IEEEXtreme Türkiye Camp 24' Program

Day 2

1 Search Algorithms

It may be necessary to determine if an array or solution set contains a specific data, and we call this finding proccess **searching**. In this article, three most common search algorithms will be discussed: linear search, binary search, and ternary search.

This visualization may help you understand how the search algorithms work: Link.

1.1 Linear Search

Simplest search algorithm is *linear search*, also know as *sequential search*. In this technique, all elements in the collection of the data is checked one by one, if any element matches, algorithm returns the index; otherwise, it returns -1.

ity is $O(N)$

0	1	2	3	4	5	6	7	8	9	Search Key: 70
7	29	48	53	63	70	76	89	94	96	7 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	29 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	48 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	53 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	63 < 70, move cursor
7	29	48	53	63	70	76	89	94	96	70 = 70, return 5

Figure 1: Example for linear search

```
int linearSearch(int *array, int size, int key){
    for (int i=0; i < size; i++)
        if (array[i] == key)
            return i;
    return -1;
}
```

1.2 Binary Search

We know linear search is quite a slow algorithm because it compares each element of the set with search key, and there is a high-speed searching technique for **sorted** data instead of linear search, which is **binary search**. After each comparison, the algorithm eliminates half of the data using the sorting property.

We can also use binary search on increasing functions in the same way.

Procedure:

- Compare the key with the middle element of the array,
- If it is a match, return the index of middle.
- If the key is bigger than the middle, it means that the key must be in the right side of the middle. We can eliminate the left side.
- If the key is smaller, it should be on the left side. The right side can be ignored.

Complexity:

$$\begin{aligned} T(N) &= T(N/2) + O(1) \\ T(N) &= O(logN) \end{aligned}$$

L				M					R	Search Key: 70
7	29	48	53	63	70	76	89	94	96	63 < 70, shift L

					L		M		R	Search Key: 70
7	29	48	53	63	70	76	89	94	96	89 > 70, shift R

					L M	R				Search Key: 70
7	29	48	53	63	70	76	89	94	96	70 = 70, return 5

Figure 2: Example for binary search

```
int binarySearch(int *array, int size, int key){
    int left = 0, right = size, mid;

    while (left < right){
        mid = (left + right) / 2;

        if (array[mid] >= key)
            right = mid;
        else
            left = mid + 1;
    }
    return array[left] == key ? left : -1 ;
}
```

1.3 Ternary Search

Suppose that we have a unimodal function, $f(x)$, on an interval $[l, r]$, and we are asked to find the local minimum or the local maximum value of the function according to the behavior of it.

There are two types of unimodal functions:

1. The function, $f(x)$ strictly increases for $x \leq m$, reaches a global maximum at $x = m$, and then strictly decreases for $m \leq x$. There are no other local maxima.

2. The function, $f(x)$ strictly decreases for $x \leq m$, reaches a global minimum at $x = m$, and then strictly increases for $m \leq x$. There are no other local minima.

In this document, we will implement the first type of unimodal function, and the second one can be solved using the same logic.

Procedure:

1. Choose any two points m_1, and m_2 on the interval $[l, r]$, where $l < m_1 < m_2 < r$.

2. If $f(m_1) < f(m_2)$, it means the maxima should be in the interval $[m_1, r]$, so we can ignore the interval $[l, m_1]$, move l to m_1

3. Otherwise, $f(m_1) \geq f(m_2)$, the maxima have to be in the interval $[l, m_2]$, move r to m_2

4. If $r - l < \epsilon$, where ϵ is a negligible value, stop the algorithm, return l. Otherwise turn to the step 1.

m_1 and m_2 can be selected by $m_1 = l + (r - l)/3$ and $m_2 = r - (r - l)/3$ to avoid increasing the time complexity.

Complexity:

$$\begin{aligned} T(N) &= T(2 \cdot N/3) + O(1) \\ T(N) &= O(logN) \end{aligned}$$

```
double f(double x);

double ternarySearch(double left, double right, double eps=1e-7){
    while (right - left > eps){
        double mid1 = left + (right - left) / 3;
        double mid2 = right - (right - left) / 3;

        if (f(mid1) < f(mid2))
            left = mid1;
        else
            right = mid2;
    }
    return f(left);
}
```

