M Ű E G Y E T E M 1 7 8 2
Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Kvantuminformatika és - kommunikáció
Házi feladat

Kvantumkeresés alkalmazása

Horváth Patrik
Kabucz Áron
Szakmáry Szabolcs

Tartalom

Bevezetés

A hagyományos számítógépek binárisan működnek, ez azt jelenti, hogy az adatok bitekben vannak tárolva. Egy-egy ilyen hagyományos bit értéke 0 vagy 1 lehet. Ezzel szemben a kvantumszámítógépek kvantumbitekkel dolgoznak a hagyományosbitek helyett. Ezen kvantumbitek képesek a 0 és 1 állapot szuperpozíciójában lenni. Ezt a tulajdonságot kihasználva a kvantumkeresés során könnyű a párhuzamosíthatóság. Ezek alapján belátható, hogy a kvantum kereséshez szükséges idő jelentősen kevesebb lehet, mint hagyományos esetben.

Első körben bemutatásra kerül néhány hagyományos rendező és kereső algoritmust, majd az ezekhez tartozó programkód részlet és teszt eredmények. Ezek után következik az RSA, Grover-, és Shor-algoritmusok ismertetése. Végül az előbbiekben említett algoritmusokhoz tartozó szimuláció. Fontos azonban kiemelni, hogy mivel a kvantuminformatika egy még nagyon gyorsan fejlődő terület, illetve a fizikai korlátozások erősen akadályozzák a kvantumszámítógépek elterjedését, így a szimulációk nem szolgáltatnak teljesen valós eredményeket.

1. Hagyományos kereső algoritmusok

Néhány hagyományos, gyakran használt kereső algoritmus:

1. Lineáris keresés
2. Bináris keresés
3. Interpolációs keresés
4. Exponenciális keresés
5. Hash-alapú keresés
6. Dijkstra algoritmus

1.1. Lineáris keresés:

Egy tömb elemein végig haladunk, amíg megtaláljuk a keresett elemet. A tömb sorszámát adjuk vissza. Maximum N lépés. Átlagosan: (N+1)/2.

1.2. Bináris keresés:

Ez az eljárás már egy rendezett tömbön való keresést valósít meg. A cél megtalálni egy adott értékű elem sorszámát, amennyiben az érték szerepel a tömbben. Az eljárás elve, hogy a keresett értékű számot összehasonlítjuk a tömb közepén lévő elemmel, amelynek három kimenetele lehet. Amennyiben az pont megegyezik a keresett értékkel, akkor kész vagyunk. Ha ez a középen lévő érték nagyobb, mint a keresett szám, akkor már csak a tömb „első felében" folytatjuk ugyanezzel a módszerrel a keresést. A harmadik lehetőség pedig az, hogy a szám kisebb, ekkor a tömb „második felében" folytatjuk. Az eljárás lépésszáma: $\log_2(N)$.

1.3. Interpolációs keresés:

Az interpolációs keresés egy speciális esete az előbb bemutatott bináris keresésnek. Ez az algoritmus feltételezi, hogy a tömbben szereplő értékek eloszlása egyenletes. A keresés a keresett elem helyzetét közelíti a tömbben, annak első és utolsó értékének aránya alapján. A futási idő nagyban függ az eloszlástól. Ideális esetben: $\log_2(\log_2(n))$, legrosszabb esetben azonban: n.

1.4. Exponenciális keresés:

Az exponenciális keresés is a bináris keresésre épít, annak előnyeit igyekszik minél jobban kihasználni. Ez az algoritmus először meghatároz egy tartományt, majd ezen végez bináris keresést. A tartomány meghatározása egy indexszel történik a következőképpen: addig növeli az indexet lépésenként a kétszeresére, amíg az vagy meg nem haladja az egész tömb elemszámát, vagy amíg legalább el nem éri a tömb index-szedik értéke a keresett számot, ebből ered az exponenciális elnevezés. Ezt követően meghívásra kerül a bináris keresés, melynek intervalluma az index felétől az indexig, vagy amennyiben az már meghaladná a tömb maximális elemszámát, akkor az elemszámig történik.

1.5. Hash-alapú keresés:

A keresés során hash függvényeket használ az adatok indexeléséhez. A hash függvény olyan matematikai művelet, amelynek eredménye, hogy minden objektumnak különböző hash kódja lesz. Ezeket a hash kódokat tárolja a hash tábla. Amennyiben a táblában mégis ütközés lenne, akkor ezeket kezelnünk kell, például láncolt listát alkalmazva. A hash függvények használatával nagyon gyorsan lehet keresni a táblában. A keresési idő általában 1(állandó).

1.6. Dijkstra lgoritmus:

A Dijkstra algoritmus egy gráfelméletben használatos kereső algoritmus. Segítségével akár irányított vagy irányítatlan gráf esetén is megtalálható a két pont közti legrövidebb út. A futás során a gráf kiindulási csúcsától kezdődően minden csúcshoz nyilvántartjuk az addigi legkisebb út költségeket, amíg végül elérünk a célcsúcshoz.

Az előbbiekben néhány kereső algoritmus lett bemutatva. Az egyszerű implementálhatóság figyelembevételével a Hash-alapú és a Dijkstra algoritmusoktól most tekintsünk el. A másik négy fajta keresés könnyen megvalósítható, például C++ programozási nyelvben, amelyre a későbbiekben ki is fogunk térni. A lineáris keresés bemenete nem feltétlenül kell egy rendezett tömb legyen, az egy véletlen szám megtalálásának idején nem változtat. Ezzel ellentétben a bináris keresés és az erre épülő interpolációs, illetve exponenciális keresés esetén azonban már rendezett tömbökön alkalmazható algoritmusok. Ebből érezhető, hogy bár ezek jóval gyorsabb futási idővel rendelkeznek, de ehhez a tömböt elsőkörben rendeznünk kell, így a következőkben a rendező algoritmusokról lesz szó.

Néhány hagyományos, gyakran használt rendező algoritmus:

1. Buborékrendezés
2. Kiválasztásos rendezés
3. Beszúrásos rendezés
4. QuickSort
5. MergeSort

2. Rendező algoritmusok

2.1. Buborékrendezés

Az algoritmus a tömb elemein egymásután sokszor végighalad, mindaddig, amíg az elvártnak megfelelően lesz rendezve a tömb. Az egyes iterációk során az algoritmus egy előre meghatározott reláció alapján összehasonlítja a szomszédos elemeket és azok sorrendjét megcseréli, ha azok nem felelnek meg a relációnak. Amennyiben a reláció egy növekvő sorrendet határoz meg akkor a rendezés során először a legnagyobb elem kerül a kívánt (azaz a tömb utolsó elemébe) helyre, tehát a következő lépésben már nem kell a tömb teljes hosszán végig menni, elég csak az utolsó előtti elemig vizsgálni és esetlegesen cserélni az elemeket. Azonban a sok iterációnak köszönhetően ez a rendezési algoritmus nem igazán hatékony. [(N-1)*N]/2 lépésből áll a tömb sorba rendezése.

2.2. Kiválasztásos rendezés:

Alapvetően hasonló a buborékrendezéshez ez az algoritmus is. Létezik minimumkiválasztásos és maximumkiválasztásos rendezés is. A minimumkiválasztásos rendezés során megkeressük a tömb legkisebb elemét és a kicseréljük a tömb első elemével, így a következő lépésben már elég egyel kisebb elemszámon végeznünk a keresést. A maximumkiválasztásos rendezés esetén természetesen a legnagyobb elemet cseréljük ki a tömb legutolsó elemével. (N^2)/2 lépésből áll átlagosan.

2.3. Beszúrásos rendezés:

A beszúrásos rendezés esetén a k-adik lépést megelőzően már sorba lett rendezve a tömb k-1 eleme. Ezt követően a k-adik elemet már be tudjuk szúrni a rendezett k-1 elemű sorba úgy, hogy a rendezettsége megmarad, a k-adik elemnél nagyobb értékű elemek sorszámát pedig 1-egyel meg kell növeljük. Az algoritmus lépésszáma: N^2.

2.4. QuickSort:

A gyorsrendezés algoritmus rekurzív. Egy konkrét értékhez (pivotelem) hasonlítjuk az elemeket. Amennyiben a viszonyítási érték nagyobb az elemnél az elemet elé tesszük, ha kisebb, akkor mögé. A viszonyítási értéket általában célszerű a középső értéknek választani. Erre a két részre mindaddig megismételjük ezt a lépést, amíg az elemszám nem lesz egy vagy nulla. Az algoritmus átlagos futási ideje: N*log(N).

2.5. MergeSort:

A quicksort algoritmushoz hasonlóan ez is rekurzív, azonban a mergesort kiküszöböli a gyorsrendezés esetén esetlegesen fellépő legrosszabb esetet, amikor a futási idő N^2 ként adódik. A mergesort esetén a rendezni kívánt tömböt felosztjuk N darab résztömbre, ezek értelemszerűen rendezett tömbök lesznek, ugyanis csak egy elemük van. Ezt követően minden lépésben kettő, már rendezett tömböt összeillesztünk a rendezés során. Az algoritmus végén visszakapjuk egy tömbbe az eredeti tömbünket, azonban már rendezve. A futási idő minden esetben: N*log(N).

3. Tesztelés

3.1 Rendező algoritmusok tesztelése

Bár alapvetően a kereső algoritmusokról szól ez a fejezet, láthatjuk, hogy ezek használatához a legtöbb esetben elengedhetetlen egy rendező algoritmus használata is. Ebből kifolyólag az algoritmusok tesztelése során elsőkörben a rendező függvények futási ideje lett lemérve.

