

**ДВАДЕСЕТ И ТРЕТА УЧЕНИЧЕСКА СЕКЦИЯ**

**УС’23**

**ТЕМА НА ПРОЕКТА**

**Софтуер за шифриране и дешифриране на текст по модел на ЕНИГМА машина**

**Автор(и):**

**Никола Радостинов Димитров 12в клас ППМГ „Никола Обрешков“ Казанлък**

(трите имена, училище, град, клас)

**Научен ръководител (консултант):**

**Цветанка Савова - Учител по информатика ППМГ „Никола Обрешков“ Казанлък**

(име, фамилия, длъжност, месторабота)

**Софтуер за шифриране и дешифриране на текст по модел на ЕНИГМА машина**

Проектът е самостоятелна разработка в сферата на криптографията. Програмата шифрира и дешифрира текст на английски език чрез ЕНИГМА шифъра. Състои се от две части.

Първата част позволява на потребител да закодира собствен текст по модел на оригиналната машина.

Втората страница имплементира алгоритъм, които разбива шифрован текст чрез редица статистически техники и извежда „ключът“ за декодирането му.

Различните алгоритми са представени чрез интуитивен и атрактивен потребителски интерфейс. В отделни страници са представени допълнителна информация за машината и насоки за употреба на софтуера. Цялата имплементация е хостната чрез GitHub Pages.

**Software for encryption and decryption using the ENIGMA cypher.**

The project is the field of cryptography. This is an independently developed algorithm for encrypting and decrypting English text with the ENIGMA cypher. It consists of two parts

The first part lets the user encrypt their own message with an accurate model of the original machine.

The second part implements statistical techniques to analyze encoded text and output the “key” to its decryption.

The two algorithms are presented though an intuitive UI that also offers additional information about the machine and instructions for operation. The software is hosted through GitHub Pages.

**Съдържание**

1. Увод

* История
* Цели на проекта
* Използвани технологии
* Връзка към сайта

1. Изложение

* Процес на шифриране и дешифриране
* Елементи на машината
* Графичен дизайн на сайта – Снимки, цветова схема, структура, главна страница
* Имплементация на кодирането
  + Визуално представяне – снимка, прехвърляне на стойности, DOM чрез Blazor, AppState
* Функция за декодиране
* Имплементация
  + Страница за декодиране – снимка, Base, AOT, async
* Допълнителни страници – responsive design, преизползване
* Хостинг - GitHub Pages, Publish, Actions, PAT, Secret, VM

1. Заключение

* По-нататъшно развитие
* Срещнати трудности
* Използвана литература

**Увод:**

**История**

ЕНИГМА машината е устройство за кодиране на текст, изобретено от немския инженер Артур Шербиус през 1918г. Тя е най-известна с това, че е използвана от нацистка Германия през Втората световна война. Работата на полския математик Мартин Рейевски и на британския учен Алан Тюринг не само води до невероятен напредък в математиката и криптографията, но и поставя началото на модерните компютри и компютърни науки.

**Цели на проекта**

Основна цел на проекта е да предостави лесно достъпен за нормалния потребител пръв поглед в криптографията и нейните употреби. Вторичен стремеж е въвеждането на собствен принос чрез „превеждане“ на алгоритмите на български език.

**Рецензия**

**Използвани технологии**

Visual Studio Code

Програмен език C# и .NET 6 за имплементиране на функционалност

HTML и CSS (включително библиотека Bootstrap) за визуално оформление

Blazor WebAssembly за интеграция между дизайн и функционалност

GitHub Pages за publishing и hosting на проекта.

**Връзка към сайта:**

<https://thefichoza.github.io/PublishDemo/Encryption>

**Това е най-актуалната версия на проекта!**

**Изложение:**

**Процес на шифриране и дешифриране**

Машината на практика заменя една буква с друга. Това се случва чрез сложен електричен и механичен процес. Шифрирането се осъществява, като се въведе оригиналния текст в машината, която го шифрира спрямо конфигурацията на различните елементи. Ако се въведе шифрираният текст при същите настройки, той ще бъде дешифриран. Най-интересната част от структурата на машината е това, че настройките се променят с всяка въведена буква, което значи, че всеки символ е закодиран по различен начин.

**Елементи на машината**

Машината се състои от 5 основни елемента: клавиатурата за въвеждане, лампичките, които показват крайния резултат, роторите, рефлекторът(отразителят) и таблото за връзки.

Кодирането на буквите се определя от пътя на електричеството. Общият случай е: **клавиатура > табло > >ротори(1,2,3) > отразител > ротори(3,2,1) > табло > лампички**

1. Клавиатурата

Оттук се въвеждат буквите от текста. Има 26 клавиша, по един за всяка буква от латинската азбука, като всеки един клавиш е свързан със собствена жица към роторите. В нашата имплементация просто използваме компютърната клавиатура.

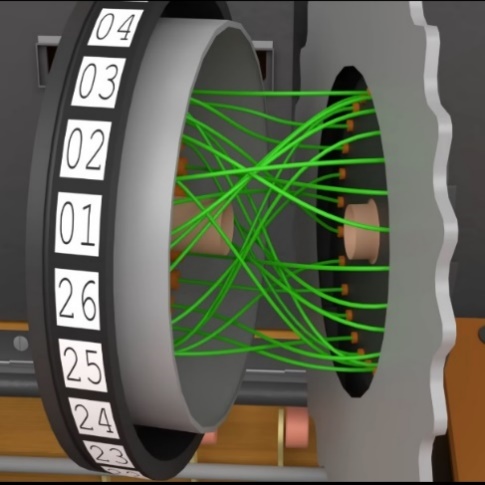
1. Лампичките

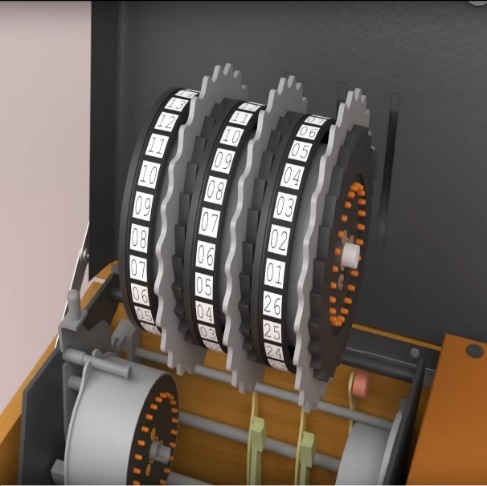
Те показват новата буква, получена след кодирането. За всяка една натисната буква светва само една лампичка. Вместо лампички ние просто използваме текстова кутия за извеждане на (де)кодирания текст.

3, 4) Роторите и отразителят

Това е най-сложната част в структурата на машината. Тези два елемента извършват почти изцяло кодирането.

Роторите:

Всяка машина идва с 5 ротора, три от които влизат в апарата. Роторът представлява зъбно колело с 26 страни за 26-те букви от азбуката. Всяка страна има по един вход и изход, през които минава кабел. Пътищата на кабелите обаче са разбъркани. Например, входът на 1(А) маже да излиза през изхода за 15(О). Изходът на един ротор може да се свърже с входа на друг т.е. след като мине и през трите ротора, една буква ще бъде сменена три пъти.

