SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

CERTIFICATE

This is to certify that Design and Analysis Algorithm Lab Mini Project entitled “**Maximum profit by buying and selling a share at most twice**” Submitted by “**Utkarsh Saboo” (RA2011003011248)**, “**Divyansh Gupta (RA2011003011252)**, and “**Purva Atul Tarale” (RA2011003011253)** for the partial fulfilment of the requirement for Semester IV Subject of Design and Analysis Algorithm Lab to the SRM Institute of Science and Technology, is a Bonafide work carried out during Semester IV in Academic Year 2021-2022.

Mrs. R. Anita
(Subject In-charge)

Declaration

We declare that this written submission represents our ideas in our own words and where others' ideas or words have been included, we have adequately cited and referenced the original sources. We also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. We understand that any violation of the above will be cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

ACKNOWLEDGEMENT

We would like to thank **Mrs. R Anita** (Subject In-charge) who has been a great inspiration and who have provided sufficient background knowledge and understanding of this subject.

Our humble prostration goes to her, for providing all the necessary resources and environment, which have aided me to complete this project successfully.

TABLE OF CONTENT

CONTRIBUTION

S.No.	Registration Number	Name	Role
1	RA2011003011252	DIVYANSH GUPTA	Representative
2	RA2011003011248	UTKARSH SABOO	Team Member
3	RA2011003011253	PURVA ATUL TARALE	Team Member

PROBLEM DEFINITION

In daily share trading, a buyer buys shares in the morning and sells them on the same day. If the trader is allowed to make at most 2 transactions in a day, whereas the second transaction can only start after the first one is complete (Buy->sell->Buy->sell). Given stock prices throughout the day, find out the maximum profit that a share trader could have made.

Examples:

Input: price[] = {10, 22, 5, 75, 65, 80}

Output: 87

Trader earns 87 as sum of 12, 75

Buy at 10, sell at 22,

Buy at 5 and sell at 80

Input: price[] = {2, 30, 15, 10, 8, 25, 80}

Output: 100

Trader earns 100 as sum of 28 and 72

Buy at price 2, sell at 30, buy at 8 and sell at 80

Input: price[] = {100, 30, 15, 10, 8, 25, 80};

Output: 72

Buy at price 8 and sell at 80.

Input: price[] = {90, 80, 70, 60, 50}

Output: 0

Not possible to earn.

DESIGN TECHNIQUE – DYNAMIC PROGRAMMING

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler subproblems, solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Let's understand this approach through an example.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,…

