of given days.

Longest Alternating Subarray Problem

Given an array containing positive and negative elements, find a subarray with alternating positive and negative elements, and in which the subarray is as long as possible.

The Longest Alternating Subarray problem differs from the problem of finding the Longest Alternating subsequence. Unlike a subsequence, a subarray is required to occupy consecutive positions within the original array.

For example, consider array `{ 1, -2, 6, 4, -3, 2, -4, -3 }`. The longest alternating subarray is `{ 4, -3, 2, -4 }`. Note that the longest alternating subarray might not be unique.

We can easily solve this problem in linear time using similar logic as Kadane’s algorithm.The idea is to maintain the longest alternating subarray “ending” at each given array index. This subarray can be a single element (if the previous element has the same sign) or consists of one more element than the longest alternating subarray ending at the previous index (if the previous element has an opposite sign).

```
import java.util.Arrays;

class SubArray
{
    // Function to find the length of the longest subarray with alternating
    // positive and negative elements
    public static void findLongestSubarray(int[] nums)
    {
        // base case
        if (nums == null || nums.length == 0) {
            return;
        }

        // stores length of longest alternating subarray found so far
        int maxLen = 1;

        // stores ending index of longest alternating subarray found so far
        int endIndex = 0;

        // stores length of longest alternating subarray ending at the current position
        int currLen = 1;

        // traverse the given array starting from the second index
        for (int i = 1; i < nums.length; i++)
        {
            // if the current element has an opposite sign than the previous element
            if (nums[i] * nums[i - 1] < 0)
            {
                // include the current element in the longest alternating subarray
                // ending at the previous index
                currLen++;

                // update result if the current subarray length is found to be greater
                if (currLen > maxLen)
                {
                    maxLen = currLen;
                    endIndex = i;
                }
            }
            // reset length if the current element has the same sign as the previous
            // element
            else {
                currLen = 1;
            }
        }
```

```
        int[] subarray = Arrays.copyOfRange(nums, (endIndex - maxLen + 1), endIndex + 1);
        System.out.println("The longest alternating subarray is "
                           + Arrays.toString(subarray));
    }

    public static void main (String[] args)
    {
        int[] nums = { 1, -2, 6, 4, -3, 2, -4, -3 };

        findLongestSubarray(nums);
    }
}
```

The time complexity of the above solution is O(n) and requires O(n) extra space, where `n` is the size of the input.