Оригиналната позиция на роторите е 1, 1, 1 т.е. вход 1 на първия ротор съвпада с кабела към клавиш А, 2 с B, 3 с C и т.н. След всеки въведен клавиш обаче първият ротор се завърта веднъж, като в този случай позициите стават 2, 1, 1. Това значи, че буквата **А** съвпада с вход **2**, **B** с **3**, **C** с **4** и т.н. Ако началната позиция е 26, 1, 1 и първият ротор се завърти веднъж, завъртането ще се промени на 1, 2, 1 т.е. **вторият** ротор ще се завърти веднъж.

Последна особеност на роторите е т.нар. двойно завъртане. То се случва при позиции 26, 25, x т.е. преди завъртане на третия ротор. В тази позиция следващият натиснат бутон ще завърти ротор 1 и 2 (1, 26, x), а по-следващият - ротор 1, 2 и 3(2, 1, x+1)

Рефлектор(отразител)

Това е втората част от роторната структура. Рефлекторът е свързан с изходите на последния трети ротор, като ги свързва две по две. Например, нека Z и G са свързани в отразителя. Той ще приеме изход Z и ще го изпрати отново в изходите на третия ротор, но на буква G. Това позволява на системата да премине през роторите още веднъж, преди да излезе. С други думи, една буква бива променена **поне 7 пъти**, преди да бъде изведена като изход.

5) Табло с връзки

Чрез това табло могат да се чифтосат всеки две не чифтосани букви, при което всяка въведена чрез клавиатурата или изведена към лампичките буква бива заменена с нейният чифт. Всяка буква може да се чифтоса с **една** друга или **да остане не чифтосана**.

**Настройки на машината:**

Всяка машина идва с пет ротора. От тях само три влизат наведнъж в машината. Последователността на използваните ротори също влияе на крайния резултат. Има V53(60)различни последователности. След това всеки ротор има 26 начални позиции. Това са (26\*26\*26)\*60 = 17,576\*60 = 1,054,560 различни настройки на роторите. Оригиналната машина използва 10 чифта в таблото. Тук уравнението е (26!)/((10!) \* (6!) \* (2^10)) = 150,738,274,937,250 комбинации. Ние обаче използваме само три връзки, което намаля бройката до (26!)/((3!) \* (20!) \* (2^3)) = 3,453,450 настройки. Това са общо 1,054,560\*3,453,450 = 3,641,870,232,000 различни настройки на машината.

**Графичен дизайн на сайта:**

**Обща структура:**

Diagram

Description automatically generated

Сайтът се състои от 6 страници. От тях една е главна, две изпълняват съответно кодиране и декодиране, а останалите три съдържат допълнителна информация за машината и потребителски насоки. Навигацията се случва почти изцяло през главната страница, като всяка друга страница е двустранно свързана с нея. Единствено кодирането и декодирането са свързани двустранно, без да минават през главната страница. Благодарение на навигационното меню двете страници имат достъп и до две от трите страници с информация. Страниците с информация се връщат единствено към главната.

**Визия на страниците:**

Дизайнът на страниците беше направен във Figma, преди да бъде реализиран като уеб интерфейс. Страниците следват модела на проекта без особени промени

Връзка към mockup-и във Figma: <https://www.figma.com/file/jkPkS2eJg42JIMZSa25FHk/ENIGMA?node-id=0%3A1&t=VgpFDxWMHLX1YMbi-1>

**Цветова схема:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Цвят |  |  |  |  |  |  |
| Код | #282828 | #797878 | #2b9d03 | #ffffff | #d9d9d9 | #ffc107 |
| Употреба | **фон** | | **елементи** | | | |

Това са основните цветове на дизайна. Различните цветове на елементите изпълняват различни функции.

Бяло – нормален текст.

Зелено( + нюанси)

* Бутон или текст с връзка – в страниците с функционалност.
* Акцентиран текст – в главната и допълнителните страници

Сиво – текстова кутия. Използвано и за таблото с връзки

Жълто – Подзаглавия в страниците с информация. Разчупва белият текст.

**Структура на Blazor приложение:**

Blazor се използва за създаването на т.нар. Single PageApplication т.е. уебсайт със само един html файл(index.html). В index.html се вика Javascript функция, която зарежда определения компонент заедно с неговия Layout(разпределение). @Body атрибутът във всяко разпределение определя позицията на съответния .razor(разширение за Blazor файл) компонент

**Index.razor:** Graphical user interface, text

Description automatically generated

**Това е главната страница на проекта. Тя е основна връзка с остатъка от проекта**

Тя не имплементира никаква функционалност сама по себе си и затова може да се реализира единствено чрез HTML и CSS. Единствено в нея нормалният текст е зелен.

**Реализация:**

***@page отразява URL-а на определеният елемент (В случая: http://(URL)/ т.е. начална страница)***

@page "/"

<PageTitle>Index</PageTitle>

<div class="container-fluid p-5">

    <div class="row">

        <div class="col-6 mt-4">

            <h1 class="home-page green-color fw-bold text-center">ENIGMA</h1>

            <p class="green-color text-center fw-bold mt-2 fs-4">Encryption and Decryption</p>

            <p class="green-color text-center fw-bold mt-2 fs-4">The Birth of Computer Science</p>

            <p class="green-color text-center mt-2 fs-5">

                The history of ENIGMA is strongly connected to the development of the first computers. Here, I attempt to retell that story.

                This programme seeks to authentically replicate

                an ENIGMA encrypting device and crack ciphers

                with both an authentic and modern method.

            </p>

            <div class="text-center d-flex justify-content-center">

                <a class="btn btn-home me-3 fs-4 mt-3" href="Encryption"><img src="Sources/Images/arrow-down-right-square-fill.svg" class="mb-1" />&nbsp ENIGMA device</a>

                <a class="btn btn-home fs-4 mt-3" href="Decryption"><span class="text-nowrap">Decryption algorithms</span></a>

            </div>

        </div>

        <div class="col-6 d-flex justify-content-center">

            <img src="Sources/Images/titleImage.png" style="max-width: 100%"/>

        </div>

    </div>

</div>

Важно е да се отбележи и употребата на Bootstrap. Bootstrap е библиотека с редица предварително подготвени класове и стойности, която бива автоматично включена във всяко ASP.NET приложение.

Някои от по-важните Bootstrap класове тук са:

* container-fluid – използвано в най-горния <div>. Кара елементът да запълни целият бащин елемент
* d-flex – употребено много често из проекта. Определя свойство display: flex. От една страна, позволява центриране на всички съдържани елементи чрез justify-content-center. От друга страна, по подразбиране нарежда елементите един до друг, без да изисква float
* col-6 – определя ширина. В Bootstrap целият екран е разделен на 12 колони. Следователно col-6 ще заеме половината от тях или width: 50%.
* m-X/p-X/fs-X – определят съответно margin, padding и font-size. X представлява цяло число от 1 до 5(6 за font-size). m и p могат да се използват самостоятелно или с пояснител (mt – margin-top, ps – padding-start(left)) като по-голямо цяло число означава по-голяма стойност. Единствено fs е наобратно, тъй като fs-1 = <h1>, fs-2 = <h2> и т.н.