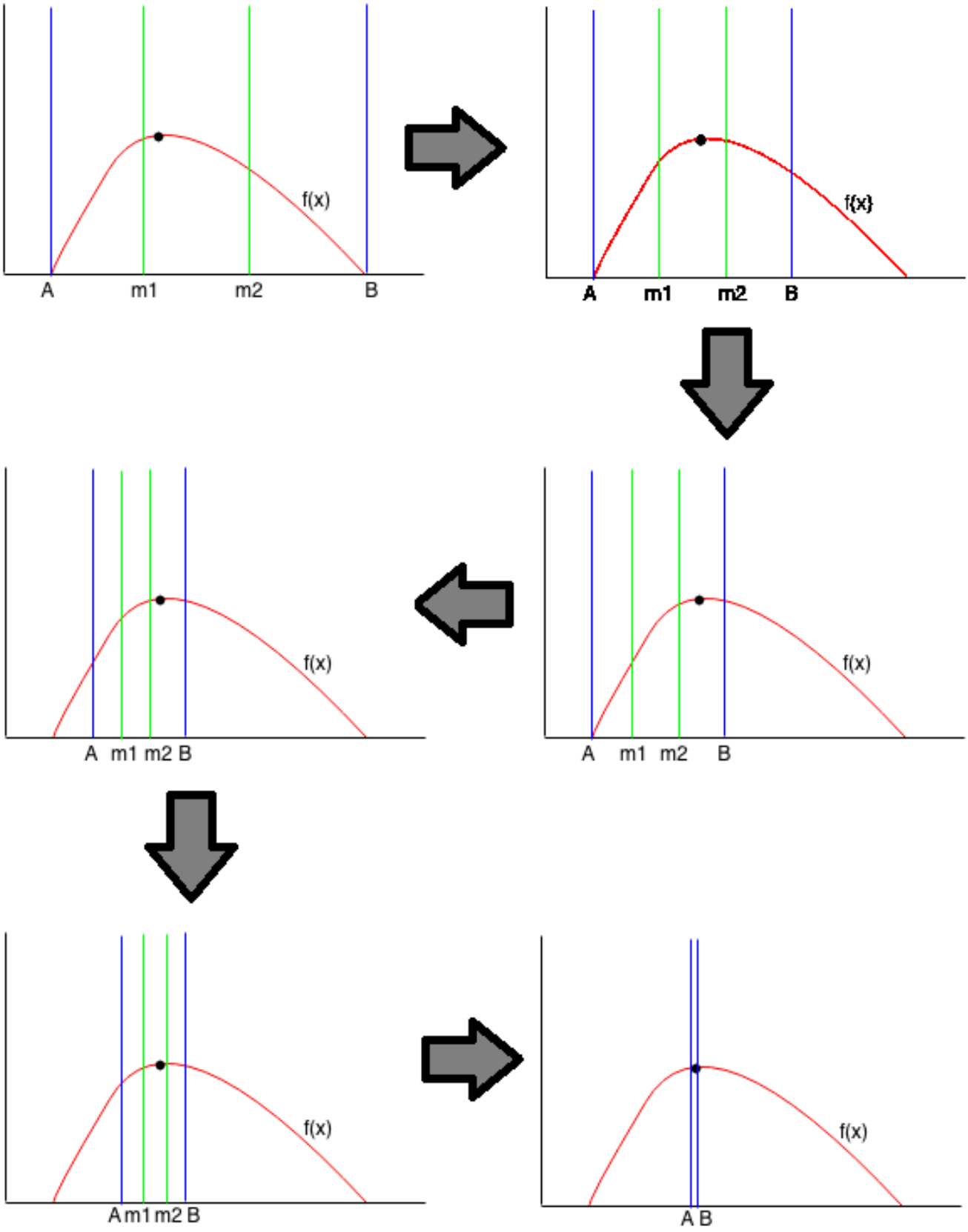

Figure 3: Example for ternary search

Introduction

Next section is about the `Number Theory`. It will be quite a generous introduction to the questions that are related to Mathematics.

Mathematics is quite essential to the programmers that want to improve themselves in the topic of competitive programming. We are going to use these topics from graph theory to the subject of strings. Therefore, a strong understanding of mathematics is fundamental.

2 Number Theory

Number theory is a study for the positive natural numbers. Numbers are split into several groups. Number theory is related to the connection between these different groups of numbers [1].

- **Even** 2, 4, 6, 8, 10, 12, ...
- **Cube** 1, 8, 27, 64, ...
- **Fibonacci** 1, 1, 2, 3, 5, 8, 13, ...

It has a long history of development. The first tablet ever found by scientists was about the Pythagorean triples. The tablet was created in 1800 BC by the Mesopotamian people. Ancient Greek, China, and Islamic states have a critical effect on the growth of number theory [2].

2.1 Primality Test

If a number can only be evenly divided by `itself` and `one`, we call this number prime.

The first ten prime numbers are in the following line.

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, ...

The main question that comes to our minds is how to find out if the given number is prime or not. Let us see a function that returns if the given number is prime or not.

We are going to test the number of n as **179424673** for each prime number algorithm. It is a very large prime number. It allows us to see the run-time differences between the algorithms.

2.1.1 Naive Approach

`Time Complexity:` $O(n)$

The first thing that pops in our minds is iterating from *2* to *n-1* and check if the given n is evenly divisible by the current number. This is the naive approach to testing the prime.

For example, let's say we have number **A**. Our plan in this algorithm is looping from 2 to **A-1**. For each value in the loop, we will try to divide **A** by the current value. If the current value divides **A** evenly, we can say **A** is not a prime number. We also know the divisor(factor) of **A** should be smaller than or equal to **A**. Therefore, we should find at least one divisor of **A**, if there is one.

```cpp
#include <iostream>
#include <time.h>

using namespace std;

bool isPrime(long long n) {
    // 0 and 1 are not prime numbers. Therefore, we can return false directly.
    if(n == 0 || n == 1) return false;
    // Check until n
    for(long long i = 2; i < n; i++)
        if(n%i == 0)
            return false;

    // If nothing divides n, return true.
    return true;
}


int main() {
    // Read the input
    long long n;
    scanf("%lld", &n);

    // Calculate the runtime of the isPrime function.
    clock_t tStart = clock();
    bool nIsPrime = isPrime(n);
    printf("Time taken: %.2fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);

    // Prepare the output string.
    string finalAnswer;
    if(nIsPrime)
        finalAnswer = "It is unquestionably a prime number.";
    else
        finalAnswer = "Hmm. I am not quite sure about that.";

    printf("%s\n", finalAnswer.c_str());
    return 0;
}
```

Output

- `The input is 179424673`
- `Time taken:` **`1.690920s`**
- `It is unquestionably a prime number.`

2.1.2 Optimized Naive Approach

`Time Complexity:` $O(\sqrt{n})$

Instead of iterating from 2 to *n-1*, we can stop when the current number exceeds the square root of the given number($\sqrt{n}$)

For example, we will test the number **100**. In the naive approach, we were looping up to *n-1*. However, we are checking redundant numbers. Since we are checking the primeness, we should start checking from 2.