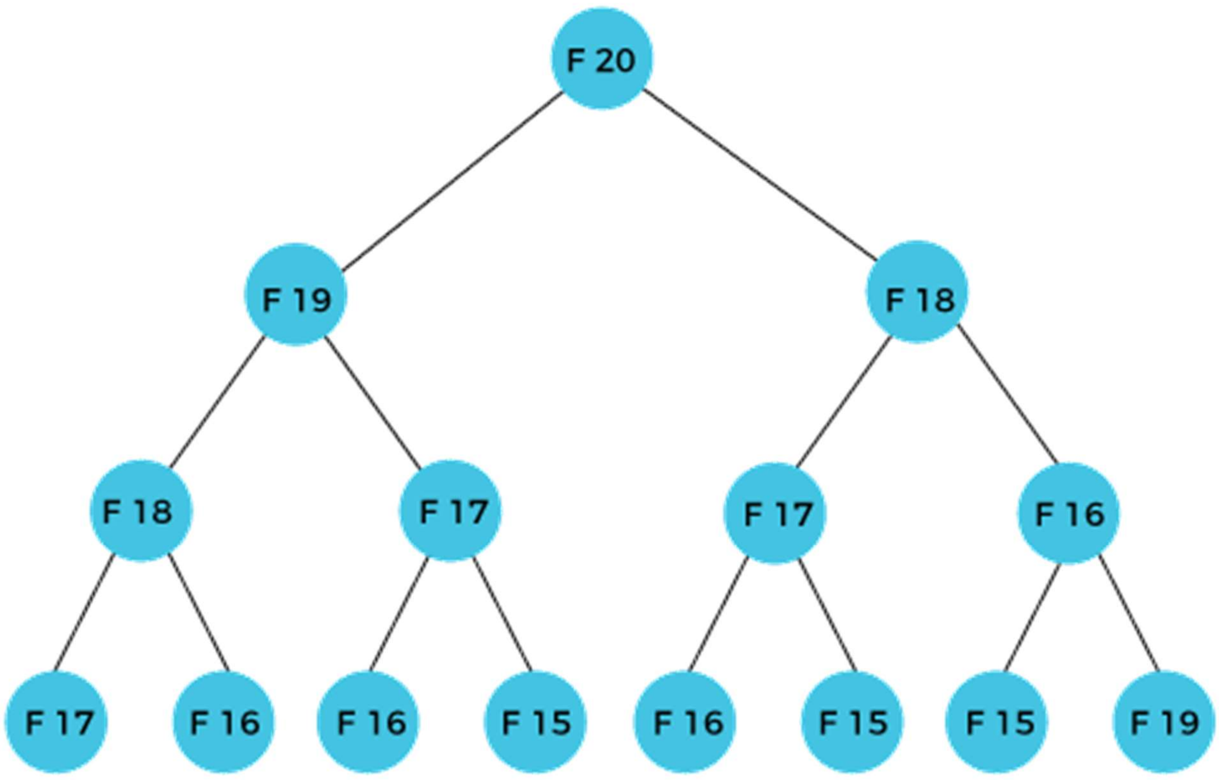
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1)**, which is equal to 1.

How can we calculate F(20)?

The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



As we can observe in the above figure that F(20) is calculated as the sum of F(19) and F(18). In the dynamic programming approach, we try to divide the problem into the similar subproblems. We are following this approach in the above case where F(20) into the similar subproblems, i.e., F(19) and F(18). If we recap the definition of dynamic programming that it says the similar subproblem should not be computed more than once. Still, in the above case, the subproblem is calculated twice. In the above example, F(18) is calculated two times; similarly, F(17) is also calculated twice. However, this technique is quite useful as it solves the similar subproblems, but we need to be cautious while storing the results because we are not particular about storing the result that we have computed once, then it can lead to a wastage of resources.

In the above example, if we calculate the F(18) in the right subtree, then it leads to the tremendous usage of resources and decreases the overall performance.

The solution to the above problem is to save the computed results in an array. First, we calculate F(16) and F(17) and save their values in an array. The F(18) is calculated by summing the values of F(17) and F(16), which are already saved in an array. The computed value of F(18) is saved in an array. The value of F(19) is calculated using the sum of F(18), and F(17), and their values are already saved in an array. The computed value of F(19) is stored in an array. The value of F(20) can be calculated by adding the values of F(19) and F(18), and the values of both F(19) and F(18) are stored in an array. The final computed value of F(20) is stored in an array.

How does the dynamic programming approach work?

The following are the steps that the dynamic programming follows:

It breaks down the complex problem into simpler subproblems.
It finds the optimal solution to these sub-problems.
It stores the results of subproblems (memorization). The process of storing the results of subproblems is known as memorization.
It reuses them so that same sub-problem is calculated more than once.
Finally, calculate the result of the complex problem.
The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

Those problems that are having overlapping subproblems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the subproblems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

Approaches of dynamic programming

There are two approaches to dynamic programming:

Top-down approach

Bottom-up approach

Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Advantages

It is very easy to understand and implement.

It solves the subproblems only when it is required.

It is easy to debug.

Disadvantages

It uses the recursion technique that occupies more memory in the call stack. Sometimes when the recursion is too deep, the stack overflow condition will occur.

It occupies more memory that degrades the overall performance.

Explanation with Diagram

Let's understand dynamic programming through an example.

```
int fib(int n)
{
  if(n<0)
  error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
}
```

In the above code, we have used the recursive approach to find out the Fibonacci series. When the value of 'n' increases, the function calls will also increase, and computations will also increase. In this case, the time complexity increases exponentially, and it becomes 2n.