**Авторски CSS класове:**

.home-page {

    font-size: 54pt;

}

.green-color {

    color: #3FE704 !important;

    text-decoration: none;

}

.btn-home {

    color: #000000;

    font-weight: bold;

    background-color: #3fe704;

}

    .btn-home:hover {

        background-color: #308d0f;

    }

    .btn-check:focus + .btn-home, .btn-home:focus {

        background-color: #308d0f;

        box-shadow: 0 0 0 0.25rem rgba(130, 138, 145, 0.5);

    }

Особености:

green-color: Ако два класа променят едно и също свойство, прилага се !important атрибутът. Ако баща и наследник имат конфликт на свойства, взима се класът на наследника.

btn-home: определя разл. цветове при преминаване на мишката и при натискане. rgba() определя rgb цвят+opacity

**MainLayout.razor**

@inherits LayoutComponentBase

    <header class="navbar-color">

        <nav class="navbar navbar-expand-sm navbar-toggleable-sm box-shadow" style="height: 17vh;">

            <div class="container-fluid p-3 d-flex mt-2">

                <a href="" class="col-5 text-center green-color nav-title-size fw-bold hoverer"><img src="Sources/Images/icon.png" class="mb-1" style="height: 10vh"/>&nbsp ENIGMA Project</a>

                <div class=" d-flex justify-content-between">

                    <a href="History" class="nav-link nav-size green-color hoverer text-nowrap">Historical context</a>

                    <a href="Instructions" class="nav-link nav-size green-color hoverer text-nowrap">Operation of the ENIGMA</a>

                    <a href="Methods" class="nav-link nav-size green-color hoverer text-nowrap">Decryption methods</a>

                </div>

            </div>

        </nav>

    </header>

<div class="page">

    <main>

        <article class="content p-0">

            @Body

        </article>

    </main>

</div>

.navbar-color{

    background-color: rgba(217, 217, 217, 0.15);

}

.nav-size

{

    font-size: 1.6vw;

}

.nav-title-size

{

    font-size: 3vw;

}

    .hoverer{

        transition: background-color 0.15s ease-in-out;

    }

.hoverer:hover {

    color: #308d0f !important;

}

Това е оригиналният Layout, сега използван единствено от Index. Създава навигационно меню с връзки към допълнителните страници. Съдържанието на Index.razor се появяват на мястото на @Body, а @inherits LayoutComponentBase позволява на компонента да функционира като Layout.

Тук се виждат и първите опити за responsive дизайн чрез употребата на vh(viewport height) и vw(viewport width). Тези две стойности са определени от размера на прозореца и следователно се адаптират спрямо неговата промяна.

transition property в клас hoverer води до по-плавна промяна на background-color при селекция на елемента.

.

**Реализация на кодирането**

**Визуално представяне - Encryption.razor**

**Graphical user interface, application

Description automatically generated**

В тази страница елементите на машината са представени визуално. Потребителят пише собствен текст на английски в текстовата кутия „Original Text“ и избира конфигурация на роторите и на таблото с връзки. При натискане на бутона Encrypt въведеният текст бива закодиран и резултатът се извежда в „Encrypted Text“

**Елементи на дизайна :**

***URL: http://(URL)/Encryption***

@page "/Encryption"

@layout EncDecLayout

<PageTitle>Encryption</PageTitle>

**Encrypted Text:**

                <div class="col-4 align-items-center d-flex flex-column">

                    <label class="text-white fw-bold fs-4">Encrypted Text:</label><br/>

                    <textarea readonly class="form-control fs-6 fw-bold textbox">@encrypted</textarea>

                </div>

**Original Text:**

                <div class="col-4 align-items-center d-flex flex-column">

                    <label class="text-white fw-bold fs-4">Original Text:</label><br/>

                    <textarea @bind="original" class="form-control fs-6 fw-bold textbox"></textarea>

                </div>

.textbox

{

    width: 85%;

    height: 85%;

    background-color: rgb(217, 217, 217);

    resize: none;

}

Чрез **@bind** съдържанието на променлива се взима от съдържанието на елемент, а поставянето на **@{име на променлива}** между отварящ и затварящ таг прави обратното

**Табло с връзки:**

<div class="col-8 d-flex flex-column justify-content-center">

                    <h1 class="text-white fw-bold fs-4 mt-5">Plugboard settings:</h1>

                    <div class="row d-flex justify-content-between mt-4">

                        <a @onclick="() => Connect('A')" class="@classes[0]">A</a>

                        <a @onclick="() => Connect('B')" class="@classes[1]">B</a>

                        <a @onclick="() => Connect('C')" class="@classes[2]">C</a>

                        <a @onclick="() => Connect('D')" class="@classes[3]">D</a>

                        <a @onclick="() => Connect('E')" class="@classes[4]">E</a>

                        <a @onclick="() => Connect('F')" class="@classes[5]">F</a>

                        <a @onclick="() => Connect('G')" class="@classes[6]">G</a>

                        <a @onclick="() => Connect('H')" class="@classes[7]">H</a>

                        <a @onclick="() => Connect('I')" class="@classes[8]">I</a>

                    </div>

                    <div class="row d-flex justify-content-between mt-3">

                        <a @onclick="() => Connect('J')" class="@classes[9]">J</a>

                        <a @onclick="() => Connect('K')" class="@classes[10]">K</a>

                        <a @onclick="() => Connect('L')" class="@classes[11]">L</a>

                        <a @onclick="() => Connect('M')" class="@classes[12]">M</a>

                        <a @onclick="() => Connect('N')" class="@classes[13]">N</a>

                        <a @onclick="() => Connect('O')" class="@classes[14]">O</a>

                        <a @onclick="() => Connect('P')" class="@classes[15]">P</a>

                        <a @onclick="() => Connect('Q')" class="@classes[16]">Q</a>

                        <a @onclick="() => Connect('R')" class="@classes[17]">R</a>

                    </div>

                    <div class="row d-flex justify-content-between mt-3 ms-5" style="width: 90%">

                        <a @onclick="() => Connect('S')" class="@classes[18]">S</a>

                        <a @onclick="() => Connect('T')" class="@classes[19]">T</a>

                        <a @onclick="() => Connect('U')" class="@classes[20]">U</a>

                        <a @onclick="() => Connect('V')" class="@classes[21]">V</a>

                        <a @onclick="() => Connect('W')" class="@classes[22]">W</a>

                        <a @onclick="() => Connect('X')" class="@classes[23]">X</a>

                        <a @onclick="() => Connect('Y')" class="@classes[24]">Y</a>

                        <a @onclick="() => Connect('Z')" class="@classes[25]">Z</a>

                    </div>

                </div>

1. Тук се вижда и употребата на Blazor за DOM Manipulation т.е. динамична промяна на дизайна. За разлика от JS, Blazor няма лесен директен достъп до елементите и техните свойства. Вместо това променям стрингови променливи, които определят класовете на елементите. Променените свойства се визуализират автоматично, без да се презарежда страницата.
2. @onclick свойството позволява викането на C# метод при натискане на елемент с мишката. Ламбда изразът е нужен, за да се придаде параметър на метода(@onclick=”Метод(параметър)” е невалиден синтаксис)

Всички елементи в Plugboard започват със свойство plug

.plug {

    background-color: rgb(217, 217, 217);

    line-height: 70px;

    text-align: center;

    font-size: 16pt;

    font-weight: bold;

    border-radius: 50%;

    width: 70px;

    color: black;

    text-decoration: none;

}

При натискане на буква се вика Connect метод

    private void Connect(char symbol)

    {

        classes[(int)(symbol-'A')] += $" plug-conn-{count/2} plug-disable";

        count++;

        temp += symbol;

        if(count%2==0)

        {

            plugs.Add($"{temp[0]}-{temp[1]}");

            temp="";

        }

        if(count==10)

        {

            for (int i = 0; i < classes.Length; i++)

            {

                classes[i]+=" plug-disable";

            }

            checker = "background-color: crimson";

        }

    }

Методът съдържа буквата във временен string до натискане на втора връзка, при което двойката бива записана в списъка с връзки. На селектираният елемент се придават класове plug-conn(0-4), който оцветява всяка двойка, и plug-disable, който не позволява натискането на същата буква отново.