- The first factor is 2. Since we know 2 divides 100, we do not need to check 50. (2*50 = 100)
- The second factor is 5. Since we know 5 divides 100, we do not need to check 20. (5*20 = 100)
- The third factor is 10. (10*10 = 100)
- The fourth factor is 20. However, we have already checked the complementary number of 20, which is 5. - Checking 20 is **unnecessary**
- The fourth factor is 50. However, we have already checked the complementary number of 50, which is 2. - Checking 50 is **unnecessary**

If we are going to find a divisor **C** of a number **A** which is smaller than or equal than $\sqrt{A}$. We are quite sure that there will be complementary **D** which is either bigger than or equal to $\sqrt{A}$. (**C** * **D** = **A**).Checking **D** is redundant since we have already checked its complementary.

Let's see another example, we will test the number 103. It is a proven prime number. We know that looping until $\sqrt{103} \approx 10.14889$ is enough. If we find number G that divides 103, we will then be sure about there will be a number Z that satisfies $Z = \frac{103}{G}$. Since we do not have a number that divides 103, we should mark 103 as a prime number.

```cpp
#include <iostream>
#include <time.h>

using namespace std;

bool isPrime(long long n) {
    // 0 and 1 are not prime numbers. Therefore, we can return false directly.
    if(n == 0 || n == 1) return false;
    // Check until i*i is smaller or equal then n
    for(long long i = 2; i*i <= n; i++)
        if(n%i == 0)
            return false;

    // If nothing divides n, return true.
    return true;
}


int main() {
    // Read the input
    long long n;
    scanf("%lld", &n);

    // Calculate the runtime of the isPrime function.
    clock_t tStart = clock();
    bool nIsPrime = isPrime(n);
    printf("Time taken: %.2fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);

    // Prepare the output string.
    string finalAnswer;
    if(nIsPrime)
        finalAnswer = "It is unquestionably a prime number.";
    else
        finalAnswer = "Hmm. I am not quite sure about that.";

    printf("%s\n", finalAnswer.c_str());
    return 0;
}
```

Output

- The input is 179424673
- Time taken: **0.000242s**
- It is unquestionably a prime number.

We are trying our algorithms with a large prime number. There will be no divisor in prime numbers. Therefore, we will iterate until the end of the loop. Performing the large prime number will lead us to the worst-case. If we take a number that is quite big but even such as 10^{10}, we would break our loop in $\mathbf{i = 2}$

There are some other algorithms for testing the primality of a number. For example, the link here explains the Miller Approach.

These two methods allow us to check if **n** is prime or not. It is just a number. What will happen if we want to find all positive prime numbers smaller than or equal to n?

We know that we can find the primeness in $O(\sqrt{n})$ for a number. The first thing that we can do is iterating through 1 to n and using the optimized naive method for each number.

2.2 Finding Primes up to N

In this section, we are going to discuss finding the prime numbers between 1 and n.

2.2.1 Naive Approach

`Time Complexity:` $O(n \cdot \sqrt{n})$

We know that an algorithm that works in $O(\sqrt{n})$ checks the primality of a number. In this approach, we are going to use this method. We are going to iterate up to N and for each number, we will run the algorithm. Since the optimized naive approach for finding primeness of a number gives us a correct result, this algorithm will also give us an accurate array of prime values up to N.

Numbers	2	3	4	5	6	7	...	N
Primality Test Operations	$\sqrt{2}$	$\sqrt{3}$	$\sqrt{4}$	$\sqrt{5}$	$\sqrt{6}$	$\sqrt{7}$	...	$\sqrt{N}$

We will do the $\sqrt{2} + \sqrt{3} + \sqrt{4}$ ... $+ \sqrt{N}$ processes. The square root sum is limited by the $N \cdot \sqrt{N}$. Therefore, we would get time complexity as $O(n \cdot \sqrt{n})$ [3].

```cpp
#include <iostream>
#include <time.h>
#include <vector>

using namespace std;

bool isPrime(int t) {
    // 0 and 1 are not prime numbers. Therefore, we can return false directly.
    if(t == 0 || t == 1) return false;
    // Check until i*i is smaller or equal then t
    for(int i = 2; i*i <= t; i++)
        if(t%i == 0)
            return false;

    // If nothing divides t, return true.
    return true;
}

int main() {
    printf("Enter the size of the array. (n) \n");
    int n;
    scanf("%d", &n);
    vector<bool> isPrimeArray(n+1);

    // Calculate the runtime of the isPrime function.
    clock_t tStart = clock();

    for(int i = 0;i <= n; i++)
        isPrimeArray[i] = isPrime(i);

    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
    return 0;
}
```

Output

- The input is 10000000
- Time taken: **5.865970s**

2.2.2 Sieve of Eratosthenes Approach

`Time Complexity:` $O(n \cdot \log\log n)$

The Sieve approach was developed by the Greek Mathematician Eratosthenes. It allows us to obtain the prime states of the numbers between 1 and n.