3.1.1. Mérési elrendezés

A tesztelésre szolgáló programkód a Microsoft Visual Studio 2022 fejlesztőkörnyezetében lett C++ programozási nyelven implementálva. Az egyes függvények megvalósítása részben, vagy szinte teljesen a www.geeksforgeeks.org weboldalon található kódok alapján készültek. A futási idők mérésére egy Intel Core I5-4460 típusú processzort használtunk, amelynek alap frekvenciája 3,2 GHz.
A rendezések egy 10000db random egész számot tartalmazó tömbön lettek futtatva mindegyik esetben. A fejlesztőkörnyezet által biztosított optimalizációk közül 3 különböző opció mellett lettek lemérve a függvények végrehajtási ideje, ezek a következőek: Od (kikapcsolt optimalizáció), O1 (maximális optimalizáció – méretre), O2 (maximális optimalizáció – sebességre). Az egyes optimalizációk mellett minden algoritmus esetén 5-5 idő eredmény lett rögzítve, majd ezen értékek átlagolva lettek. A mérési eredmények mikroszekundumban lettek rögzítve.

3.1.2. Buborékrendezés

A buborékrendezést megvalósító programkód részlet:

```
//bubble sort
void bubbluesort(int arr[], int n)
{
	int i, j;
	bool swapped;
	for (i = 0; i < n - 1; i++) {
		swapped = false;
		for (j = 0; j < n - i - 1; j++) {
			if (arr[j] > arr[j + 1]) {
				swap(arr[j], arr[j + 1]);
				swapped = true;
			}
		}
		if (swapped == false)
			break;
```

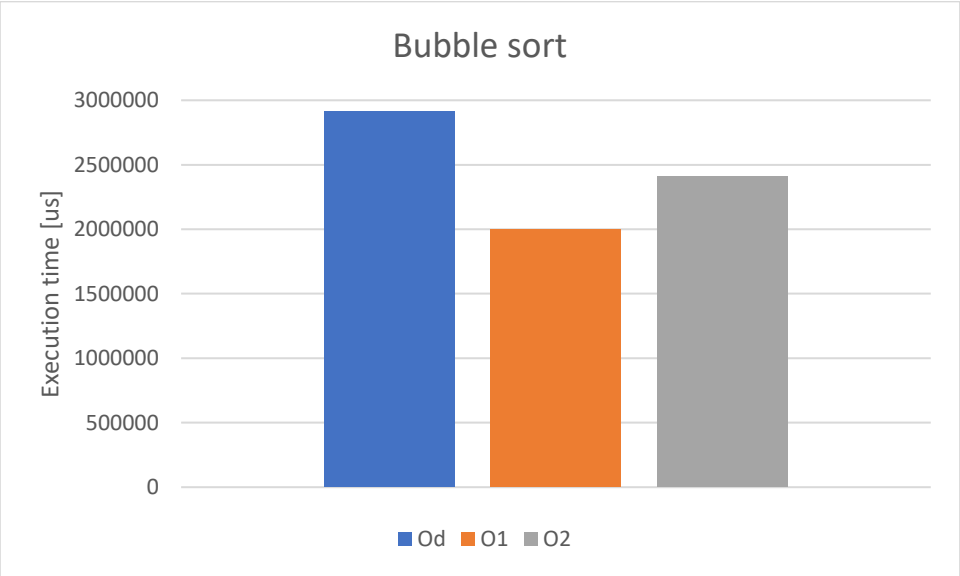
```
        }
}
```

A teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Teszt 1	Teszt 2	Teszt 3	Teszt 4	Teszt 5	Átlag
Bubble sort	Od	3282520	2867193	2562355	2855785	2991174	2911805
	O1	1771142	2303144	1758155	2086901	2067222	1997313
	O2	2346547	3063800	2498482	1757054	2373961	2407969

1. táblázat - Buborékrendezés

Jól látható, hogy jelen esetben az optimalizálások használata egész jelentős teljesítménybeli növekedést is jelenthet, továbbá az is megfigyelhető, hogy az egyes optimalizációs szintekhez tartozó eredmények szórása viszonylag nagy, ez minden bizonnyal a számítógépen futó különböző háttérfolyamatoknak tudható be. Ezen megállapítások a többi algoritmus teszteredményeire is igazak többé-kevésbé.
Az optimalizációs szintekhez tartozó átlageredményeket ábrázoló diagram:



1. ábra - Buborékrendezés

3.1.3. Kiválasztásos rendezés

A kiválasztásos rendezést megvalósító programkód részlet:

```
//selection sort
void selectionsort(int arr[], int n)
{
        int i, j, indexminimum;

        for (i = 0; i < n - 1; i++)
        {
                indexminimum = i;
                for (j = i + 1; j < n; j++)
                        if (arr[j] < arr[indexminimum])
                                indexminimum = j;
```

```
                swap(arr[indexminimum], arr[i]);
            }
}
```

A teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Teszt 1	Teszt 2	Teszt 3	Teszt 4	Teszt 5	Átlag
Selection sort	Od	675502	516139	435840	335816	623559	517371,2
	O1	760599	656153	751350	720978	555635	688943
	O2	603579	587619	571069	473250	307771	508657,6

2. táblázat - Kiválasztásos rendezés

Az előző esethez képest a kiválasztásos rendezés során pont, hogy az O1 optimalizáció mellett nyújtotta a leglassabb teljesítményt a program, míg a másik két esetben közel azonos eredmény született.
Az optimalizációs szintekhez tartozó átlageredményeket ábrázoló diagram:

2. ábra - Kiválasztásos rendezés

3.1.4. Beszúrásos rendezés

A beszúrásos rendezést megvalósító programkód részlet:

```
//insertion sort
void insertionsort(int arr[], int n)
{
	int i, key, j;
	for (i = 1; i < n; i++) {
		key = arr[i];
		j = i - 1;

		while (j >= 0 && arr[j] > key) {
			arr[j + 1] = arr[j];
			j = j - 1;
		}
		arr[j + 1] = key;
	}
}
```

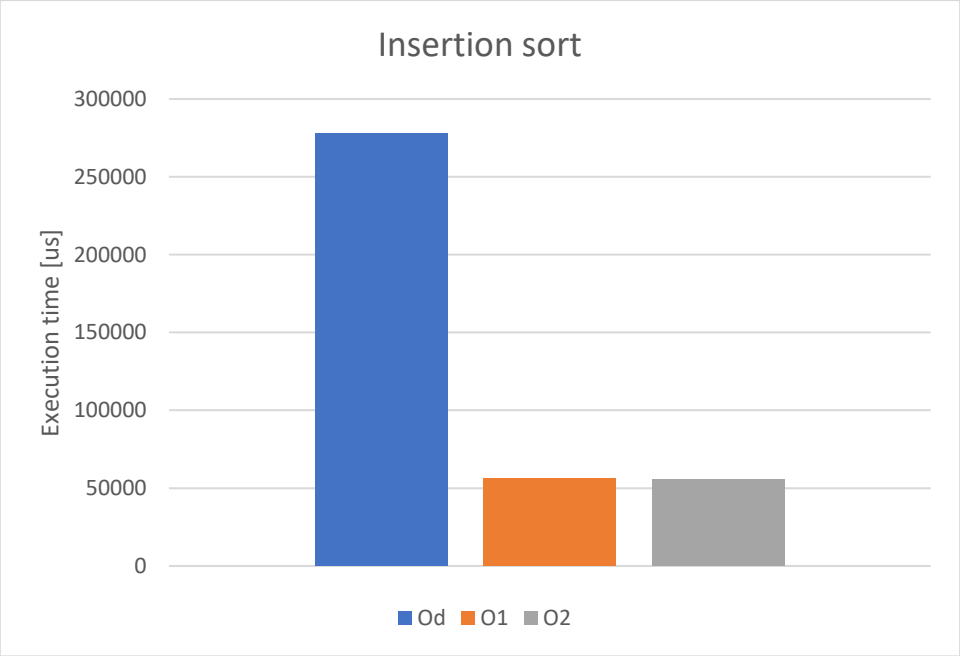
A teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Teszt 1	Teszt 2	Teszt 3	Teszt 4	Teszt 5	Átlag
Insertion sort	Od	136250	364379	361414	192992	334203	277847,6
	O1	38904	44861	80248	65908	51158	56215,8
	O2	28192	75511	52743	63943	59349	55947,6

3. táblázat - Beszúrásos rendezés

Az optimalizációs szintek közötti eltérés ebben az esetben volt a legjelentősebb. Míg az O1 és O2 szintekhez tartozó átlageredmény nagyjából megegyezik, addig az Od-hez tartozó majdnem ötszöröse a másik kettőének.