След пет избрани чифта, цялото табло спира да е интерактивно до нулиране на настройките чрез „Reset settings“ бутона.

Background pattern

Description automatically generated

***Всеки две еднакво оцветени букви са чифт. Числото до „Number of plug pairs“ става червено при достигане на максималния брой връзки. Именно това определя променливата checker.***

**Ротори:**

**A picture containing graphical user interface

Description automatically generated**

                        <RotorElement @bind-Index="rotor\_positions[0]" Position="0"></RotorElement>

                        <RotorElement @bind-Index="rotor\_positions[1]" Position="1"></RotorElement>

                        <RotorElement @bind-Index="rotor\_positions[2]" Position="2"></RotorElement>

Използван е авторски компонент <RotorElement> с цел намаляне на повтарящия се код(същото ще бъде направено и с таблото с връзки)

**RotorElement.razor**

Представлява един ротор. Съдържа две основни части:

1. **Падащо меню** – за избиране на трите ротора

    <div class="dropdown-hover" @onmouseover="UpdateDropdown">

    <h2 class="green-color fw-bold fs-3 p-2 p-container">@AppState.Rotor\_Config[Position]</h2>

    <div class="dropdown-body">

        <h2 class="@rotorProps[0]" @onclick="() => SetRotor(1)">1</h2>

        <h2 class="@rotorProps[1]" @onclick="() => SetRotor(2)">2</h2>

        <h2 class="@rotorProps[2]" @onclick="() => SetRotor(3)">3</h2>

        <h2 class="@rotorProps[3]" @onclick="() => SetRotor(4)">4</h2>

        <h2 class="@rotorProps[4]" @onclick="() => SetRotor(5)">5</h2>

    </div>

    </div>

.dropdown-hover{

    border-top-right-radius: 40px;

    border-top-left-radius: 40px;

}

.dropdown-hover:hover{

    background-color: #797878;

}

.dropdown-body{

    display: none;

    background-color: #797878;

    position: absolute;

    margin-top: -5px;

    border-bottom-left-radius: 40px;

    border-bottom-right-radius: 40px;

    transition: display 0.25s ease-in-out;

}

.dropdown-body > p{

    padding: 5% 0 5%;

}

.dropdown-hover:hover .dropdown-body{

    display: block;

}

Падащото меню е позиционирано абсолютно спрямо страницата т.е. независимо от други елементи. Това му позволява за покрива останалите елементи, без да ги измества надолу.

Бележка: Методът UpdateDropdown(), който актуализира падащото меню при смяна на ротор, а активен само в индивидуалния ротор т.е. Промяната ще се отрази в едното падащо меню, но не и в другите. За щастие менютата остават скрити през повечето време и следователно мога да ги актуализирам в момента на визуализация чрез @onmouseover свойство.

1. **Основна част** – определя началните позиции

    <button @onclick="() => btnIncrease()" class="btn btn-encrypt"><img src="Sources/Images/caret-up-fill.svg" class="img-size"/></button>

<p class="fw-bold bg-white p-2 fs-5 mb-2 text-center " style="width: 39px">@FormatSettings(Index)</p>

    <p class="fw-bold p-2 fs-5 mb-2 text-center" style="width: 39px;background-color: yellow">@FormatSettings(Index+1)</p>

    <p class="fw-bold bg-white p-2 fs-5 mb-2 text-center" style="width: 39px">@FormatSettings(Index+2)</p>

    <button @onclick="() => btnDecrease()" class="btn btn-encrypt"><img src="Sources/Images/caret-down-fill.svg" class="img-size" /></button>

Основна тук е употребата на параметри:

    [Parameter]

    public int Index { get; set; }

    [Parameter]

    public int Position { get; set; }

    [Parameter]

    public EventCallback<int> IndexChanged { get; set; }

Променлива с [Parameter] етикет може да бъде извикана в тага като свойство

<RotorElement @bind-Index="rotor\_positions[0]" Position="0"></RotorElement>

EventCallback изпраща сигнал на горния елемент, когато дадена стойност е актуализирана. Намр. в btnIncrease

    private Task btnIncrease()

    {

        Index = (Index + 1) % 26;

        return IndexChanged.InvokeAsync(Index);

    }

@bind-Index свързва параметър Index на дадения ротор с елемент на целочислен масив rotor\_positions т.е. промяната на една от стойностите се отразява и върху другата.

**CascadingAppState.razor** - Прехвърляне на стойности между компоненти

<CascadingValue Value="this">

    @ChildContent

</CascadingValue>

@code {

    [Parameter]

    public RenderFragment ChildContent { get; set; }

    private string buttonState, pageLink;

    private int[] rotor\_config;

    public string ButtonState

    {

        get

        {

            return buttonState;

        }

        set

        {

            buttonState = value;

            StateHasChanged();

        }

    }

    public string PageLink

    {

        get

        {

            return pageLink;

        }

        set

        {

            pageLink = value;

            StateHasChanged();

        }

    }

    public int[] Rotor\_Config

    {

        get

        {

            return rotor\_config;

        }

        set

        {

            rotor\_config=value;

            StateHasChanged();

        }

    }

}

Специфичната структура не променя самият дизайн, но ми позволява да създам атрибути, които са достъпни от всеки .razor компонент в приложението. Това се постига чрез [CascadingParameter]

    [CascadingParameter]

    public CascadingAppState AppState { get; set; }

Rotor\_Config е масивът с избраните ротори. Поставен е тук, за да избегна сложната комуникация между роторите.

Другите две стойности пък се използват в навигационното меню **EncDecLayout.razor**

За момента **бутонът „EN“ няма функция**. В бъдеще ще бъде добавена **поддръжка на български език**, която ще се осъществява чрез него.

            <div class="col-4 d-flex justify-content-center">

                <a class="btn btn-secondary nav-size me-3 p-2 ps-3 pe-3" style="border-radius: 30px !important;"><img src="Sources/Images/circle-fill.svg" class="mb-1">&nbspEN</a>

                <a href="@AppState.PageLink" class="btn nav-size btn-home p-2 ms-3 ps-3 pe-3" style="border-radius: 30px !important;">@AppState.ButtonState</a>

            </div>

Менюто се използва от две страници – **Encryption и Decryption**. Горният код реализира връзката между двете страници.

След отваряне на втората страница обаче същата връзка трябва да връща обратно към първата. **AppState** се използва, за да може съдържанието на бутона динамично да се променя спрямо отворената страница.