The algorithm is quite straightforward. We need to start from 1 to $\sqrt{n}$. If the current number(i) is prime, we can mark every number that's evenly divisible by i as not prime [4].

How do we have the time complexity of O($n * loglogn$)? We need to go until to $\sqrt{n}$ for deleting the numbers [5]. But why do we have $loglogn$ in the time complexity? How many processes do we do in each prime until O(n)?

For example, N = 25;

2 will delete `4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24` - ($N/2$) process

3 will delete `6, 9, 12, 15, 18, 21, 24` - ($N/3$) process

5 will delete `10, 15, 20, 25` - ($N/5$) process

...

For each prime number until N, we will do N/p_i process for each prime number(p_i). So in total, we will be doing the following processes,

$$TotalProcess = \sum_{i=1}^{N} \frac{N}{p_i}$$

We can write the following equation according to the Euler proof [6].

$$\ln\ln n = \sum_{i=1}^{n} \frac{1}{p_i}$$

Then, we can write the total process as in the following. Since $p_i < n$ covers $p_i < \sqrt{n}$, we can write `n` instead of $\sqrt{n}$.

$$TotalProcess = N \cdot \sum_{i=1}^{N} \frac{1}{p_i}$$
$$TotalProcess = N \cdot \log\log N$$

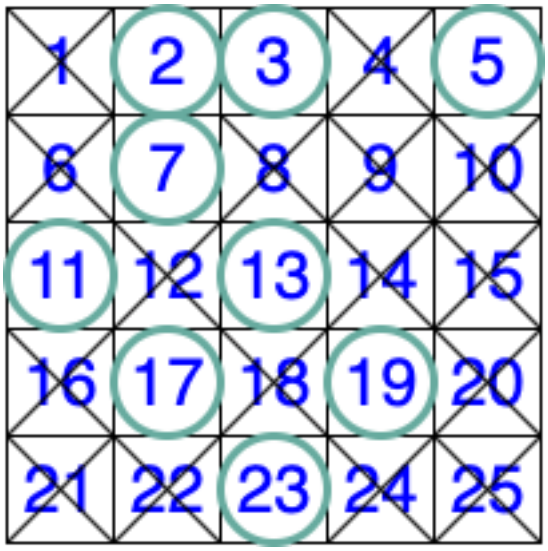

Figure 1: The visual representation of the Sieve Algorithm.

```cpp
#include <iostream>
#include <time.h>
#include <vector>

using namespace std;

void sieve(int n, vector<bool> &isPrimeArray){
    isPrimeArray[0] = false, isPrimeArray[1] = false;

    for(int i = 2; i*i < n; i++)
        if(isPrimeArray[i])
            for(int j = i*2; j < n; j += i)
                isPrimeArray[j] = false;
}

int main() {
    printf("Enter the size of the array. (n) \n");
    int n;
    scanf("%d", &n);

    // Initially, start marking every node with true.
    vector<bool> isPrimeArray(n+1, true);

    // Calculate the runtime of the function.
    clock_t tStart = clock();
    sieve(n, isPrimeArray);
    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);

    return 0;
}
```

Output

- `The input is 10000000`
- `Time taken:` **`0.245363s`**

2.3 Modular Arithmetic

Many problems requires a knowledge in modular arithmetic. Therefore, it makes modular arithmetic quite an important topic.

2.3.1 Properties of Modular Arithmetic

Congruence

- a and b concurrent $\mod n$ if the reminder of a/n and b/n are equal.

Addition

- if $a + b = c$, then $a \mod n + b \mod n \equiv c \mod n$.
- if $a \equiv b \mod n$, then $a + k \equiv b + k \mod n$ for any integer k.
- if $a \equiv b \mod n$, then $-a \equiv -b \mod n$.
- if $a \equiv b \mod n$ and $c \equiv d \mod n$, then $(a + c) \equiv (b + d) \mod n$

Multiplication

- if $a \cdot b = c$, then $a \mod n \cdot b \mod n \equiv c \mod n$.
- if $a \equiv b \mod n$, then $a \cdot k \equiv b \cdot k \mod n$ for any integer k.
- if $a \equiv b \mod n$ and $c \equiv d \mod n$, then $(a \cdot c) \equiv (b \cdot d) \mod n$

Exponentiation

- if $a \equiv b \mod n$, then $a^k \mod n \equiv b^k \mod n$ for any positive integer k.

Division

- if $GCD(k, n) = 1$ and $(k \cdot a) \equiv (k \cdot b) \mod n$, then $a \equiv b \mod n$

For further readings and proofs, we can visit this link.

2.3.2 Inverse Modular

Inverse modular of a in $\mod m$ is the value of b that makes the following equation true.

$$a \cdot b \equiv 1 \mod m$$

Some equations do not have the modular inverse. Therefore, we might not find the modular inverse of some equations.