One solution to this problem is to use the dynamic programming approach. Rather than generating the recursive tree again and again, we can reuse the previously calculated value. If we use the dynamic programming approach, then the time complexity would be O(n).

When we apply the dynamic programming approach in the implementation of the Fibonacci series, then the code would look like:

```
static int count = 0;
int fib(int n)
{
if(memo[n]!= NULL)
return memo[n];
count++;
```

```
  if(n<0)
  error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
memo[n] = sum;
}
```

In the above code, we have used the memorization technique in which we store the results in an array to reuse the values. This is also known as a top-down approach in which we move from the top and break the problem into sub-problems.

Bottom-Up approach

The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

There are two ways of applying dynamic programming:

Top-Down
Bottom-Up

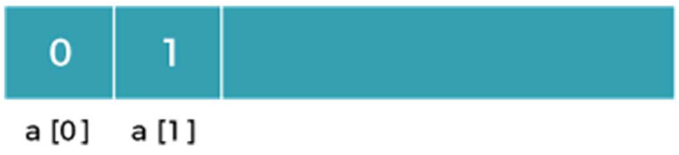
The bottom-up is the approach used to avoid the recursion, thus saving the memory space. The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward. In the bottom-up approach, we start from the base case to find the answer for the end. As we know, the base cases in the Fibonacci series are 0 and 1. Since the bottom approach starts from the base cases, so we will start from 0 and 1.

Key points

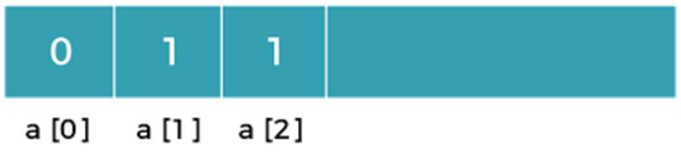
We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
We use for loop to iterate over the sub-problems.
The bottom-up approach is also known as the tabulation or table filling method.

Let's understand through an example.

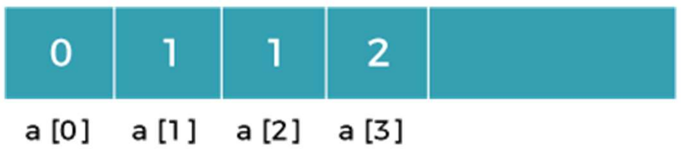
Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:



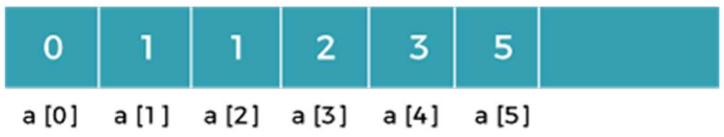
Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value of a[2] shown as below:



The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:



The value of a[4] will be calculated by adding a[2] and a[3], and it becomes 3 shown as below:



The value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:

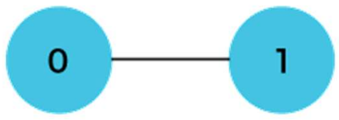
The code for implementing the Fibonacci series using the bottom-up approach is given below:

```
int fib(int n)
{
   int A[];
   A[0] = 0, A[1] = 1;
   for( i=2; i<=n; i++)
   {
       A[i] = A[i-1] + A[i-2]
   }
   return A[n];
}
```

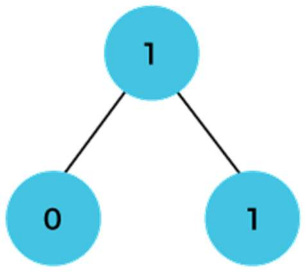
In the above code, base cases are 0 and 1 and then we have used for loop to find other values of Fibonacci series.

Let's understand through the diagrammatic representation.