Az optimalizációs szintekhez tartozó átlageredményeket ábrázoló diagram:



3. ábra - Beszúrásos rendezés

3.1.5. Gyorsrendezés (quicksort)

A gyorsrendezést megvalósító programkód részlet:

```
//quick sort
int partition(int arr[], int start, int end)
{

	int pivot = arr[start];

	int count = 0;
	for (int i = start + 1; i <= end; i++) {
		if (arr[i] <= pivot)
			count++;
	}

	int pivotIndex = start + count;
	swap(arr[pivotIndex], arr[start]);

	int i = start, j = end;

	while (i < pivotIndex && j > pivotIndex) {

		while (arr[i] <= pivot) {
```

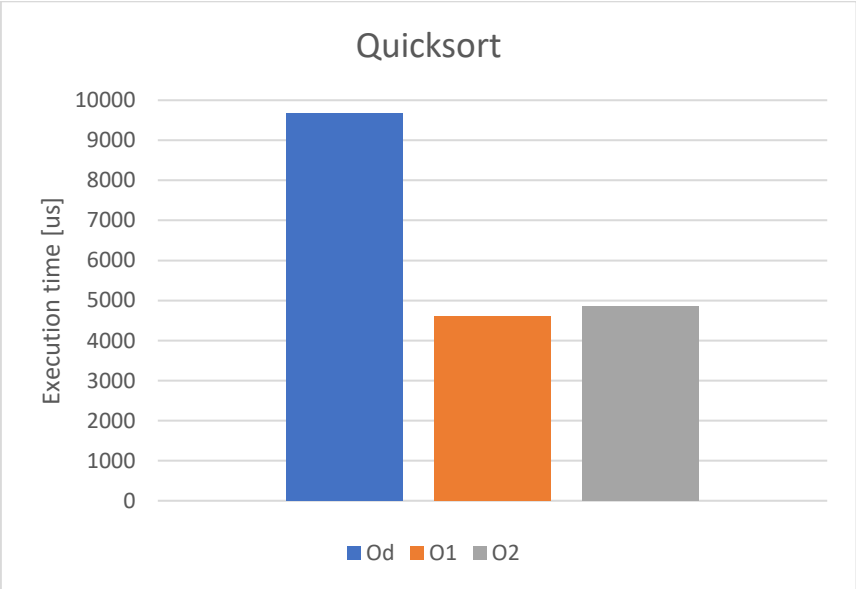
```
                        i++;
                }

                while (arr[j] > pivot) {
                        j--;
                }

                if (i < pivotIndex && j > pivotIndex) {
                        swap(arr[i++], arr[j--]);
                }
        }

        return pivotIndex;
}

void quicksort(int arr[], int start, int end)
{
        if (start >= end)
                return;

        int p = partition(arr, start, end);

        quicksort(arr, start, p - 1);
        quicksort(arr, p + 1, end);
}
```

A teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Teszt 1	Teszt 2	Teszt 3	Teszt 4	Teszt 5	Átlag
Quicksort	Od	10427	9955	10100	8145	9795	9684,4
	O1	5497	3262	3241	5666	5393	4611,8
	O2	4879	4630	4918	4857	4960	4848,8

4. táblázat - Gyorsrendezés

A megvizsgált rendező algoritmusok közül ugyanazon tömbre a gyorsrendezés bizonyult a leggyorsabbnak még kikapcsolt optimalizáció mellett is.
Az optimalizációs szintekhez tartozó átlageredményeket ábrázoló diagram:



4. ábra – Gyorsrendezés

3.1.6. Összefuttatásos rendezés (merge sort)

Az összefuttatásos rendezést megvalósító programkód részlet:

```
//merge sort
void merge(int array[], int const left,
	int const mid, int const right)
{
	auto const subArrayOne = mid - left + 1;
	auto const subArrayTwo = right - mid;

	auto* leftArray = new int[subArrayOne],
		* rightArray = new int[subArrayTwo];

	for (auto i = 0; i < subArrayOne; i++)
		leftArray[i] = array[left + i];
	for (auto j = 0; j < subArrayTwo; j++)
		rightArray[j] = array[mid + 1 + j];

	auto indexOfSubArrayOne = 0,
		indexOfSubArrayTwo = 0;

	int indexOfMergedArray = left;

	while (indexOfSubArrayOne < subArrayOne &&
		indexOfSubArrayTwo < subArrayTwo)
	{
		if (leftArray[indexOfSubArrayOne] <=
			rightArray[indexOfSubArrayTwo])
		{
			array[indexOfMergedArray] =
				leftArray[indexOfSubArrayOne];
			indexOfSubArrayOne++;
		}
		else
		{
			array[indexOfMergedArray] =
				rightArray[indexOfSubArrayTwo];
			indexOfSubArrayTwo++;
		}
		indexOfMergedArray++;
	}

	while (indexOfSubArrayOne < subArrayOne)
	{
		array[indexOfMergedArray] =
			leftArray[indexOfSubArrayOne];
		indexOfSubArrayOne++;
		indexOfMergedArray++;
	}

	while (indexOfSubArrayTwo < subArrayTwo)
	{
		array[indexOfMergedArray] =
			rightArray[indexOfSubArrayTwo];
		indexOfSubArrayTwo++;
		indexOfMergedArray++;
	}
}

void mergesort(int array[],
	int const begin,
	int const end)
{
	if (begin >= end)
		return;

	auto mid = begin + (end - begin) / 2;
```

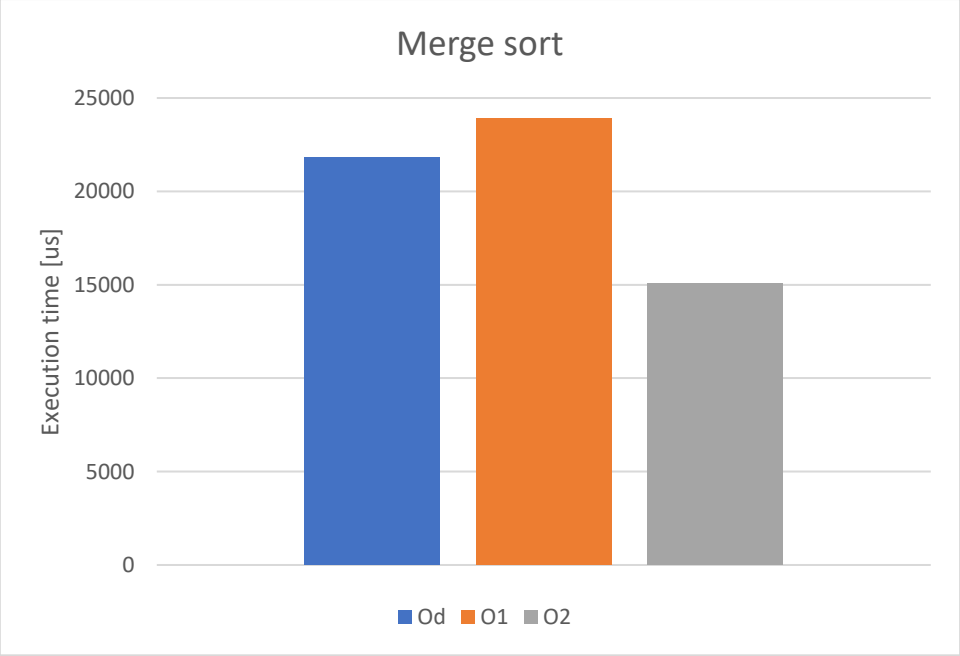
```
    mergesort(array, begin, mid);
    mergesort(array, mid + 1, end);
    merge(array, begin, mid, end); }
```

A teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Teszt 1	Teszt 2	Teszt 3	Teszt 4	Teszt 5	Átlag
Merge sort	Od	21765	25307	17161	23613	21185	21806,2
	O1	19662	30045	21187	21275	27432	23920,2
	O2	10354	18001	17716	19856	9400	15065,4

5. táblázat - Összefuttatásos rendezés

Az optimalizációs szintekhez tartozó átlageredményeket ábrázoló diagram:



5. ábra - Összefuttatásos rendezés

3.2. Kereső algoritmusok tesztelése

Most, hogy már láttunk pár példát, hogy hogyan és mekkora végrehajtási idő mellett rendezhetőek sorba különböző algoritmusok használatával tömbök, ideje ezen rendezett tömbökön való keresésre rátérni.

3.2.1. Mérési elrendezés

A rendező algoritmusok teszteléséhez hasonlóan a kereső algoritmusokat is a Microsoft Visual Studio 2022-ben próbáltam megvalósítani. Ezen függvények gyorsasága és a számítógépben található processzor viszonylag magas alapfrekvenciájának köszönhetően azonban túlságosan gyorsan lefutottak, így általában 0 és 2 mikroszekundum közötti eredményt kaptam a futási időre.

A második próbálkozás során egy STM32F207ZG mikrokontrollert használtam a vizsgálatra. Fejlesztőkörnyezetként a kártya gyártója által biztosított STM32CubeIDE-t választottam. A tesztelések során még mindig viszonylag kis értékeket kaptam. A minél nagyobb pontosság elérése érdekében átállítottam a rendszer órajelét a legkisebbre, amely hardveresen engedélyezett volt, ez 25 MHz. Bár lineáris keresésre már egész nagy futási időket kaptam, éreztem, hogy a gyorsabb algoritmusok esetén még ez az órajel frekvencia is túlnagy lehet.

A végső megoldást az jelentette, hogy egy még gyengébb processzorral rendelkező mikrokontrollert választottam, ami nem más volt, mint az STM32F446RE kártya. Ezzel a hardver választással már egészen 4 MHz-ig tudtam csökkenteni az órajel frekvenciáját.

A használt fejlesztőkörnyezet is biztosít többféle optimalizációt. Ezek közül a következőkkel teszteltem a futási időket: O0 (kikapcsolt optimalizáció), O1 (alacsony szintű optimalizáció), Of (magas szintű optimalizáció – sebességre).
A mikrokontrolleren mért adatok a különböző optimalizációk és tömb méretek mellett konzisztensek voltak, ez természetesen abból adódik, hogy míg egy hagyományos számítógép processzora nem csak a tesztet futtató programkód végrehajtásával foglalkozik, addig a mikrokontroller csak azt a kódot futtatja, amire fel van programozva.
A méréseket három különböző (1000, 10000, 30000) elemszámú tömbre végeztem. Mindhárom esetben a tömb elemei egyenletes eloszlásúak és előre rendezettek voltak, hogy a kis teljesítményű processzort ne terheljem a rendezés megvalósításával. Az időket most is mikroszekundumban mértem és rögzítettem.