Извикването на даден Layout се случва в съответната страница:

@layout EncDecLayout

**Изпълнение на функционалност:**

Бутонът **“Encrypt”** в страницата за кодиране събира всички стойности и изпълнява кодирането

    private void Execute()

    {

        encrypted="";

        Enigma a = new Enigma(AppState.Rotor\_Config, rotor\_positions, plugs);

        foreach (char sym in original)

        {

            if(sym>='a' && sym<='z') encrypted+=char.ToLower(a.Encrypt(char.ToUpper(sym)));

            else if(sym>='A' && sym<='Z') encrypted+=a.Encrypt(sym);

            else encrypted+=sym;

        }

    }

Кодирането се изпълнява символ по символ, като малките букви биват превърнати в големи, а останалите символи се пропускат.

**Функционални елементи: класове Enigma, Rotor и Plugs**

1. **Enigma:**
2. class Enigma
3. {
4. int[] Reflector = { 24, 17, 20, 7, 16, 18, 11, 3, 15, 23, 13, 6, 14, 10, 12, 8, 4, 1, 5, 25, 2, 22, 21, 9, 0, 19 };
5. Rotor Rotor1,Rotor2, Rotor3;
6. public Plugboard plugboard;
7. public Enigma(int[] rotorConfig, int[] rotorPositions, string[] plugList)
8. {
9. Rotor1 = new Rotor(rotorConfig[0], rotorPositions[0]);
10. Rotor2 = new Rotor(rotorConfig[1], rotorPositions[1]);
11. Rotor3 = new Rotor(rotorConfig[2], rotorPositions[2]);
12. plugboard = new Plugboard(plugList);
13. }
14. public Enigma(int[] rotorConfig)
15. {
16. Rotor1 = new Rotor(rotorConfig[0]);
17. Rotor2 = new Rotor(rotorConfig[1]);
18. Rotor3 = new Rotor(rotorConfig[2]);
19. plugboard = new Plugboard();
20. }
21. public Enigma(Key key)
22. {
23. Rotor1 = new Rotor(key.rotor\_config[0], key.start\_position[0]);
24. Rotor2 = new Rotor(key.rotor\_config[1], key.start\_position[1]);
25. Rotor3 = new Rotor(key.rotor\_config[2], key.start\_position[2]);
26. plugboard = new Plugboard();
27. }
28. public void Rotate()
29. {
30. if(Rotor2.Notch())
31. {
32. Rotor2.TurnOver();
33. Rotor3.TurnOver();
34. }
35. if (Rotor1.Notch())
36. {
37. Rotor2.TurnOver();
38. }
39. Rotor1.TurnOver();
40. }
41. public char Encrypt(char element)
42. {
43. Rotate();
44. int trans = element - 65;
45. trans = plugboard.Connect(trans);
46. trans = Rotor1.Forward(trans);
47. trans = Rotor2.Forward(trans);
48. trans = Rotor3.Forward(trans);
49. trans = Reflector[trans];
50. trans = Rotor3.Back(trans);
51. trans = Rotor2.Back(trans);
52. trans = Rotor1.Back(trans);
53. trans = plugboard.Connect(trans);
54. return (char)(trans + 65);
55. }
56. public void SetPositons(int[] rotorPositions)
57. {
58. Rotor1.Position = rotorPositions[0];
59. Rotor2.Position = rotorPositions[1];
60. Rotor3.Position = rotorPositions[2];
61. }
62. }

Класът имплементира следните данни и методи:

1. Конструктори, които създават структурата и прехвърлят началните настройки на класовете Rotor и Plugboard. Всеки ротор е създаден с позиция и начално завъртане. Таблото може и да е празно
2. Масивът Reflector, който симулира отразителя. За вход приемаме индексите на елементите, а за изход – стойностите им
3. Метод Rotate позволява завъртането на роторите след всеки символ. Имплементирано е и двойното завъртане
4. Метод Encrypt превръща буквата в число от 0 до 26(опростява математиката във функциите), прекарва данните през всеки индивидуален елемент в ред:

**Табло > ротори(1, 2, 3) > Рефлектор > Ротори(3, 2, 1) > Табло**

Накрая отново превръщаме числото в буква и го извеждаме

1. **Rotor:**

Класът съдържа три атрибута: forward, reverse и position

private int[] forward, reverse;

private int position;

forward представя 26-те връзки вход-изход, а reverse представя изход-вход. Завъртането на ротора се представя чрез цялото число positon(0-25). За по-лесна работа с масивите използваме нулева индексация. Методите са следните:

1. Конструктор

public Rotor(int num, int pos)

{

switch (num)

{

case 1:

forward = RotorSelection.forward1;

reverse = RotorSelection.reverse1;

break;

case 2:

forward = RotorSelection.forward2;

reverse = RotorSelection.reverse2;

break;

case 3:

forward = RotorSelection.forward3;

reverse = RotorSelection.reverse3;

break;

case 4:

forward = RotorSelection.forward4;

reverse = RotorSelection.forward4;

break;

case 5:

forward = RotorSelection.forward5;

reverse = RotorSelection.reverse5;

break;

default:

throw new IndexOutOfRangeException("Invalid Rotor");

}

position = pos;

}

Конструкторът приема като параметри номер на ротора(1-5) и позиция(0-25). Статичният клас RotorSelection съдържа информация за свързването на всеки един от петте ротора, използвани от немската ENIGMА машина(модел Swiss K).Двата масива forward и reverse показват пътя между връзките съответно преди отразителя(вход-изход) и след него(изход-вход)

1. Forward

public int Forward(int element)

{

return (forward[(element + position) % 26] - position + 26) % 26;

}

Представя пътуването през ротора в посока вход-изход. Входовете на връзките са представени чрез индексите, а изходите – чрез елементите на масива. Чрез действията с position и 26 представяме ефекта на завъртането върху входа и изхода, като държим крайния резултат между 0 и 25

1. Back

public int Back(int element)

{

return (reverse[(element + position) % 26] - position + 26) % 26;

}

Имплементира връзката изход-вход. Процесът е същият, само че с масива reverse, който разменя позициите на индексите и елементите.