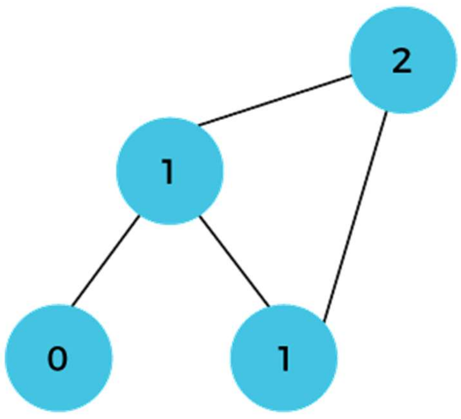
Initially, the first two values, i.e., 0 and 1 can be represented as:



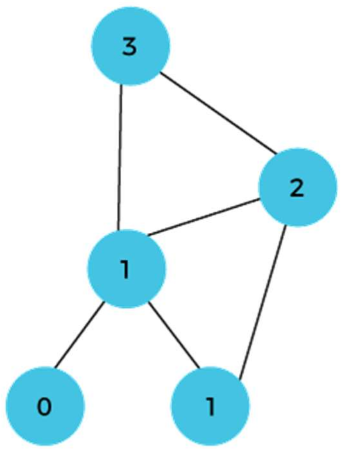
When i=2 then the values 0 and 1 are added shown as below:



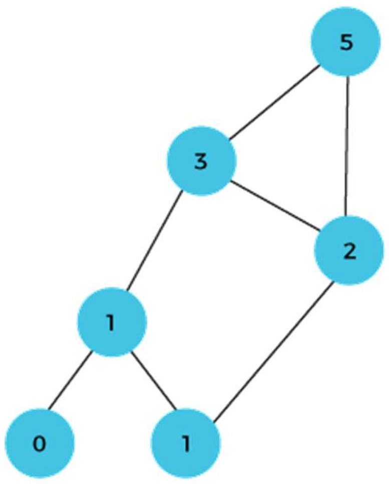
When i=3 then the values 1and 1 are added shown as below:



When i=4 then the values 2 and 1 are added shown as below:



When i=5, then the values 3 and 2 are added shown as below:



In the above case, we are starting from the bottom and reaching to the top.

ALGORITHM FOR THE PROBLEM

A Simple Solution is to consider every index ‘i’ and do the following

Max profit with at most two transactions =
MAX {max profit with one transaction and subarray price[0..i] +
max profit with one transaction and subarray price[i+1...n-1] }
i varies from 0 to n-1.

Maximum possible using one transaction can be calculated using the following O(n) algorithm
The maximum difference between two elements such that the larger element appears after the smaller number
The time complexity of the above simple solution is O(n2).