3.2.2. Lineáris keresés

A lineáris keresést megvalósító programkód részlet:

```
//linear search
int linearsearch(int arr[],
        int n, int x)
{
        int i;
        for (i = 0; i < n; i++)
                if (arr[i] == x)
                        return i;
        return -1;
}
```

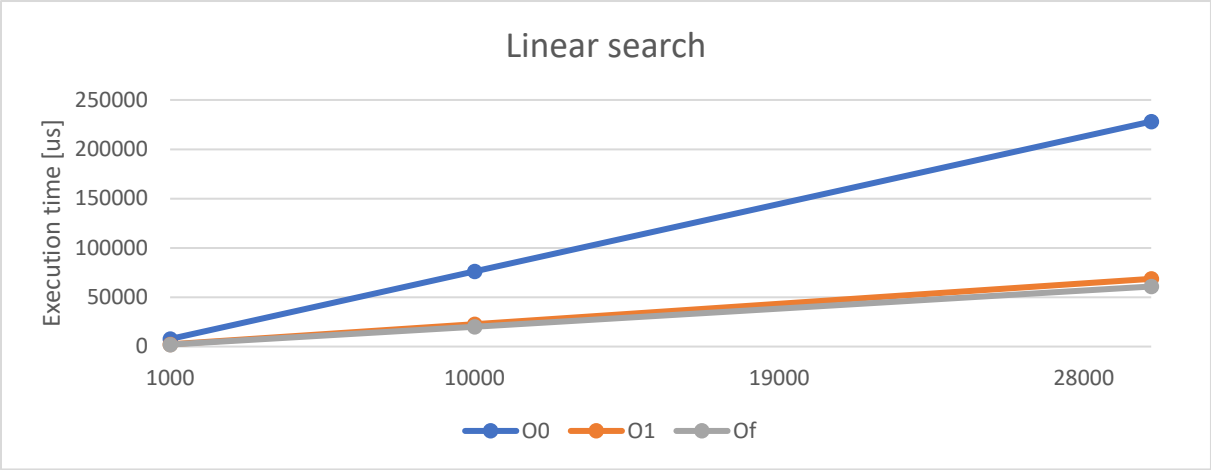
A teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Tömb elemszáma: 1000	Tömb elemszáma: 10000	Tömb elemszáma: 30000
Linear search	O0	7646,25	76132,25	228106,75
	O1	2298	22770	68646
	Of	2031	20175	60871

6. táblázat - Lineáris keresés

Annak érdekében, hogy a kereső függvény maximális ideig fusson a keresett számot nem tartalmazta a tömb, amin a keresés futott. Jól látható, hogy a futási idő nagyjából lineárisan nőtt a tömb elemszámának függvényében. Az algoritmus ismerete mellett ez nem meglepő, minél több számot kell megvizsgálni, annál több ideig tart a végrehajtás. Az arányok a különböző optimalizációk mellett is megmaradtak, azonban jóval gyorsabb volt a végrehajtás, ha nem kikapcsolt optimalizációval lettek futtatva a tesztek.

Az optimalizációs szintekhez tartozó eredményeket ábrázoló diagram:



6. ábra – Lineáris keresés

3.2.3. Bináris keresés

A bináris keresést megvalósító programkód részlet:

```
//binary search
int binarysearch(int arr[], int low, int high, int x)
{
	if (high >= low) {
		int mid = low + (high - low) / 2;

		if (arr[mid] == x)
			return mid;

		if (arr[mid] > x)
			return binarysearch(arr, low, mid - 1, x);

		return binarysearch(arr, mid + 1, high, x);
	}
	return -1;
}
```

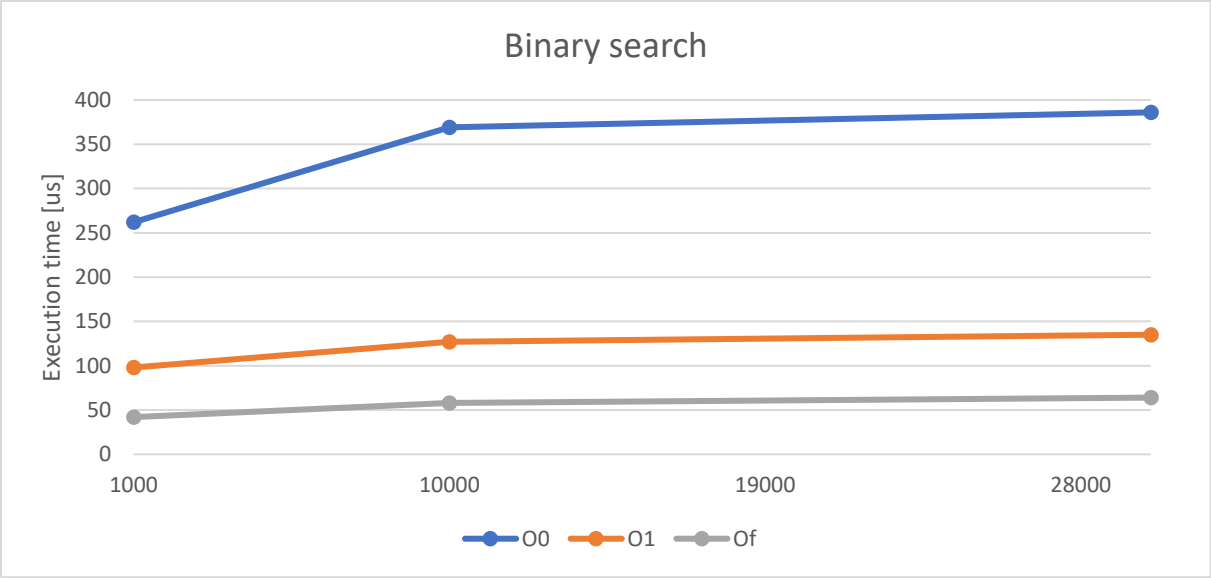
A teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Tömb elemszáma: 1000	Tömb elemszáma: 10000	Tömb elemszáma: 30000
Binary search	O0	262	369	386
	O1	98	127	135
	Of	42	58	64

7. táblázat - Bináris keresés

A keresett számot itt is olyan értékre állítottam, amely nem volt eleme a tömbnek, ugyanis ebben az esetben kell a program a legtöbb összehasonlítást végrehajtsa. Amennyiben egy olyan számot kerestem, amely eleme volt a tömbnek, akkor a futási idő valamivel alacsonyabb volt a legtöbb esetben. Az optimalizációs szintek a bináris keresés esetén is nagyban hozzájárulnak a gyors futási időkhöz. Az elemszám függvényében nyilvánvalóan ebben az esetben is nőnek a mért értékek, de korántsem olyan mértékben, mint a lineáris keresés esetén. Bár ahhoz, hogy egyértelműen látszódjon a futási idő görbéje kevés három pontra illeszteni azt, de érezhető, hogy leginkább a logaritmus függvényére hasonlít a következő ábrán.

Az optimalizációs szintekhez tartozó eredményeket ábrázoló diagram:



7. ábra - Bináris keresés

3.2.4. Interpolációs keresés

Az interpolációs keresést megvalósító programkód részlet:

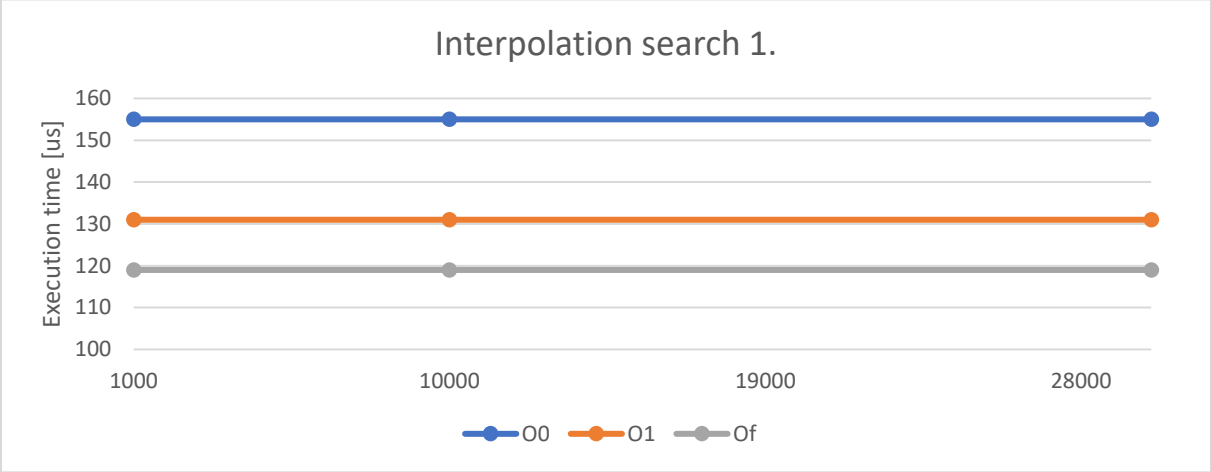
```
//interpolation search
int interpolationsearch(int arr[], int low, int high, int x)
{
	int pos;
	if (low <= high && x >= arr[low] && x <= arr[high]) {
		pos = low
			+ (((double)(high - low) / (arr[high] - arr[low]))
				* (x - arr[low]));

		if (arr[pos] == x)
			return pos;

		if (arr[pos] < x)
			return interpolationsearch(arr, pos + 1, high, x);

		if (arr[pos] > x)
			return interpolationsearch(arr, low, pos - 1, x);
	}
	return -1;
}
```

Az interpolációs keresés esetén a tesztelést két különböző esetre osztottam. Az első eset, amikor a keresett elem része a tömbnek, a második esetben viszont nem volt abban megtalálható. Mindkét esetben gyorsan lefutottak a függvények, ez persze annak is köszönhető, hogy a tömbben szereplő értékek eloszlása egyenletes volt.
Első esetben a teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Tömb elemszáma: 1000	Tömb elemszáma: 10000	Tömb elemszáma: 30000
Interpolation search 1.	O0	155	155	155
	O1	131	131	131
	Of	119	119	119