1. TurnOver – Завърта ротора с една позиция. Имплементира и връзка 26 => 1

public void TurnOver() => position = (position + 1) % 26;

1. Notch – проверява дали роторът е стигнал края си, за да знае дали да завърти следващия

public bool Notch() => position == 25;

1. **RotorSelection**

public class RotorSelection

{

public static int[] forward1 = { 15, 4, 25, 20, 14, 7, 23, 18, 2, 21, 5, 12, 19, 1, 6, 11, 17, 8, 13, 16, 9, 22, 0, 24, 3, 10 },

forward2 = { 25, 14, 20, 4, 18, 24, 3, 10, 5, 22, 15, 2, 8, 16, 23, 7, 12, 21, 1, 11, 6, 13, 9, 17, 0, 19 },

forward3 = { 4, 7, 17, 21, 23, 6, 0, 14, 1, 16, 20, 18, 8, 12, 25, 5, 11, 24, 13, 22, 10, 19, 15, 3, 9, 2 },

forward4 = { 8, 12, 4, 19, 2, 6, 5, 17, 0, 24, 18, 16, 1, 25, 23, 22, 11, 7, 10, 3, 21, 20, 15, 14, 9, 13 },

forward5 = { 16, 22, 4, 17, 19, 25, 20, 8, 14, 0, 18, 3, 5, 6, 7, 9, 10, 15, 24, 23, 2, 21, 1, 13, 12, 11 };

public static int[] reverse1 = { 22, 13, 8, 24, 1, 10, 14, 5, 17, 20, 25, 15, 11, 18, 4, 0, 19, 16, 7, 12, 3, 9, 21, 6, 23, 2 },

reverse2 = { 24, 18, 11, 6, 3, 8, 20, 15, 12, 22, 7, 19, 16, 21, 1, 10, 13, 23, 4, 25, 2, 17, 9, 14, 5, 0 },

reverse3 = { 6, 8, 25, 23, 0, 15, 5, 1, 12, 24, 20, 16, 13, 18, 7, 22, 9, 2, 11, 21, 10, 3, 19, 4, 17, 14 },

reverse5 = { 9, 22, 20, 11, 2, 12, 13, 14, 7, 15, 16, 25, 24, 23, 8, 17, 0, 3, 10, 4, 6, 21, 1, 19, 18, 5 };

}

Представя точната кабелировка на всеки от петте ротора. Данните са извадени от Wikipedia:

|  |  |
| --- | --- |
| **Нормална последователност:** | ABCDEFGHIJKLMNOPQRSTUVWXYZ |
| **Ротор 1:** | PEZUOHXSCVFMTBGLRINQJWAYDK |
| **Ротор 2:** | ZOUESYDKFWPCIQXHMVBLGNJRAT |
| **Ротор 3:** | EHRVXGAOBQUSIMZFLYNWKTPDJC |
| **Ротор 4:** | IMETCGFRAYSQBZXWLHKDVUPOJN |
| **Ротор 5:** | QWERTZUIOASDFGHJKPYXCVBNML |

1. **Plugboard**

Атрибути:

HashSet<int> usedChars;

Dictionary<int, int> maps;

usedChars – следи чифтосаните букви, за да не бъдат използвани пак

maps – представя самите връзки

Методи:

1. Конструктор – стандартен

public Plugboard()

{

usedChars = new HashSet<int>();

maps = new Dictionary<int, int>();

}

1. AddPlug – проверява, че форматът е правилен, буквите не са използвани и не е достигнат лимитът. След това добавя връзката към maps

public bool AddPlug(string conn)

{

if (Regex.IsMatch(conn, "^[A-Z]-[A-Z]$"))

{

int symbol1 = conn[0] - 65,

symbol2 = conn[2] - 65;

if (!(usedChars.Contains(symbol1) || usedChars.Contains(symbol2)))

{

usedChars.Add(symbol1);

usedChars.Add(symbol2);

maps[symbol1] = symbol2;

return true;

}

}

return false;

}

1. RemoveLastPlug – премахва последната връзка. Използван при декодирането

public void RemoveLastPlug()

{

usedChars.Remove(maps.LastOrDefault().Key);

usedChars.Remove(maps.LastOrDefault().Value);

maps.Remove(maps.LastOrDefault().Key);

}

1. Connect – ако буквата има чифт, го връща

public int Connect(int symbol)

{

if (usedChars.Contains(symbol))

{

if (maps.Values.Contains(symbol))

{

return maps.FirstOrDefault(x => x.Value == symbol).Key;

}

else

{

return maps[symbol];

}

}

return symbol;

**Същност на декодирането:**

Процесът на дешифриране се базира на два основни факта

1. Възможно е да се намерят настройките на роторите, без да се открият връзките. Това ни позволява през да преминем през 1,054,560 настройки на ротора, а чак след това да прегледаме 3,453,450 настройки на връзките единствено за правилните ротори. Следователно намаляваме броя възможности над **800** пъти:

**от 1,054,560\*3,453,450=3,641,870,232,000 до 1,054,560+3,453,450=4,508,010**

1. Процесът на замяна разпределя броят на всяка буква в азбуката сравнително поравно. В нашите езици обаче някои букви се използват по-често от други. Разликата може да се измери чрез т.нар. Index of Coincidence т.е. колко вероятно е две произволно избрани букви да са еднакви. Изчислява се чрез следното уравнение:

Където:

k – множеството на буквите от азбуката

ai – колко пъти се среща дадена буква в текста

n – общият брой букви

Стойността на уравнението при нормален английски текст е около 0,067. Следователно трябва просто да открием тази стойност за всяка конфигурация на роторите и да изведем най-близкият резултат до 0,067.

Внимание: методът изисква достатъчно дълъг и близък до нормалния английски текст. (аз лично го тествах с „да бъдеш или да не бъдеш“. Монологът съдържа доста архаизми, поетизми и стари форми, но системата за декодиране все пак сработи). Препоръчителна е дължина от поне 500-600 букви(монологът е с около 1100).

1. Quadgram scores– Горният метод работи за роторите, но е неподходящ за откриване на връзките. Затова анализираме и Quadgrams т.е. поредици от четири букви в текст. Във външен файл е описано колко е вероятно всяка от над 46,000 комбинации, от TION(най-често) до AOLI(най-рядко), да е част от английски текст. Използвайки тази информация, можем да намерим вероятността нашият текст да е част от английския, като намерим шанса всеки quadgram едновременно да се е появил. Toва представлява quadgram score. Например, за да намерим quadgram score на думата ATTACK:

p(ATTACK) = p(ATTA)\*p(TTAC)\*p(TACK)

Важен детайл: дори вероятността на най-честата комбинация от букви – TION, е едва 0,32%. Ако се опитаме да умножим всички quadgrams, ще получим неизчислимо малък резултат. Затова, вместо p(TION) взимаме log(p(TION)). Тъй като log(a)+log(b)=log(a\*b), умножението се свежда до събиране. Това е причината стойностите във файла да са отрицателни (log(x)<0 ако x<1).

Процес: за всяка комбинация от връзки намираме quadgram score на текста и накрая извеждаме конфигурацията при **най-голямата** стойност

**Имплементация:**

1. **Страница за декодиране – Decryption.razor**

**Graphical user interface, website

Description automatically generated**

* Тук се изпълнява разбиването на шифъра. Потребителя въвежда криптиран текст в горното текстово поле, избира метод за декодиране и натиска **„Start Algorithm“.** След около **6 минути** в долната кутия ще се появи конфигурацията, чийто резултат **най-близък** до стандартен английски текст.
* Ако обърнете внимание на навигационното меню, ще видите че текстът на бутона се е сменил, но останалите елементи са останали същите. Точно в това се състои функцията на **EncDecLayout.razor** и **AppState.**
* Чрез бутонът **“Frequency Analysis >”** потребителят може да избере между два метода за разбиване на шифъра. Засега само единия метод работи, а при смяна на метода се появява само запълващ текст.