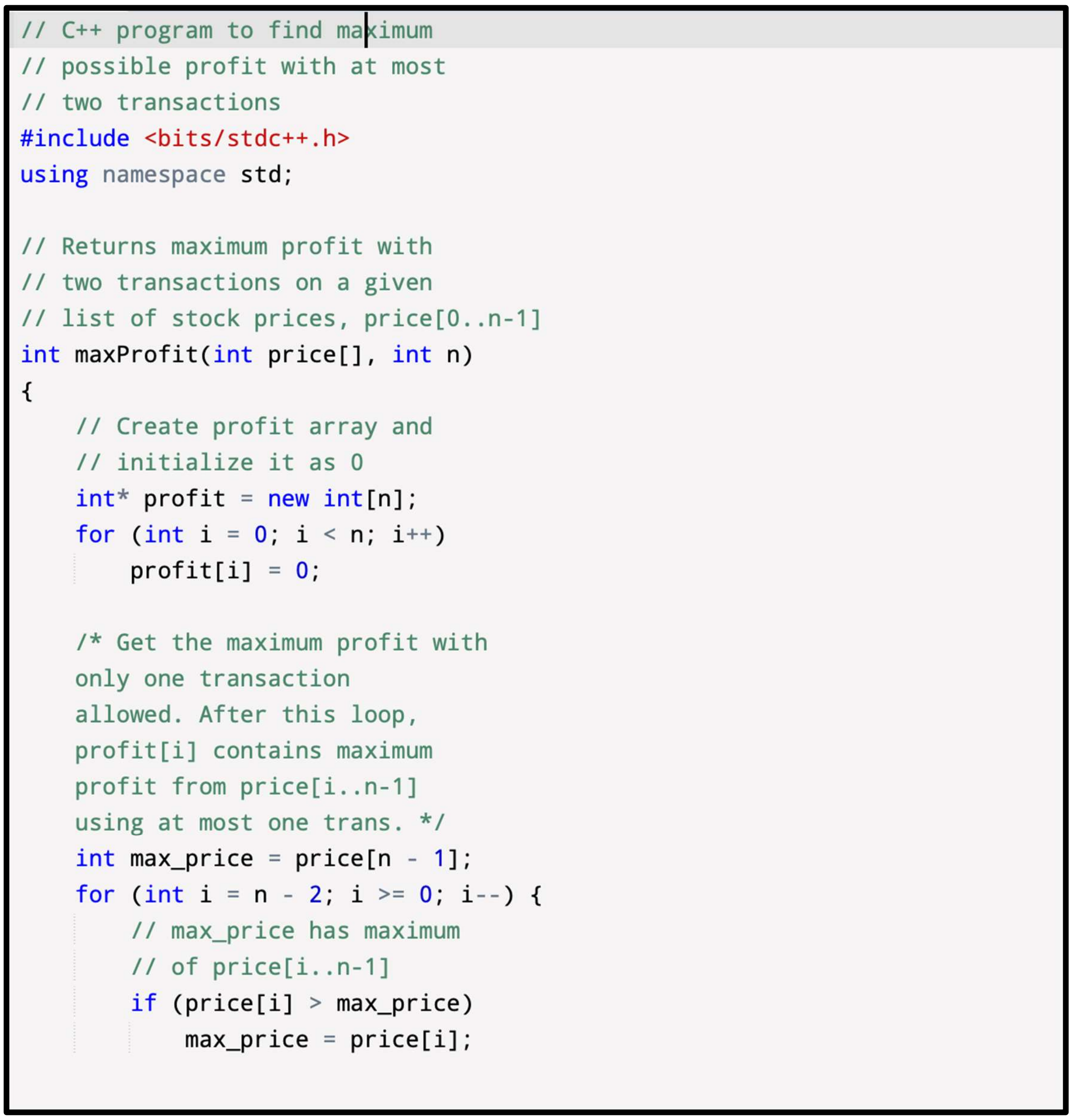
We can do this O(n) using the following Efficient Solution. The idea is to store the maximum possible profit of every subarray and solve the problem in the following two phases.

1) Create a table profit[0...n-1] and initialize all values in it 0.
2) Traverse price[] from right to left and update profit[i] such that profit[i] stores maximum profit achievable from one transaction in subarray price[i..n-1]
3) Traverse price[] from left to right and update profit[i] such that profit[i] stores maximum profit such that profit[i] contains maximum achievable profit from two transactions in subarray price[0...i].
4) Return profit[n-1]

To do step 2, we need to keep track of the maximum price from right to left side, and to do step 3, we need to keep track of the minimum price from left to right. Why we traverse in reverse directions? The idea is to save space, in the third step, we use the same array for both purposes, maximum with 1 transaction and maximum with 2 transactions. After iteration i, the array profit[0...i] contains the maximum profit with 2 transactions, and profit[i+1...n-1] contains profit with two transactions.

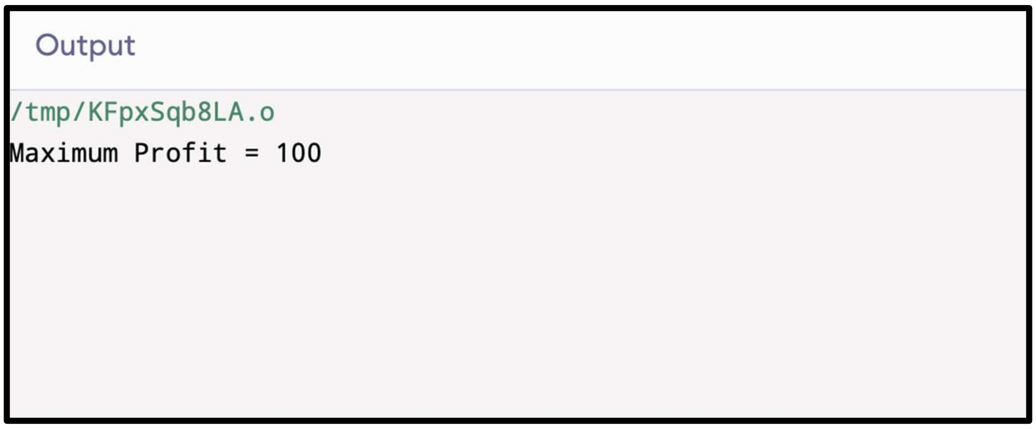
Implementation of the Idea (Code of the Project)