8. táblázat - Interpolációs keresés 1

Bár a teszt eredményei alapján úgy tűnik, hogy a futási idő a tömb elemszámától független, azonban ez minden bizonyára nincs így. A tesztelés lehetőségeit erősen korlátozták a hardver fizikai paraméterei, ennek köszönhetően nem tudtam nagyságrendekkel nagyobb tömbön futtatni a méréseket. Mindenesetre abban biztosak lehetünk az eredményeket látva, hogy nagyon lassan nő a végrehajtási idő, amely megfeleltethető az általános ismertetésben leírtakkal.
Első esetben az optimalizációs szintekhez tartozó eredményeket ábrázoló diagram:



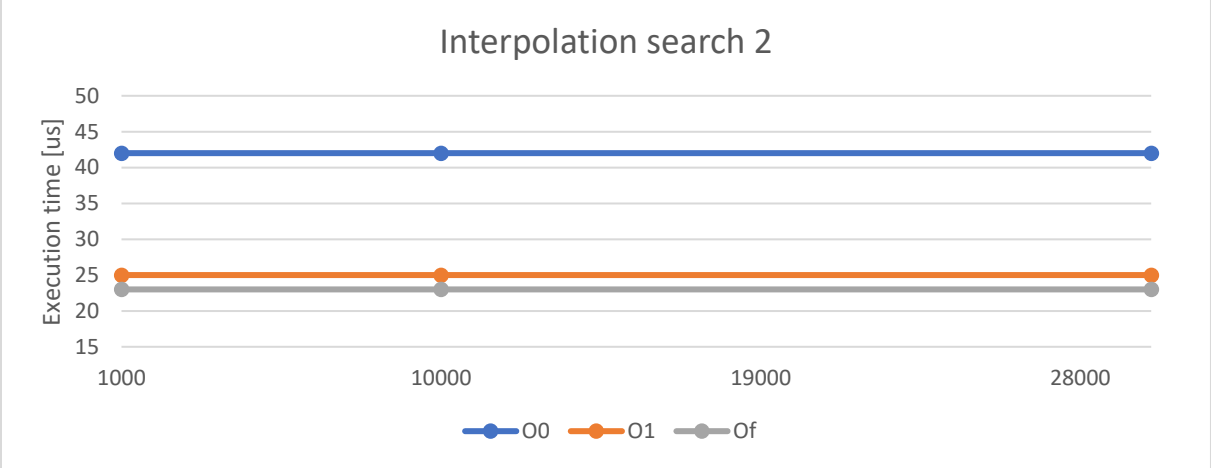
8. ábra - Interpolációs keresés 1

Második esetben a teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Tömb elemszáma: 1000	Tömb elemszáma: 10000	Tömb elemszáma: 30000
Interpolation search 2.	O0	42	42	42
	O1	25	25	25
	Of	23	23	23

9. táblázat - Interpolációs keresés 2

Az első esetben leírtak természetesen itt is igazak. A különbség mindössze annyi, hogy a keresett szám a tömb határértékein kívülre esett, így azok arányaiból gyorsan meg tudta határozni a függvény, hogy nem része a tömbnek.
Második esetben az optimalizációs szintekhez tartozó eredményeket ábrázoló diagram:



7. ábra - Interpolációs keresés 2

3.2.5. Exponenciális keresés

Az interpolációs keresést megvalósító programkód részlet:

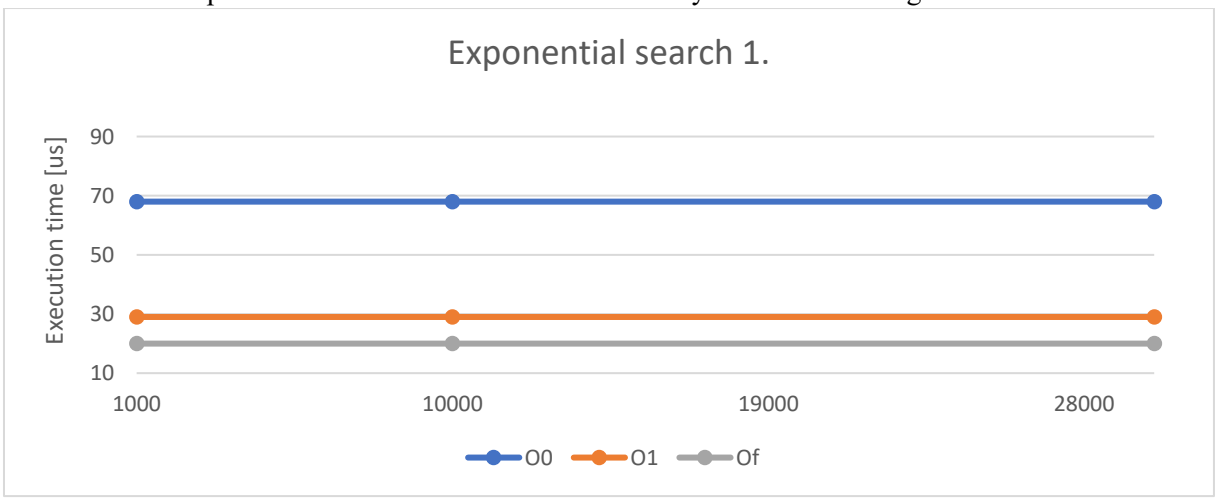
```
//exponential search
int exponentialsearch(int arr[], int n, int x)
{
	if (arr[0] == x)
		return 0;

	int index = 1;
	while (index < n && arr[index] <= x)
		index = index * 2;

	return binarysearch(arr, index / 2,
		min(index, n - 1), x);
}
```

Az interpolációs kereséshez hasonlóan az exponenciális keresést is két különböző esetre osztottam fel. Az első esetben a keresett szám ugyancsak része a tömbnek, viszont itt még van egy plusz kikötés, ami szerint a tömb elején kell elhelyezkedjen az elem. Második esetben pedig nem lehet része a tömbnek az elem. Az első eset plusz kikötésére azért volt szükség, mert ezen keresési algoritmus során a futási idő nagyban függ attól, hogy a tömb melyik részén van az elem.
Első esetben a teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Tömb elemszáma: 1000	Tömb elemszáma: 10000	Tömb elemszáma: 30000
Exponential search 1.	O0	68	68	68
	O1	29	29	29
	Of	20	20	20

10. táblázat - Exponenciális keresés 1

A keresett szám tömbben elfoglalt sorszáma mindegyik tömb méretnél azonos volt, így az algoritmusnak ugyanazon összehasonlításokat kellett elvégeznie, amíg megtalálta a számot. Ebből adódik, hogy a futási idők megegyeznek a különböző tömb méretekre.
Első esetben az optimalizációs szintekhez tartozó eredményeket ábrázoló diagram:



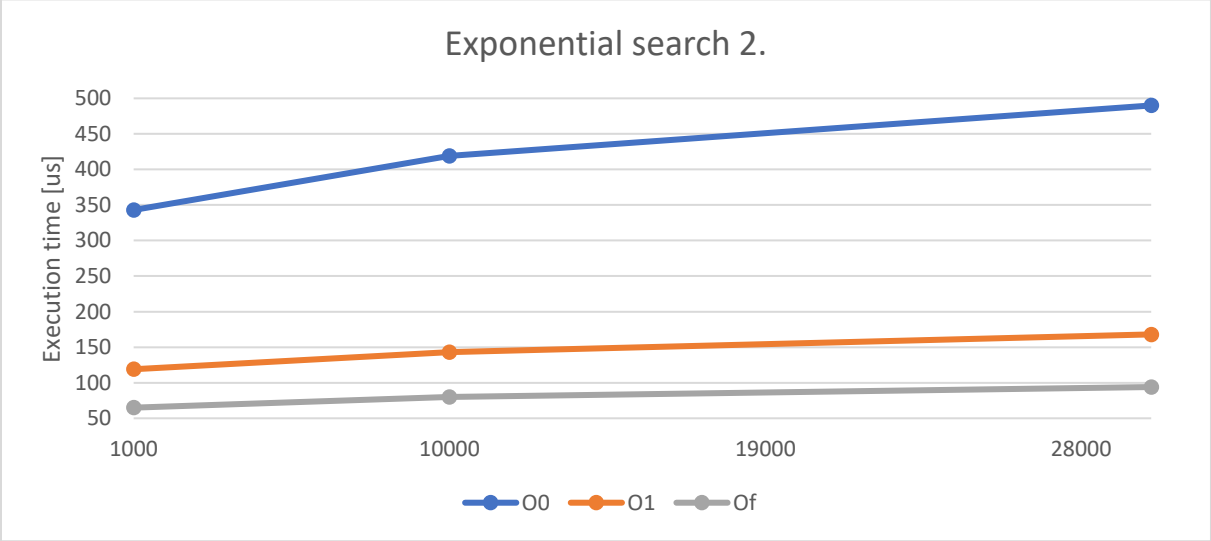
8. ábra - Exponenciális keresés 1

Második esetben a teszt eredményeit tartalmazó táblázat:

	Optimalizáció	Tömb elemszáma: 1000	Tömb elemszáma: 10000	Tömb elemszáma: 30000
Exponential search 2.	O0	343	419	490
	O1	119	143	168
	Of	65	80	94

11. táblázat - Exponenciális keresés 2

Ebben az esetben már természetesen az elemszám növekedésével nőtt a futási idő is. Ez annak köszönhető, hogy mivel a keresett elem nem volt része a tömbnek, így az egyre nagyobb egyre több összehasonlítás futott le, ami megnöveli a végrehajtás idejét.
Második esetben az optimalizációs szintekhez tartozó eredményeket ábrázoló diagram:



9. ábra - Exponenciális keresés 2

4. A kriptográfia és az RSA – algoritmus:

4.1. A kriptográfia

A kriptográfia, másnéven titkosítás vagy rejtjelezés egy több ezer éves múltra visszatekintő tudomány. Célja nem más, minthogy valamilyen „írott” szöveget úgy alakítson át, hogy az illetéktelen személyek számára visszafejthetetlen, vagy kvázi – visszafejthetetlen legyen. A kriptográfiai algoritmus az az eljárás, mellyel során a titkosítás történik. Egy titkosítás akkor tekinthető jónak, ha a „visszafejthetetlensége” mellett rendelkezik azzal a tulajdonsággal is, hogy a jogosult személy(ek) valamilyen kulcs birtokában képesek a rejtjelezett üzenetet hatékonyan és egyértelműen visszakapni. A kriptográfiával körülbelül egyidejű tudomány a kriptoanalízis, mely azzal foglalkozik, hogy a titkosított üzeneteket bármilyen kulcs, vagy jogosultság nélkül visszafejtse vagy feltörje.