• Naive Approach

In the naive approach, we need to iterate up to m and check if the current element satisfies the condition. If it satisfies the condition($a \cdot b \equiv 1 \mod m$) we select i as the inverse modulo of the a. The run-time of this algorithm becomes O(m)

• Optimized Approach for Prime Number m

Fermat's little theorem allows us to write the following for the prime number called m [7].

$$a^{m-1} \equiv 1 \mod m$$

Let's refer to the modular inverse of a as b. If we multiply both sides with b we would get the following formula,

$$b \equiv a^{m-2} \mod m$$

We get the latter formula, Since $a \cdot b \equiv 1 \mod m$. b deletes one of the a in a^{p-1} and makes it a^{p-2}

Therefore,the modular inverse of a, which is b becomes $a^{m-2} \mod m$. We can use fast exponentiation for finding a^{m-2}. The run-time of this algorithm is O(log(m))

2.4 GCD - Greatest Common Divisor

Greatest Common Divisor(GCD) of two numbers `A` and `B` is the largest number `D` that evenly divides both of the numbers `A` and `B`.

- GCD of 2 and 4 is 2
- GCD of 3 and 4 is 1
- GCD of 3 and 6 is 3
- GCD of 10 and 15 is 5

If GCD(a, b) is equal to 1, we call a and b coprime numbers.

2.4.1 Naive Approach

`Time Complexity: O(min(n,m))`

We can start from 1 and proceed until the minimum of these two numbers. If we encounter a number that is both divisible by A and B, we can say that the number is a common divisor. However, we have to go until the minimum of A and B. We are trying to find the greatest divisor. The GCD might equal to the minimum of A and B.

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
int main() {
    long long n,m;
    scanf("%lld%lld", &n, &m);

    long long minVal = min(n,m), gcd = 0;
    // Calculate the untime.
    clock_t tStart = clock();
    for(int i=1;i<=minVal;i++){
        // If n is evenly divisible by i, n%i will return 0.
        // C++ is a weakly typed language.
        // Therefore, We can change types.
        // False can cast to the integer as 0.
        // Therefore, If n is evenly divisible by i, n%i will return 0(False).
        if(!(n % i) && !(m % i))
            gcd = i;
    }
    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
    printf("The GCD is %lld\n", gcd);
    return 0;
}
```

Output

- `The input is 282542151(94180717*3) 470903585(94180717*5)`

- `Time taken:` **`2.652236s`**
- `The GCD is 94180717`

2.4.2 Euclidean Approach

`Time Complexity:` $O(\log(n+m))$

Euclid states that If A and B has GCD of any C, $A - K \cdot B$ has the GCD of C as well. We are going to use $A \mod B$ instead of $A - K \cdot B$. If we give K the biggest value that makes $A - K \cdot B$ the smallest non-negative number, we would get $A \mod B$. For example,

$$A = 50 \ \& \ B = 15. \text{ We know that } C = GCD(A, B) = 5$$

According to Euclid Method, let's give K some values,

$$K = 1,\ A - K \cdot B = 50 - 1 \cdot 15 = 35$$

$$K = 2,\ A - K \cdot B = 50 - 2 \cdot 15 = 20$$

$$K = 3,\ A - K \cdot B = 50 - 3 \cdot 15 = 5,\ C = GCD(A, B) = 5$$

$K = 3$ is the final value that makes A the smallest non-negative number. Let's find $A \mod B$

$$A = 50 \mod 15 = 5,\ B = 15,\ C = GCD(A, B) = 5$$

Therefore, we can say that changing A with $A = A - B \cdot K$ or changing B with $B = B - A \cdot K$ does not change the GCD of A and B. However both a and b should be positive [9].

So we know that $A \mod B$ does not change GCD. Therefore, we can take $A \mod B$ first.After this modulo operation, B will be bigger than A ($B > A$). So we can take $B \mod A$. After this modulo operation, A will be bigger than ($A > B$). So that, we can now do this recursively until one of them is zero. Let's give an example,

$$A = 13 \ \& \ B = 17$$

$$A \mod B \mid A = 13 \ \& \ B = 17$$

$$B \mod A \mid A = 13 \ \& \ B = 4$$

$$A \mod B \mid A = 1 \ \& \ B = 4$$

$$B \mod A \mid A = 1 \ \& \ B = 0$$

Since we hit 0 on the side of B, we can proudly say that our answer(GCD(A,B)) is 1.

The time complexity of this algorithm is logarithmic. The proof of this comes from the taking GCD two consecutive Fibonacci numbers. Further reading materials can be found here.

```cpp
#include <iostream>
#include <algorithm>

using namespace std;


long long calculateGCD(long long n, long long m){
    long long temp = 0;


    while(n != 0){
        temp = n;
        n = m%n;
        m = temp;
    }

    return m;
}

int main() {
    long long n,m;
    scanf("%lld%lld", &n, &m);

    long long minVal = min(n,m), gcd = 0;

    // Calculate the runtime of the function.
    clock_t tStart = clock();

    gcd = calculateGCD(n, m);

    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);

    printf("The GCD is %lld\n", gcd);
    return 0;
}
```

Output

- The input is 282542151(94180717*3) 470903585(94180717*5)
- Time taken: **0.000003s**
- The GCD is 94180717

2.5 LCM - Least Common Multiple

Least Common Multiple(LCM) of two numbers `A` and `B` is the minimum number `D` that is divisible by both of the numbers `A` and `B`.

- LCM of 2 and 4 is `4`
- LCM of 3 and 4 is `12`
- LCM of 3 and 6 is `6`
- LCM of 10 and 15 is `30`

2.5.1 Naive Approach

`Time Complexity: O(n•m)`

We can start from the maximum number and go until the multiplication of these two numbers. If we encounter that both are divisible by A and B, we can say that number is the LCM. Since we have found the LCM of A and B, we do not need to go further. We can terminate the loop.

The biggest LCM(N, M) that we can find is $N \cdot M$ if N and M have no common factors. So, there will be no common number that is divisible by both N and M except $N \cdot M$.