Below are the implementations of the above idea.



```cpp
// C++ program to find maximum
// possible profit with at most
// two transactions
#include <bits/stdc++.h>
using namespace std;

// Returns maximum profit with
// two transactions on a given
// list of stock prices, price[0..n-1]
int maxProfit(int price[], int n)
{
    // Create profit array and
    // initialize it as 0
    int* profit = new int[n];
    for (int i = 0; i < n; i++)
        profit[i] = 0;

    /* Get the maximum profit with
    only one transaction
    allowed. After this loop,
    profit[i] contains maximum
    profit from price[i..n-1]
    using at most one trans. */
    int max_price = price[n - 1];
    for (int i = n - 2; i >= 0; i--) {
        // max_price has maximum
        // of price[i..n-1]
        if (price[i] > max_price)
            max_price = price[i];
        // we can get profit[i] by taking maximum of:
        // a) previous maximum, i.e., profit[i+1]
        // b) profit by buying at price[i] and selling at
        // max_price
        profit[i]
            = max(profit[i + 1], max_price - price[i]);
    }

    /* Get the maximum profit with two transactions allowed
    After this loop, profit[n-1] contains the result */
    int min_price = price[0];
    for (int i = 1; i < n; i++) {
        // min_price is minimum price in price[0..i]
        if (price[i] < min_price)
            min_price = price[i];

        // Maximum profit is maximum of:
        // a) previous maximum, i.e., profit[i-1]
        // b) (Buy, Sell) at (min_price, price[i]) and add
        // profit of other trans. stored in profit[i]
        profit[i] = max(profit[i - 1],
                        profit[i] + (price[i] - min_price));
    }
    int result = profit[n - 1];

    delete[] profit; // To avoid memory leak

    return result;
```

```
// Driver code
int main()
{
    int price[] = { 2, 30, 15, 10, 8, 25, 80 };
    int n = sizeof(price) / sizeof(price[0]);
    cout << "Maximum Profit = " << maxProfit(price, n);
    return 0;
}
```



```
/tmp/KFpxSqb8LA.o
Maximum Profit = 100
```

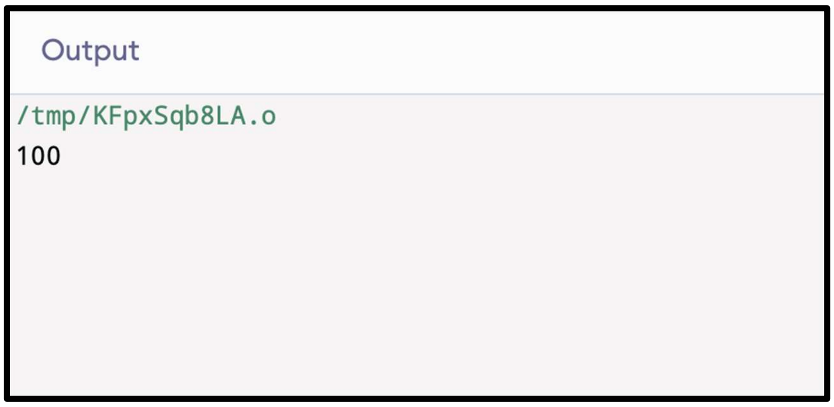
The time complexity of the above solution is O(n).