4.1.1. A szimmetrikus kulcsú titkosítás

A szimmetrikus kulcsú titkosítás legfőbb jellemzője, hogy ebben az esetben a titkosításhoz és a visszafejtéshez használt kulcs megegyezik. Ennek eredményeképpen ezt a kulcsot feltétlenül titokban kell tartani. Ez a titkosító eljárások legegyszerűbb módja. A régebbi titkosító eljárások ezen az alapon működtek. Ilyen például a betűeltolásos titkosítás, a DES

(1976) (vagy későbbi változata a 3DES), az AES (1997) vagy az IDEA (1991). Ennek a fajta rejtjelezésnek nagy előnye az egyszerűsége, hátránya azonban az, hogy mivel a titkosító és visszafejtő kulcs közös, ezért azt a kommunikáció előtt meg kell osztani a kommunikáló felek között, ami egy nagyfokú biztonsági kockázatot jelent. További nehézség az, hogy a kommunikáció összes résztvevőjének különböző kulcsot kell biztosítani, hiszen amennyiben ezek megegyeznének, kölcsönösen el tudnák olvasni egymás üzeneteit.További érdekesség, hogy a második világháborúban a Harmadik Birodalom által használt Enigma nevet viselő titkosító berendezés is ilyen elven működött. Mint azt ez az eset is mutatja, a számítógépek megjelenésével, majd ezek teljesítményének növekedésével ezek a fajta kódok viszonylag egyszerűen feltörhetővé váltak.

4.1.2. Az asszimmetrikus (vagy nyílt) kulcsú titkosítás

Mivel a szimmetrikus kulcsú titkosítási eljárások nem nyújtottak elegendő biztonságot, szükség volt újfajta eljárások kidolgozására. 1976-ban Diffie és Hellman publikált egy új titkosítási eljárást, melynek lényege az volt, hogy a kommunikáció minden résztvevője két, nem azonos kulccsal rendelkezik. A két kulcs (jellemzően e-vel és d-vel jelöljük őket) a nyilvános (vagy publikus) kulcs (az e-vel jelölt), míg a d-vel jelölt kulcs a titkos (vagy privát) kulcs. Fontos megjegyezni, hogy ez az titkosítási eljárás úgy van kidolgozva, hogy a privát kulcs ismeretében a visszafejtés gyors, azonban a publikus kulcsból a privát kulcs meghatározása „nagyon nehéz”, ideális esetben lehetetlen. De mit jelent az, hogy nagyon nehéz? Ideális esetben ez azt jelentené, hogy bizonyítottan nem létezik olyan algoritmus, mely képes a problémát (jelen esetben a titkos kulcs visszafejtését a nyilvános kulcsból) polinomiális időben megoldani. Sokszor előfordul azonban, hogy bár nem ismerünk olyan algoritmust amire ez igaz lenne, nem bizonyított a tény, hogy nem is létezik ilyen algoritmus. De miért baj az, hogy a titkos kulcs nem visszafejthető polinomiális időben? Ez azt jelenti, hogy a kulcs ugyan visszafejthető, de megfelelően bonyolult kulcsok és titkosítási eljárás választása esetén ez észszerűtlenül sok időt venne igénybe. Valamint, hogy a kulcsok hosszának növelésével a számítási idő exponenciális jelleggel növekszik. Ilyen elven működik napjaink egyik leggyakrabban használt titkosító algoritmusa, az RSA algoritmus is.

4.2. Az RSA algoritmus

Az RSA – algoritmus napjaink leggyakrabban használt titkosítási eljárásaink egyike. Használják többek között internetes kommunikáció titkosítására, digitális aláírások létrehozására, illetve az elektronikus tranzakciók során is. Legfőbb jellemzője, hogy nyílt kulcsú, más néven aszimmetrikus.

4.2.1. Története

Az eljárást 1977-ben publikálta Ron Rivest, Adi Shamir és Len Adleman. Az RSA a nevük kezdőbetűiből áll össze.

4.2.2. Alapja

Az eljárás a nagy számok prímfelbontásának nehézségére alapszik. A számelmélet alaptétele alapján minden 1-nél nagyobb pozitív egész szám felbontható prímszámok szorzatára, és ez a felbontás a sorrendtől eltekintve egyértelmű. A problémát az adja, hogy nagy számok esetén jelenleg nem ismerünk minden esetben hatékony algoritmust a prímfelbontásra. Azonban adott hosszúságú számok közül is vannak melyek könnyebben és vannak melyek nehezebben bonthatók fel prímszámok szorzatára. A tudomány jelenlegi

állása szerint két, közel azonos nagyságú prímszám szorzataként előálló számok felbontása a legnehezebb, nem véletlen, hogy az RSA algoritmusban is igyekszünk így választani számokat.

4.2.3. Működése

A használt RSA kód egyik legfontosabb tulajdonsága a használni kívánt modulus hossza (N). Ez a hossz határozza meg a kódoláshoz, és ami még fontosabb: a feltöréshez szükséges számítások idejét. Ez a kezdetben 128 bit volt, azonban ez a technika fejlődésével és a számítógépek teljesítményének növekedésével már nem bizonyult biztonságosnak. A kódolás biztonságát ennek a számnak a növelésével igyekeztek fenntartani, hiszen ahogy korábban említettük, jelen tudásunk szerint nincs olyan algoritmus, mely a prímtényezőkre bontást polinomiális időben végezni tudja. Ennek következményeképpen a modulus hosszának növelése exponenciálisan növeli a feltöréshez szükséges időt. A későbbiekben a modulus hossza volt 256 bit, majd 512 bit is, napjainkban azonban már az 1024 bites modulus hossz az elterjedt, ami 309 decimális számjegynek felel meg. A modulus hosszának növelése olyan szempontból hátrányos, hogy ezáltal a titkosítási és (a kulcs birtokában) dekódolási fázis számításigénye is megnő.
A titkosítás első lépése az, hogy választunk két, közel azonos nagyságú (N/2 bit hosszúságú) prímszámot, általában p-vel és q-val jelöljük őket. Ezek után előállítjuk a φ(N) számot, mely φ(N)=(p-1) * (q-1) módon áll elő. Ezután választunk két egész számot (jellemzően e-t és d-t), melyekre igaz, hogy e*d ≡ 1 mod φ(N). Azaz, hogy az e*d szorzat kongruens legyen 1-gyel modulo φ(N). Ez azt jelenti, hogy az e*d szorzat és az 1 φ(N) ugyanazon maradékosztályába tartoznak. Az így kapott e-t nyilvános exponensnek, míg a d-t privát exponensnek nevezzük. Az (N,e) páros a publikus kulcs, míg az (N,d) páros a privát kulcs. A privát kulcsot kihirdetjük, ennek a segítségével tudnak számunkra üzenetet küldeni, míg a privát kulcsot titokban tartjuk, ezzel tudjuk a nekünk küldött kódolt üzeneteket visszafejteni. Az üzenetküldés úgy zajlik, hogy fogjuk az M üzenetet, mely leképezhető valamilyen egész számra, majd elvégezzük a $C = M^e \bmod N$ számítást, mellyel előáll a titkosított C üzenet. Ezek után a C üzenetet elküldik nekünk valamilyen csatornán. A hozzánk beérkezett kódolt üzenetből mi pedig az $M = C^d \bmod N$ számítás segítségével visszakapjuk az eredeti (M) üzenetet. A kódolás és dekódolás közötti kapcsolat matematikai hátterében az Euler – Fermat tétel áll. De mi történik, amikor valaki elfogja a nekünk küldött titkosított üzenetet?
A nyilvános kulcsunkat mindenki ismerheti, ezért a N és az e számok ismertek az illetéktelenek számára is. Azonban a visszafejtéshez szükség van a d úgynevezett privát exponensre is. A d számításhoz azonban szükséges a φ(N) ismerete, ami, mint p és q szorzata áll elő. Ezeket a számokat az illetéktelen lehallgatók azonban csak úgy kaphatják meg, amennyiben képesek N-t faktorizálni. Ez pedig az a probléma, amelyről korábban megállapítottuk, hogy korántsem egyszerű, sőt „nagyon nehéz”. Már-már annyira nehéz, hogy a gyakorlatban legtöbbször megpróbálni sem érdemes. Azonban van néhány körülmény, melyet, ha figyelmen kívül hagyunk, jelentősen csökkenthetjük a kódolásunk biztonságát, és megkönnyíthetjük támadóink dolgát.

4.2.4. Az RSA támadások

A támadásokat alapvetően két nagy csoportba oszthatjuk, léteznek az úgynevezett implementáció függő támadások, melyek csak adott helyzetben alkalmazhatóak, illetve vannak a helytelen alkalmazáson alapuló támadások is, ahol a kódoló fél nem kellő körültekintése vezet a kódolás biztonságának csökkenéséhez. Nézzünk ezek közül néhányat, a teljesség igénye nélkül.

4.2.4.1. Implementáció függő támadások

Ezen típusú támadások közös jellemzője, hogy az elméleti megfontolásokon kívül, komoly technikai feltételekkel is rendelkeznek.