```cpp
#include <iostream>
#include <algorithm>
#include <time.h>
using namespace std;

int main() {
    long long n, m;
    scanf("%lld%lld", &n, &m);
    long long maxVal = max(n, m), lcm = 0;

    // Calculate the runtime of the function.
    clock_t tStart = clock();
    for(long long i=maxVal;i<=n*m;i++){
        // If i is both divisible by n and m, is lcm of these two numbers.
        if(i%n == 0 && i%m == 0){
            lcm = i;
            break;
        }
    }
    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
    printf("The LCM is %lld\n", lcm);
    return 0;
}
```

Output

- `The input is 6630 12673`

- `Time taken:` **`0.791119s`**
- `The LCM is 84021990`

2.5.2 GCD Approach

`Time Complexity:` $O(\log min(n, m))$

In this algorithm, we will use GCD(M, N) for finding LCM(N, M). We are going use the relationship between GCD(M, N) and LCM(M, N).

$$LCM(A, B) \cdot GCD(A, B) = A \cdot B$$

Let's prove the correctness of this formula by using Unique Factorization Theorem [10].

Let $a_1, a_2, a_3 \cdots a_n$ become prime numbers.

$$M = a_1^{b_1} \cdot a_2^{b_2} \cdots a_k^{b_k} \quad \text{and} \quad N = a_1^{c_1} \cdot a_2^{c_2} \cdots a_k^{c_k}.$$

Some `b` or `c` values can be 0.

$$LCM(M, N) = a_1^{d_1} \cdot a_2^{d_2} \cdots a_k^{d_k} \quad and \quad GCD(N, M) = a_1^{e_1} \cdot a_2^{e_2} \cdots a_k^{e_k}.$$

So we can write,

$$d_i = min(b_i, c_i) \quad and \quad e_i = max(b_i, c_i)$$

$$d_i + e_i = b_i + c_i$$

With using the latter formula, we can write the following equations.

$$LCM(M, N) \cdot GCD(M, N) = a_1^{d_1+e_1} \cdot a_2^{d_2+e_2} \cdots a_k^{d_k+e_k}$$

$$M \cdot N = a_1^{b_1+c_1} \cdot a_2^{b_2+c_2} \cdots a_k^{b_k+c_k}$$

Therefore, we can say that

$$LCM(A, B) \cdot GCM(A, B) = A \cdot B$$

Let's give an example and say A, B, C, D, E, and F are prime numbers,

$$X = A \cdot B \cdot C \cdot F \ \ \& \ \ Y = D \cdot E \cdot F$$

$$GCD(X, Y) = F \ \ \& \ \ LCM(X, Y) = A \cdot B \cdot C \cdot D \cdot E \cdot F$$

$$GCD(X, Y) \cdot LCM(X, Y) = A \cdot B \cdot C \cdot D \cdot E \cdot F^2$$

$$X \cdot Y = A \cdot B \cdot C \cdot D \cdot E \cdot F^2$$

As we can see from the proof and example, we can write the following formula.

$$LCM(A, B) = \frac{A \cdot B}{GCD(A, B)}$$

```cpp
#include <iostream>
#include <algorithm>

using namespace std;

// The function that finds GCD of two numbers.
long long calculateGCD(long long n, long long m){
    if(n == 0) return m;
    return calculateGCD(m%n, n);
}

// The function that finds LCM of two numbers.
long long calculateLCM(long long n, long long m){
    return n * m / calculateGCD(n,m);
}

int main() {
    long long n,m;
    scanf("%lld%lld", &n, &m);

    long long minVal = min(n,m), lcm = 0;

    // Calculate the runtime of the sieve function.
    clock_t tStart = clock();

    lcm = calculateLCM(n, m);

    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);

    printf("The LCM is %lld\n", lcm);

    return 0;
}
```

Output

- The input is 6630 12673
- Time taken: **0.000005s**
- The LCM is 84021990

2.6 Benzout's Identity

`Time Complexity:` $O(\log min(n, m))$

Benzout Identity algorithm allows us to find the **x** and **y** integer values from the any **a** and **b** values in the following formula [11]. Benzout Identity algorithm is also called Extended Euclidean Algorithm.

$$a \cdot x + b \cdot y = gcd(a, b)$$

We know that we can use a recursive formula for finding the GCD of a and b by using the Euclidean Algorithm. Therefore, we can give similar values in the formula for finding **x** and **y**. So let's say that we will give (b mod a, a) instead of (a,b) and (x_1, y_1) instead of (x,y) in the latter formula.

$$(b \mod a) \cdot x_1 + a \cdot y_1 = gcd(a, b)$$

$$b \bmod a = b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a$$

$$(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a) \cdot x_1 + a \cdot y_1 = gcd(a, b)$$

Let's rearrange the latter formula.

$$(b - \left\lfloor \frac{b}{a} \right\rfloor \cdot a) \cdot x_1 + a \cdot y_1 = gcd(a, b)$$

$$b \cdot x_1 + a \cdot \left(y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1 \right) = gcd(a, b)$$

If we put x instead of x_1 and y instead of y_1, we would get the formula of x and y.

$$y = x_1$$
$$x = \left(y_1 - \left\lfloor \frac{b}{a} \right\rfloor \cdot x_1 \right)$$

This was the one execution of the recursive approach. We should find `x` and `y` until we hit the end of the Euclidean GCD approach. In other words, we need to repeat this equation as in the Euclidean approach.