Algorithmic Paradigm: Dynamic Programming

Another approach:

Initialize four variables for taking care of the first buy, first sell, second buy, second sell. Set first buy and second buy as INT_MIN and first and second sell as 0. This is to ensure to get profit from transactions. Iterate through the array and return the second sell as it will store maximum profit.

```cpp
#include <iostream>
#include<climits>
using namespace std;

int maxtwobuysell(int arr[],int size) {
    int first_buy = INT_MIN;
    int first_sell = 0;
    int second_buy = INT_MIN;
    int second_sell = 0;

    for(int i=0;i<size;i++) {

        first_buy = max(first_buy,-arr[i]);//we set prices to negative, so
            the calculation of profit will be convenient
        first_sell = max(first_sell,first_buy+arr[i]);
        second_buy = max(second_buy,first_sell-arr[i]);//we can buy the
            second only after first is sold
        second_sell = max(second_sell,second_buy+arr[i]);

    }
    return second_sell;
}

int main() {

    int arr[] = {2, 30, 15, 10, 8, 25, 80};
    int size = sizeof(arr)/sizeof(arr[0]);
    cout<<maxtwobuysell(arr,size);
    return 0;
}
```


Output

```
/tmp/KFpxSqb8LA.o
100
```

Time Complexity: O(N)

Auxiliary Space: O(1)

DRY RUN

Sample Input:

n = 7

prices = {2,30,15,10,8,25,80}

Dry Run:

Main function:

n = 7

prices = {2,30,15,10,8,25,80}

Print(maxprofit(prices,n))

MaxProfit function:

profit = {0,0,0,0,0,0,0}

max_price = 80

Running first loop:

1) i=5

if (price[i] < min_price) -> False

profit = {0,0,0,0,0,55,0}

2) i=4

if (price[i] < min_price) -> False

profit = {0,0,0,0,72,55,0}

3) i=3

if (price[i] < min_price) -> False

profit = {0,0,0,72,72,55,0}

4) i=2

if (price[i] < min_price) -> False

profit = {0,0,72,72,72,55,0}

5) i=1

if (price[i] < min_price) -> False

profit = {0,72,72,72,72,55,0}

6) i=0

if (price[i] < min_price) -> False

profit = {78,72,72,72,72,55,0}

min_price = 2

Running second loop:

1) i=1

   profit = {78,100,72,72,72,55,0}

2) i=2

   profit = {78,100,85,72,72,55,0}

3) i=3

   profit = {78,100,85,80,72,55,0}

4) i=4

   profit = {78,100,85,80,78,55,0}

5) i=5

   profit = {78,100,85,80,78,92,0}

6) i=6

   profit = {78,100,85,80,78,92,100}

return profit[-1] = 100

COMPLEXITY ANALYSIS

Time Complexity

In Dynamic programming problems, Time Complexity is **the number of unique states/subproblems * time taken per state**.

In this problem, for a given n, there are **n** unique states/subproblems. For convenience, each state is said to be solved in a constant time. Hence the time complexity is **O(n * 1).**

This can be easily cross verified by the for loop we used in the bottom-up approach. We see that we use only one for loop to solve the problem. Hence the time complexity is **O (n) or linear.**

This is the power of dynamic programming. It allows such complex problems to be solved efficiently.

Space Complexity

We use one array called cache to store the results of n states. Hence the size of the array is n. Therefore, the space complexity is **O(n)**.

DP as Space-Time tradeoff

Dynamic programming makes use of space to solve a problem faster. In this problem, we are using **O(n)** space to solve the problem in **O(n)** time. Hence, we trade space for speed/time. Therefore, it is aptly called the **Space-Time tradeoff.**

CONCLUSION

In conclusion, dynamic programming is an exceptional variant of recursion that compensates for its drawbacks. However, DP can sometimes be hard to understand and grasp, making it a popular candidate for coding interviews.

Whether you are a student preparing for a job or a professional, understanding how DP works can benefit you. Head over to these websites and resources to better understand DP and solve problems on your own because that is the best way to learn something new.

RESOURCES

- GeekforGeeks - http://www.geeksforgeeks.org
- JavaPoint - https://www.javatpoint.com