A **kulcskereséses támadás** alapja, hogy amennyiben egy, a privát kulcsot használó szervert támadunk. A szerver memóriájában nyilvánvalóan jelen kell lennie a privát a kulcsnak, de ez az idő nagy részében kódoltan tárolódik. Azonban amikor a szerver a kulcsot használja, akkor kell lennie olyan időszakoknak, amikor a kulcs kódolatlanul a memóriában van. Amennyiben a támadó el tudja érni, hogy a rendszer a megfelelő pillanatban összeomoljon, akkor nem maradt más feladata, mint hogy megkeresse a kulcsot a memóriában (vagy, amennyiben ahhoz nincs hozzáférése, akkor a memória dump-ban). Ez azonban egy nagy méretű adathalmaz, ami elméletben nagyon megnehezítené a keresést. Azonban, mivel a kulcsokat általában álvéletlenszám generátorral generálják, az elkészült kulcs olyan adat, melynek entrópiája igen nagy. Ennek az információnak a birtokában már jóval egyszerűbb megtalálni a memóriában a megfelelő, 1024 bites bitsorozatot.

A **számítási idő mérésén alapuló támadások** abban az esetben kerül előtérben, amikor az RSA műveletek végzése közben a kulcs nem kerül ki a rendszerből. Ilyenek például a smarcardokon alapuló alkalmazások. Itt nyilvánvalóan nem férünk hozzá a rendszer memóriájához sem. Ilyenkor egy lehetőség annak a mérése, hogy a kártyának mennyi időbe telik egy RSA művelet. Ehhez hasonló eljárás, mikor nem az időt mérjük, hanem a rendszer áramfelvételét, majd ebből következtetünk, az elvégzett műveletekre.

4.2.4.2. Helytelen alkalmazáson alapuló támadások

Ezek a támadások arra alapulnak, hogy a kódolást végző személy, vagy rendszer nem kellő körültekintéssel jár el, a kezdeti prímszámok megválasztásakor. Ennek egyik példája az az eset, amikor p és q számok ikerprímek (azaz q = p + 2), ebben az esetben a feltörés rendkívül egyszerűvé válik. Helytelen alkalmazások esetén, bár az előálló modulus mérete ezt nem indokolná, mégis könnyen feltörhetővé válik a kódolásunk. Nézzünk néhány példát a helytelen alkalmazásokra:

Közös modulus: Előfordulhat, hogy a rendszer úgy szeretne erőforrásokat spórolni, hogy nem generál minden résztvevőnek saját modulust, hanem ugyanahhoz a modulushoz generál e és d értékeket. Ez azért veszélyes, mert matematikailag belátható, hogy N faktorizációja, és a privát kulcs ismerete egyenértékű. Ezáltal a kommunikáció minden résztvevője kiszámíthatja más résztvevők privát kulcsát is, csökkentve a rendszer biztonságát.

Kis privát exponens használata: A számítási igény csökkentése érdekében néhány alkalmazásban (jellemzően a gyengébb hardware-eken futókban) kis méretű privát exponenst (d) használnak. Igaz, hogy ezáltal a számítási idő jelentősen csökkenthető, viszont matematikailag megmutatható, hogy amennyiben d értéke eléggé kicsi, abban az esetben a kód feltörhetővé válik. Jelen állás szerint $d \geq N^{0,292}$ a biztonsági határ.

A modulus faktorizációján alapuló támadások: A támadások ezen alfaja a publikus kulcs ismeretében N faktorizációjával igyekszik megfejteni a privát exponenst. A ma ismert legjobb algoritmus a Number Field Sieve algoritmus melyet 1993-ban publikáltak. A jelenlegi nagyteljesítményű célgépek

segítségével már az 512 bites RSA kulcsok is biztonsági kockázatnak vannak kitéve.

4.2.4.3. Kvantumalgoritmusok használata

Ahogy azt korábban láthattuk, megfelelő körültekintéssel, és a modulus hosszának folyamatos növelésével az RSA kódolás bár nem bizonyítottan, de védve van a klasszikus támadásokkal szemben. Nincs kizárva, hogy a jövőben találnak olyan klasszikus algoritmusokat, amelyek ismeretében ez már nem lesz igaz, azonban ez egyelőre nincs kilátásban. Éppen ezért a RSA kódolásra jelenleg a legnagyobb fenyegetést a kvantumszámítógépek és a kvantumalgoritmusok jelentik. Ezen algoritmusok a Lov Grover indiai-amerikai informatikus 1996-ban publikált Grover algoritmus és a Peter Shor által 1994-ben publikált Shor algoritmus.

5. A kvantumkeresés

Az emberiség történelme során mindig is nagy szerepet játszott a rendezetlen adatbázisokban való keresés. Elegendő csak arra gondoljuk, hogy a bennünket körülvevő környezetre lehet, mint erőforrások rendezetlen adatbázisa tekinteni, melyben a megtalálni kívánt erőforrás a keresett adat. A rendezetlen adatbázisokban történő keresés azonban meglehetősen nehézkes. A két alkalmazható keresési algoritmus a véletlen keresés, illetve a kimerítő keresés. Egy N elemű adatbázis esetén azonban mindkettő módszerben legrosszabb esetben N lépés kell, hogy a keresett elemet (adatot) megtaláljuk (amennyiben véletlen keresés esetén nem választunk olyan véletlen elemet, amelyet már választottunk és a keresett elem csak egyszer szerepel az adatbázisban). A számítógépek megjelenésével tovább nőtt az adatbázisokban való keresés hatékonyságát növelő módszerek iránti igény. A született klasszikus megoldások azonban feltételeznek valamilyen rendet az adatbázisunkban, ezért ezen keresőalgoritmusok hatékony működéséhez megfelelő rendező algoritmusokra is szükség van. A rendezetlen keresés hatékonyságának növelésére egészen a kvantumos algoritmusok megjelenéséig kellett várni.

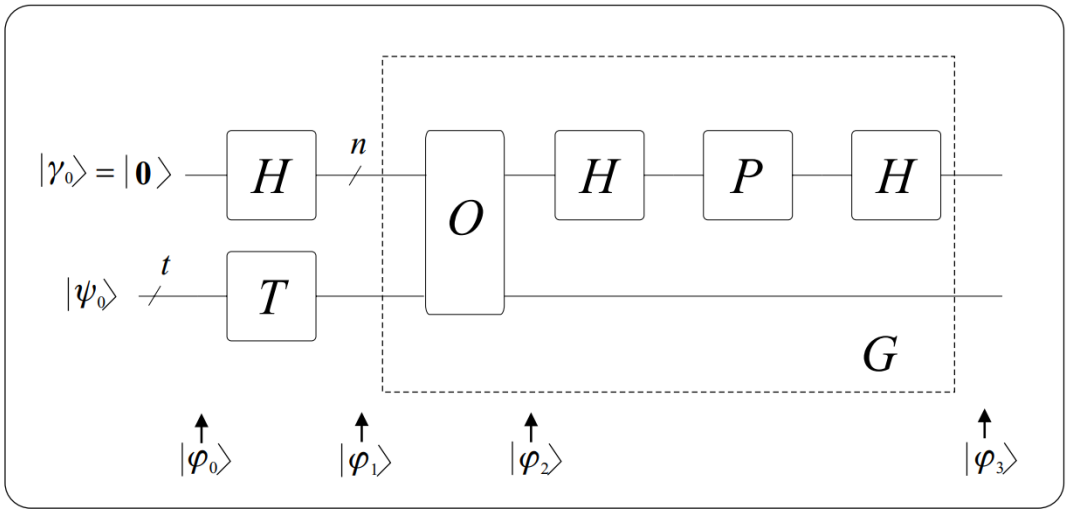
5.1. A Grover algoritmus

A Grover algoritmus 1996-os publikálása nagyon csábító eredményeket vetített előre. Az addigi maximális N lépés helyett csupán $\sqrt{N}$ maximális lépés szükségességét vetítette előre rendezetlen adatbázisokban való keresés esetén.

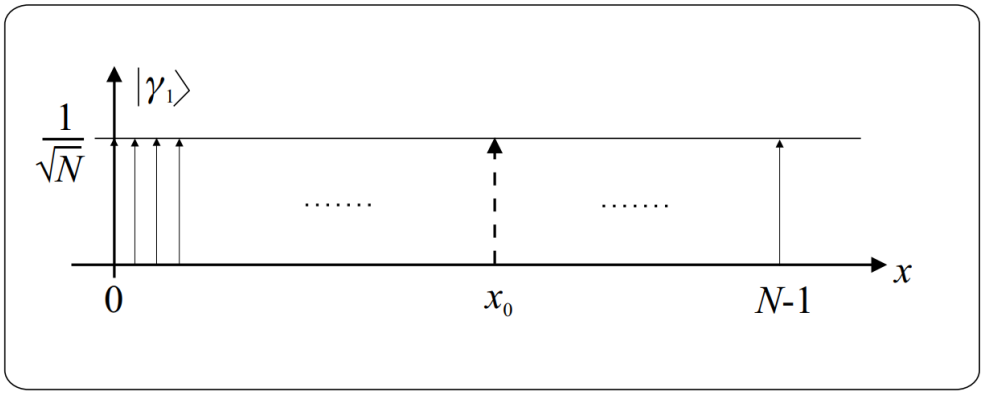
5.1.1. A Grover algoritmus működése- alapok