```cpp
#include <iostream>

using namespace std;

// Take x and y as the reference to change them while going inside the recursion
int gcd(int a, int b, int & x, int & y) {
    // End of the recursion
    if (a == 0) {
        x = 0;
        y = 1;
        return b;
    }

    int x1, y1;
    int d = gcd(b % a, a, x1, y1);
    // Find x and y value, recursively
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}


int main() {
    int a,b;
    scanf("%d %d", &a, &b);
    int x,y;
    // Calculate the runtime of the sieve function.
    clock_t tStart = clock();

    gcd(a, b, x, y);

    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);

    printf("The x is %d, the y is %d\n", x, y);
}

```

Output

- The input is 282542151(94180717*3) 470903585(94180717*5)
- Time taken: **0.000004s**
- The x is 2, the y is -1

Let's check if we find the values correctly or not.

$$gcd(282542151, 470903585) = 94180717$$
$$a \cdot x - b \cdot y = gcd(a, b)$$
$$\text{a} = 282542151,\ \text{b} = 470903585,\ \text{x} = 2,\ \text{y} = \text{-1}$$
$$282542151 \cdot 2 - 470903585 * 1 = 94180717 = gcd(282542151, 470903585)$$

3 Exponentiation

3.1 Naive Approach

`Time Complexity: O(k)`

The problem is finding the n^k. In the naive approach, we simply multiply n with itself k times.

```cpp
#include <iostream>
#include <time.h>

using namespace std;

#define mod 1000000007

long long exp(long long n, long long k){
    long long res = 1;
    while(k--){
        res *= n;
        // Since it might be too large for long long, we take the modulo.
        res %= mod;
    }
    return res;
}

int main() {
    long long n,k;
    scanf("%lld%lld", &n, &k);

    // Calculate the runtime of the function.
    clock_t tStart = clock();

    long long res = exp(n, k);

    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);

    printf("%lld\n", res);

    return 0;
}

```

Output

- `The input is 2 100000000`
- `Time taken:` **`1.575307s`**
- `494499948`

3.2 Fast Exponentiation Approach

`Time Complexity:` $O(\log k)$

We will get a benefit from the following rule for finding the exponentiation.

$$n^k = \begin{cases} n^{k/2} * n^{k/2} & \text{if k is even} \\ n^{(k-1)/2} * n^{(k-1)/2} * n & \text{if k is odd} \end{cases} \tag{1}$$

Since we know the rule, we can call the rule recursively.

```cpp
#include <iostream>
#include <time.h>

using namespace std;

#define mod 1000000007

long long fastExp(long long n, long long k){
    if (k == 0)
        return 1;

    n %= mod;
    long long temp = fastExp(n, k >> 1);

    // If k is odd return n * temp * temp
    // If k is even return temp * temp

    // The product of two factors that are each bounded by mod is bounded by
    // mod squared. If we multiply this product with yet another factor that
    // is bounded by mod, the result will be bounded by mod cubed. As mod is
    // equal to 1000000007 by default, the result might not fit into
    // long longs, leading to an overflow. To avoid that, we must take
    // the modulus of n * temp before multiplying it with temp yet again.
    if (k & 1)
        return n * temp % mod * temp % mod;
    return temp * temp % mod;
}
int main() {
    long long n,k;
    scanf("%lld%lld", &n, &k);

    // Calculate the runtime of the function.
    clock_t tStart = clock();

    long long res = fastExp(n, k);
    printf("Time taken: %.6fs\n", (double)(clock() - tStart)/CLOCKS_PER_SEC);
    printf("%lld\n", res);