1. **Изпълняване на алгоритъма**

**Процесът на декодиране се изпълнява в пет стъпки:**

**Base, inject, Handler, Decrypt, Quadgrams**

1. **Създаване на база**

Дешифриращият алгоритъм изпълнява голям брой операции, които бързо биха запълнили .razor файла. Затова отделям функционалността в т.нар. Base файл (DecryptionBase.cs). Всяка база наследява ComponentBase

public class DecryptionBase : ComponentBase

и след това бива извикана в **.razor** компонента

@inherits DecryptionBase

Методите на базата се извикват директно в **.razor** файла. **Не е нужна инициализация!**

1. **Събиране на информация от статични файлове**

Преди алгоритъмът да започне работа, трябва първо да извадя информацията за **quadgrams** oт външния текстови файл.

За да го отворя, първо създавам HttpClient чрез Dependency Injection

@inject HttpClient httpClient

Системата инициализира представителя **httpClient** автоматично.

В OnInitializedAsync изпълнявам GET Request за файла и след това прехвърлям съдържанието в базата чрез неин метод

        string fileLine = await httpClient.GetStringAsync("Sources/quadgrams.txt");

        SetFileLines(fileLine);

1. **Операции при натискане на бутона**

**Към @onclick на бутона е привързан следния метод**

    private async Task HandleDecrypt()

    {

        var watch = System.Diagnostics.Stopwatch.StartNew();

        working = "pe-none";

        plainText = await Task.Run(() => Decrypt(cipherText));

        working = "";

        watch.Stop();

        System.Console.WriteLine(watch.ElapsedMilliseconds);

    }

Определянето на метода като **async** позволява на визуалните елементи да останат интерактивни, дори когато функцията още върви. Повечето методи се изпълняват **достатъчно бързо** и следователно могат да бъдат изпълнени синхронично. Изпълнението на **HandleDecrypt** обаче отнема **няколко минути**, така че **употребата на асинхроничен метод е задължителна.**

1. **Изпълнение на същинските операции в DecryptionBase.cs**

        public static int[,] vars = { { 1, 2, 3 }, { 1, 2, 4 }, { 1, 2, 5 }, { 1, 3, 2 }, { 1, 3, 4 },

        { 1, 3, 5 }, { 1, 4, 2 }, { 1, 4, 3 }, { 1, 4, 5 }, { 1, 5, 2 }, { 1, 5, 3 }, { 1, 5, 4 },

        { 2, 1, 3 }, { 2, 1, 4 }, { 2, 1, 5 }, { 2, 3, 1 }, { 2, 3, 4 }, { 2, 3, 5 }, { 2, 4, 1 },

        { 2, 4, 3 }, { 2, 4, 5 }, { 2, 5, 1 }, { 2, 5, 3 }, { 2, 5, 4 }, { 3, 1, 2 }, { 3, 1, 4 },

        { 3, 1, 5 }, { 3, 2, 1 }, { 3, 2, 4 }, { 3, 2, 5 }, { 3, 4, 1 }, { 3, 4, 2 }, { 3, 4, 5 },

        { 3, 5, 1 }, { 3, 5, 2 }, { 3, 5, 4 }, { 4, 1, 2 }, { 4, 1, 3 }, { 4, 1, 5 }, { 4, 2, 1 },

        { 4, 2, 3 }, { 4, 2, 5 }, { 4, 3, 1 }, { 4, 3, 2 }, { 4, 3, 5 }, { 4, 5, 1 }, { 4, 5, 2 },

        { 4, 5, 3 }, { 5, 1, 2 }, { 5, 1, 3 }, { 5, 1, 4 }, { 5, 2, 1 }, { 5, 2, 3 }, { 5, 2, 4 },

        { 5, 3, 1 }, { 5, 3, 2 }, { 5, 3, 4 }, { 5, 4, 1 }, { 5, 4, 2 }, { 5, 4, 3 } };

        Dictionary<string, decimal> quad;

        private string[] fileLines;

        Key min;

        decimal hillClimb;

        string format;

public string Decrypt(string cipherText)

        {

            format = "";

            char decSym;

            min = new Key();

            Key temp;

            Enigma enigma;

            int[] histogram = new int[26];

            foreach (char sym in cipherText)

            {

                if (sym > 96 && sym < 123) format += (char)(sym - 32);

                else if (sym > 64 && sym < 91) format += sym;

            }

            for(int rotorConfigs=0;rotorConfigs<60;rotorConfigs++)

            {

                int[] rotor\_config = { vars[rotorConfigs, 0], vars[rotorConfigs, 1], vars[rotorConfigs, 2] };

                enigma = new Enigma(rotor\_config);

                for (int test3 = 0; test3 < 26; test3++)

                {

                    int[] positions = new int[3];

                    positions[2] = test3;

                    for (int test2 = 0; test2 < 26; test2++)

                    {

                        positions[1] = test2;

                        for (int test1 = 0; test1 < 26; test1++)

                        {

                            positions[0] = test1;

                            enigma.SetPositons(positions);

                            foreach (char sym in format)

                            {

                                decSym = enigma.Encrypt(sym);

                                histogram[decSym - 65]++;

                            }

                            temp = new Key(rotor\_config, positions, histogram);

                            if (temp.ioc > min.ioc) { min = temp; System.Console.WriteLine(min.ToString());}

                            histogram = new int[26];

                        }

                    }

                }

            }

            enigma = new Enigma(min);

            foreach (var item in format)

            {

                min.text += enigma.Encrypt(item);

            }

            return $"BEST - {DiscoverPlugs()}";

        }

Функция: свежда текстът до главни букви, преминава през всяко възможно нагласяне на роторите, изчислява Index of coincidence за всеки резултат и извежда конфигурацията с най-близък до 0,067 индекс

**Класът Key:**

Съдържа атрибути за номера и позиции на роторите(по подразбиране ротори 1, 2 и 3 с позиции 1, 1, 1), атрибут за Index of Coincidence, както и метод за изчисляването му чрез масив histogram. Масивът съдържа информация за абсолютната честота на всяка буква от азбуката в текста

1. **Hill Climbing:**
2. private string DiscoverPlugs()
3. {
4. Enigma enigma;
5. quad = new Dictionary<string, decimal>();
6. foreach (string line in fileLines)
7. {
8. string[] a = line.Split(' ');
9. quad[a[0]] = decimal.Parse(a[1]);
10. }
11. min.hillClimb = HillClimb(min.text);
12. for (char i = 'A'; i <= 'Z'; i++)
13. {
14. for (char j = (char)(i+1); j <= 'Z'; j++)
15. {
16. enigma = new Enigma(min);
17. string text = "";
18. if (enigma.plugboard.AddPlug($"{i}-{j}"))
19. {
20. foreach (var item in format)
21. {
22. text += enigma.Encrypt(item);
23. }
24. hillClimb = HillClimb(text);
25. if (hillClimb > min.hillClimb)
26. {
27. min.hillClimb = hillClimb;
28. min.plugboard.Add($"{i}-{j}");
29. min.text = text;
30. }
31. enigma.plugboard.RemoveLastPlug();
32. }
33. }
34. }
35. return min.ToString();
36. }
37. public decimal HillClimb(string text)
38. {
39. decimal Climb = 0;
40. for (int k = 0; k < text.Length - 4; k++)
41. {
42. string four = text.Substring(k, 4);
43. if (quad.ContainsKey(four)) Climb += quad[four];
44. else Climb += -6;
45. }
46. return Climb;
47. }

Функция: имплементира изчисляването на quadgram scores, познато още като Hill Climbing. Методът минава през всички възможности за индивидуална връзка, включително липсата на такава, закодира текста чрез тях и проверява дали резултатът се е подобрил.