    return 0;
}

```

Output

- `The input is 2 100000000`
- `Time taken:` **`0.000005s`**
- `494499948`

3.2.1 Calculating Fibonacci with Fast Matrix Exponentiation

`Time Complexity:` $O(\log n)$

This section is about finding the Nth Fibonacci Number by using the fast exponentiation approach. We all heard the Fibonacci Sequence.

We are going to use the matrix representation of the Fibonacci sequence. We can write the following equation. [17]

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} F_{n+1} + F_n \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

$$\begin{bmatrix} F_{n+3} \\ F_{n+2} \end{bmatrix} = \begin{bmatrix} F_{n+2} + F_{n+1} \\ F_{n+2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \times \begin{bmatrix} F_{n+1} \\ F_n \end{bmatrix}$$

Therefore, we can write the following rule for the Fibonacci sequence in matrix form,

$$\begin{bmatrix} F_{n+2} \\ F_{n+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n \times \begin{bmatrix} F_2 \\ F_1 \end{bmatrix}$$

So far so good, we have a general matrix equation for the finding Nth Fibonacci number. The fast exponentiation comes to play in here. We can find the matrix exponentiation as in the following.

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{cases} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n/2} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n/2} & \text{if n is even} \\ \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{(n-1)/2} * \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{(n-1)/2} & \text{if n is odd} \end{cases} \tag{2}$$

The latter formula gives us the find the matrix that has Nth Fibonacci number. Since we are finding our matrix Exponentiation in $\log n$ processes, we would get the time complexity of $O(\log n)$.

4 Prefix Sum

Prefix Sum dizisi bir dizinin prefixlerinin toplamlarıyla oluşturulan bir veri ya pısıdır. Prefix sum dizisinin i indeksli elemanı girdi dizisindeki 1 indeksli elemandan i indeksli elemana kadar olan elemanların toplamına eşit olacak şekilde kurulur. Başka bir deyişle:

$$sum_i = \sum_{j=1}^{i} a_j$$

Örnek bir A dizisi için prefix sum dizisi şu şekilde kurulmalıdır:

A Dizisi	4	6	3	12	1
Prefix Sum Dizisi	4	10	13	25	26
	4	4 + 6	4 + 6 + 3	4 + 6 + 3 + 12	4 + 6 + 3 + 12 + 1

Prefix sum dizisini kullanarak herhangi bir $[l, r]$ aralığındaki elemanların toplamını şu şekilde kolaylıkla elde edebiliriz:

$$sum_r = \sum_{j=1}^{r} a_j$$

$$sum_{l-1} = \sum_{j=1}^{l-1} a_j$$

$$sum_r - sum_{l-1} = \sum_{j=l}^{r} a_j$$

4.1 Örnek Kod Parçaları

Prefix Sum dizisini kurarken $sum_i = sum_{i-1} + a_i$ eşitliği kolayca görülebilir ve bu eşitliği kullanarak $sum[]$ dizisini girdi dizisindeki elemanları sırayla gezerek kurabiliriz.

```
const int n;
int sum[n+1], a[n+1];
// a dizisi girdi dizimiz, sum dizisi de prefix sum dizimiz olsun.

void build() {
  for (int i = 1 ; i <= n ; i++)
    sum[i] = sum[i - 1] + a[i];
  return;
}

int query(int l,int r) {
  return sum[r] - sum[l - 1];
}
```

4.2 Zaman Karmaşıklığı

Prefix sum dizisini kurma işlemimizin zaman ve hafıza karmaşıklığı $O\ (N)$. Her sorguya da $O(1)$ karmaşıklıkta cevap verebiliyoruz.

Prefix sum veri yapısı ile ilgili problem: Link.

5 Suffix Sum

Suffix Sum is a precomputation technique in which the sum of all the elements of the original array from an index i till the end of the array is computed.

Therefore, this suffix sum array will be created using the relation:

$$suffixSum[i] = arr[i] + arr[i+1] + arr[i+2] \ldots + arr[n-1]$$

Naive Approach:

The naive approach to solve the problem is to traverse each element of the array and for each element calculate the sum of remaining elements to its right including itself using another loop.

Below is the implementation of the approach:

```
#include<bits/stdc++.h>
using namespace std;

// Driver's code
int main() {

	vector<int> arr = { 10, 14, 16, 20 };

	int n = arr.size();

	// initialize the suffix sum array with all elements as 0
	vector<int> suffixSum(n, 0);

	for(int i=0; i<n; i++) {
		// calculate the sum of remaining elements to the right
		for(int j=i; j<n; j++) {
			suffixSum[i] += arr[j];
		}
	}

	// Printing the computed suffix sum array
	cout << "Suffix sum array: ";
	for(int i=0; i<suffixSum.size(); i++) {
		cout << suffixSum[i] << " ";
	}

	return 0;
}
```

Output
Suffix sum array: 60 50 36 20

Time Complexity: O(n*n) where n is size of input array. This is because two nested loops are executing.
Auxiliary Space: O(N), to store the suffix sum array.

Approach: To fill the suffix sum array, we run through index N-1 to 0 and keep on adding the current element with the previous value in the suffix sum array.

- Create an array of size N to store the suffix sum.
- Initialize the last element of the suffix sum array with the last element of the original array **suffixSum[n-1] = arr[n-1]**
- Traverse the original array from N-2 to 0
    - For each index i find the suffix sum and store it at suffixSum[i]
    - **suffixSum[i] = suffixSum[i + 1] + arr[i]**
- Return the computed suffix sum array.

Below is the implementation of the above approach to create a suffix sum array:

```
// suffix sum array
#include <bits/stdc++.h>
using namespace std;

// Function to create suffix sum array
vector<int> createSuffixSum(vector<int> arr, int n)
{
   // Create an array to store the suffix sum
   vector<int> suffixSum(n, 0);

   // Initialize the last element of
   // suffix sum array with last element
   // of original array
   suffixSum[n - 1] = arr[n - 1];

   // Traverse the array from n-2 to 0
   for (int i = n - 2; i >= 0; i--)

      // Adding current element
      // with previous element from back
      suffixSum[i] = suffixSum[i + 1] + arr[i];

   // Return the computed suffixSum array
   return suffixSum;
}

// Driver Code
int main()
{
   vector<int> arr = { 10, 14, 16, 20 };
   int N = arr.size();

   // Function call to fill suffix sum array
   vector<int> suffixSum = createSuffixSum(arr, N);

   // Printing the computed suffix sum array
   cout << "Suffix sum array: ";
   for (int i = 0; i < N; i++)
      cout << suffixSum[i] << " ";
}
```