Функцията HillClimb извършва логическата работа. Тя преминава през всеки 4 последователни букви(quadgrams) и събира вероятностите им

Линии 4-9: прехвърлят информацията за quadgrams речник quad със структура { quadgram: вероятност }

Атрибут min е от типа Key и е получен след действията с роторите. Тази част от кода се пуска само след като роторите са намерени

Методът RemoveLastPlug(ред 31), който се грижи, че след всеки опит конфигурацията се връща в начално състояние

Ред 45: -6 е вероятността по подразбиране за всички комбинации от 4 букви, които не са във файла.

**Ahead of Time Compilation:**

Тъй като C# обикновено е употребяван **в сървъра, а не в client-а,** компилаторът **интерпретира Blazor кода в WebAssembly**, преди да го изпълни. В следствие всеки **C#** оператор се изпълнява около **три пъти по-бавно** от обикновено. При процесът на декодиране, който изпълнява **милиони цикли** с оператори, това вдига времето за изпълнение **от 6 до 20 минути**. Този проблем се решава чрез т.нар. **Ahead of Time Compilation** т.е. интерпретацията се извършва **предварително** и готовият файл се зарежда. В този случай **размерът** на основния файл скача от около **5MB до 16**, но за сметка на това операторите вървят с **нормална скорост.** Използването на АОТ се определя **в .csproj файла**

    <RunAOTCompilation>true</RunAOTCompilation>

**Hosting:**

AOT компилацията се изпълнява единствено след публикуване на проекта. Получените от това файлове обаче не могат да бъдат активирани сами по себе си, а вместо това биват качени на платформа за хостване. Аз лично използвам GitHub Pages, като съм автоматизирал процеса чрез следния GitHub Action.

name: Deploy to GitHub Pages

# Run workflow on every push to the master branch

on:

  push:

    branches: [ main ]

jobs:

  deploy-to-github-pages:

    # use ubuntu-latest image to run steps on

    runs-on: ubuntu-latest

    steps:

    # uses GitHub's checkout action to checkout code form the master branch

    - uses: actions/checkout@v3

    # sets up .NET Core SDK 6.0.406

    - name: Setup .NET Core SDK

      uses: actions/setup-dotnet@v3

      with:

        dotnet-version: 6.0.406

    #Installs wasm-tools

    - name: Install wasm-tools

      run: dotnet workload install wasm-tools

    # publishes Blazor project to the release-folder

    - name: Publish .NET Core Project

      run: dotnet publish PublishDemo.csproj -c Release -o release --nologo

    # changes the base-tag in index.html from '/' to 'PublishDemo' to match GitHub Pages repository subdirectory

    - name: Change base-tag in index.html from / to PublishDemo

      run: sed -i 's/<base href="\/" \/>/<base href="\/PublishDemo\/" \/>/g' release/wwwroot/index.html

    # copy index.html to 404.html to serve the same file when a file is not found

    - name: copy index.html to 404.html

      run: cp release/wwwroot/index.html release/wwwroot/404.html

    # add .nojekyll file to tell GitHub pages to not treat this as a Jekyll project. (Allow files and folders starting with an underscore)

    - name: Add .nojekyll file

      run: touch release/wwwroot/.nojekyll

    - name: Commit wwwroot to GitHub Pages

      uses: JamesIves/github-pages-deploy-action@v4

      with:

        token: ${{ secrets.PAT\_TOKEN }}

        BRANCH: gh-pages

        FOLDER: release/wwwroot

**Функция:** при всяко push действие GitHub публикува, форматира и качва файла за Hosting

**Особености:**

1. **Install wasm-tools:** wasm-tools за нужни за изпълнението на АОТ. Тяхното инсталиране трябва да се случи в самият код, а не върху самият компютър, тъй като действията се изпълват на виртуална машина.
2. **Change base tag:** Нормалният <base> на index.html. създава следния URL при хостване: https://{име на акаунт}.github.io/

Обаче нужният URL е: https://{име на акаунт}.github.io/{repository}

Директната промяна на файла би развалила локалното изпълнение на проекта(чрез localhost). Затова промяната се изпълнява тук

1. **.nojekyll:** в.NET структурата се използват папки, които започват с **\_.** Те биват игнорирани от GitHub, който обикновено използва Jekyll. Добавянето на **.nojekyll** файл решава този проблем
2. **PAT\_TOKEN:** Автоматичната обработка на repository изисква допълнителни права. Те се дават чрез т.нар. Personal Access Token, който се създава за целия акаунт и после бива вграден в дадено repository под формата на Secret. PAT\_TOKEN е името на тази тайна. То дава достъп до токена и разширените права.

**Заключение:**

Приложението може да кодира текст със съответен шифър и да декодира кодирания текст. Развит е атрактивен потребителски интерфейс, който позволява лесен достъп до различните функции на проекта.

**Срещнати трудности:**

Отвъд редицата малки спънки, свързани с имплементацията на всеки софтуерен проект, имаше два основни проблема, които трябваше да реша.

1. **Представяне на механичен процес чрез софтуерни оператори**

Целият процес на кодиране и декодиране на машината се базира единствено на пътя на електричеството през веригата. Машината в нито един момент не борави с данни или изчисления. Това значи, че трябваше да намеря начин да представя механичната работа на уреда чрез програмни структури и оператори. Изпълнението на тази задача изискваше сериозно обмисляна на отделните елементи и как си взаимодействат. Например, за да представим правилно завъртането на роторите, трябваше да вземем в предвид как то променя входа и изхода, но и взаимодействието със съседните ротори, които също имат завъртане. Друг пример е вече споменатото двойно завъртане, при което специфичният дизайн на уреда води до особености, които трябва да се намерят и имплементират.

1. **Проблеми с оптимизацията и скоростта**

Този проблем се появи чак при декодирането. Тъй като трябва да анализираме текстове от стотици символи при милиони различни настройки, дори малките пропуски в оптимизацията много бързо се натрупват. В един момент, поради прекалено много такива пропуски, преглеждането само на 1000 различни конфигурации на роторите отнемаше около 5 минути. За да реша този проблем, направих пълна преработка на кода, като го организирах по принципите на ООП т.е. в класове(вижте разликите между първият и вторият commit в GitHub, ако се заинтересовани). След това направих преглед на употребата на ресурсите от паметта и процесора, за да забележа и премахна претоварващите методи. В момента, за същото време(6-7 минути) алгоритъмът анализира всички 1,054,560 различни настройки.

**По-нататъшно развитие:**

**Използвана литература:**

Използвах главно онлайн ресурси, всички на английски.

Връзки към по-важните:

* [Структура на машината](https://www.youtube.com/watch?v=ybkkiGtJmkM)
* [Данни за дизайна на роторите](https://en.wikipedia.org/wiki/Enigma_rotor_details)
* [Информация за Index of Coincidence](https://www.youtube.com/watch?v=kty-dCB4AAk)
* [Practical Cryptography – всякаква информация за дешифриране на кодове](http://practicalcryptography.com/)
* [Обяснение и имплементация на Hill Climbing](http://practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams/)

Връзка към GitHub страница - <https://github.com/TheFichoza/ENIGMA.git>