Az algoritmus működéséhez 2 kvantumregiszter szükséges. A „felső” (később a második ábrán) n bites. A regiszter n bitszámát úgy választjuk ki, hogy az adatbázis N elemszáma előálljon $N = 2^n$ alakban. Amennyiben N elemszáma nem kettő hatványa, kiegészítjük érdektelen értékekkel. Ezt a regisztert egy n bites Hadamard kapura vezetjük, míg az alsót egy ismeretlen T kapura. A kapuk után előálló állapot, azaz a valószínűségi amplitúdók eloszlása a második ábrán látható. Az x értékek az adatbázis elemei, x_0 a keresett érték. Gyakori jelölés, hogy az adatbázisban azokat az x elemet/elemeket, melyet/melyeket keresünk jelzett állapotnak/állapotoknak, míg a többi állapotot jelzetlen állapotnak nevezzük. Az amplitúdók értékei egységesen $\frac{1}{\sqrt{N}}$. Ez beleillik a korábban tanultakba, miszerint a valószínűségi amplitúdók négyzetösszegei az 1 értéket kell kiadják, ami ebben az esetben teljesül is. A **G,** azaz a Grover operátor

első állomása az úgynevezett Orákulum. Az Orákulum feladata az, hogy felismerje az adatbázisban a jelzett elemet, majd ennek valószínűségi amplitúdóját megszorozza -1-gyel, míg a jelzetlen állapotokét változatlanul hagyja.
A **G** következő lépése, a jelzett állpapot valószínűségi amplitúdójának növelése, míg a jelzetlenek amplitúdóinak elnyomása. Ennek matematikai eszköze az átlag körüli tükrözés. Ennek egy precízebb megfogalmazása az, hogy minden értéket kivonunk az átlag kétszereséből. Az átlagunk valamivel kisebb mint $\frac{1}{\sqrt{N}}$, hiszen van egy $-\frac{1}{\sqrt{N}}$ értékünk is. Ezt az átlagot jelöljük $\overline{a}$ -val. Látható, hogy ennek hatására a jelzetlen állapotok valószínűségi amplitúdóinak értéke az eredeti átlag alá csökken. Jelzett állapoté azonban $2\overline{a} + \frac{1}{\sqrt{N}}$ lesz. Ehhez a második lépéshez szükség van két Hadamard kapura és egy vezérelt fáziseltoló kapura (amit P-vel jelölünk), melyet 180º-os forgatásra állítunk be a megfelelő valószínűségi amplitúdóra. Ezek után a **G** operátort a **G** = HPHO alakban írhatjuk fel. Az algoritmust úgy használjuk, hogy a jelzett érték fázistolását, majd az átlag körüli tükrözést újra és újra elvégezzük, ezáltal jobban kiemelve a valószínűségi amplitúdóját, és jobban elnyomva a jelzetlen állapotokét. Fontos kérdés azonban, hogy hányszor éri meg iterálni.



10. ábra - a Grover algoritmus implementációja



11. ábra - A valószínűségi amplitúdók eloszlása a fázisforgatás előtt

5.1.2. A Grover algoritmus működése – az iterációk száma

Az iterációk számának eldöntéséhez egy geometriai interpretációt használunk. Ehhez egy olyan bázist írunk fel, melyben a kezdeti $|\gamma_1\rangle$ állapotot felírhatjuk $|\alpha\rangle$ és $|\beta\rangle$ segítségével. Itt $|\alpha\rangle$ tartalmazza jelzetlen x értékeket míg $|\beta\rangle$ a jelzetteket. $|\alpha\rangle$ és $|\beta\rangle$ egy ortogonálist bázist alkotnak, melyben $|\gamma_1\rangle$ $|\alpha\rangle$-val egy $\frac{\Omega\gamma}{2}$ fokos szöget zár be. Megmutatható, hogy minden iterációban $\Omega\gamma$ szöget forgatunk $|\beta\rangle$ felé a $|\gamma_1\rangle$ pozíciójából kezdve. $\Omega\gamma$-t N és M segítségével számíthatjuk, ahol N a korábbiakhoz hasonlóan az adatbázis elemszáma, M pedig a jelzett elemek száma az adatbázisban. Célunk az, hogy $|\gamma_1\rangle$-et beleforgassuk a $|\beta\rangle$ tengelybe, majd ezután elvégezzük a mérést. Ez azonban nem mindig tehető meg. Sok esetben meg kell elégednünk azzal, hogy $|\gamma_1\rangle$-et a lehető legközelebb forgatjuk $|\beta\rangle$-hez. Ez a gyakorlatban azt jelenti, hogy a mérés valamekkora valószínűséggel hibás lesz. Mivel a lehető legkevesebb iterációt szeretnénk végezni, ezért belátható, hogy az optimális iterációszám: $L_{opt_0} \cong \frac{\pi}{4}\sqrt{\frac{N}{M}}$. Mint azt a bevezető részben is említettük a Grover algoritmus nagy előnye az, hogy N elemszámú adatbázis esetén a jelzett elem megtalálásához szükséges lépések száma N négyzetgyökével skálázódik.

5.1.3. A Grover algoritmus használata az RSA kód feltörésér

Ahogy abban az RSA-ról szóló fejezetben is szó volt, a kódolás alapját azt képezi, hogy két nagy prímszám szorzatát kiszámolni egyszerű, azonban egy nagy számot visszafejteni prímtényezőkre „nagyon nehéz”. A kérdés az, hogy tudunk-e ezen a nehézségen könnyíteni a Grover algoritmus segítségével. A válasz az, hogy igen. A feladatunk nem más, mint megkeresni N osztóját egy természetes számokat tartalmazó adatbázisban. Az adatbázis mérete a 2 és $\sqrt{N}$ közötti számok száma, ugyanis elég eddig keresni, hogy megtaláljuk N osztói közül a kisebbet, ahonnan a nagyobb már triviálisan adódik. Az is egyértelműen látszik, hogy M = 1. Ezekből az iterációk számára a következő kifejezés adódik: $L_{opt_0} \cong \frac{\pi}{4}\sqrt{\sqrt{N}}$, ami azt jelenti, hogy Grover algoritmust használva a számítási idő N negyedik gyökével skálázódik. Létezik a Grover algoritmusnak más felhasználási lehetősége is az RSA feltörésére, azonban annak hatékonysága nem éri az imént vázoltét, ezért csak említést teszünk róla.

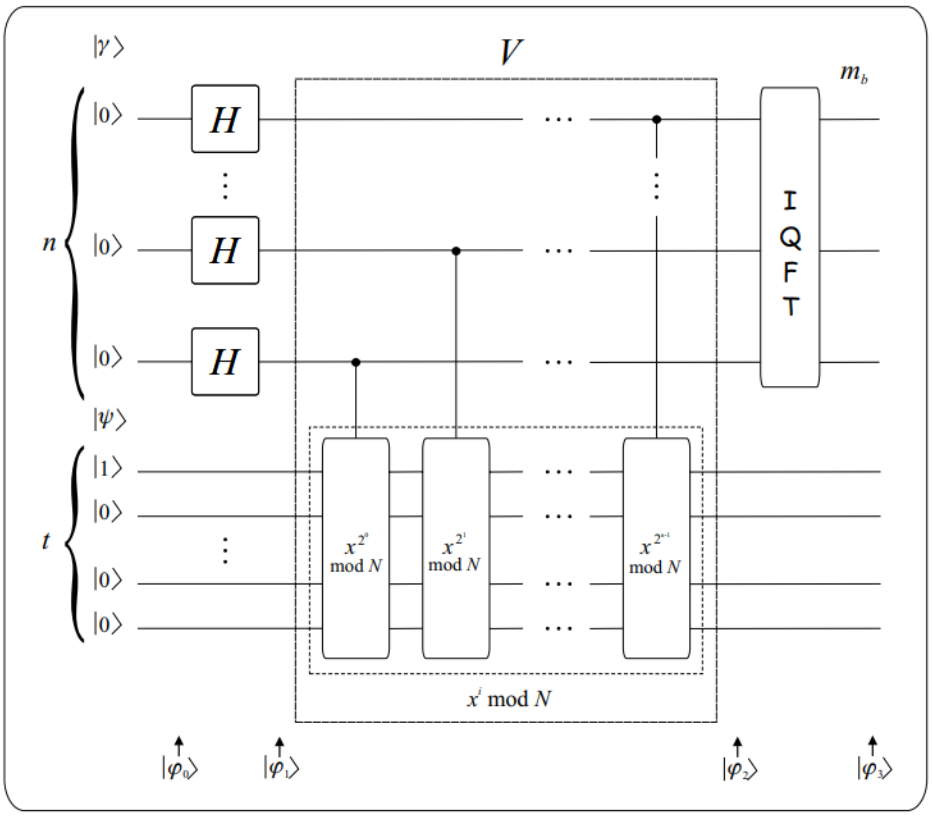
5.2. A Shor algoritmus

Ahogy azt a korábbiakban tárgyaltuk, jelenleg nem ismerünk olyan klasszikus algoritmust, mely a számok prímtényezőkre bontását polinomiális időben el tudná végezni. Peter Shor amerikai matematikus 1994-ben feltalált egy olyan kvantumalgoritmust, mely rendelkezik ezzel a kívánatos tulajdonsággal. Ez az algoritmus az ő nevét viselő Shor algoritmus, mely egy klasszikus és egy kvantumos részből áll.

5.2.1. A Shor algoritmus – működés

A Shor algoritmus működése egy klasszikus lépéssel kezdődik, mégpedig azzal, hogy véletlenszerűen kiválasztunk egy **a** számot, mely kisebb, mint a faktorizálandó N számunk. Ebben a lépésben azt vizsgáljuk meg, hogy **a** és N relatív prímek-e. Amennyiben ezek nem relatív prímek, úgy **a** N egyik faktora, és az algoritmus véget ér. Ha ez nem így van, azaz **a** és N legnagyobb közös osztója 1 (azaz relatív prímek), akkor az előkészítési fázis véget ért, áttérhetünk az algoritmus kvantumos részére. A Shor algoritmus lényegi részének alapját a rend keresése adja. Ebben a kontextusban a rend

egy számelméleti fogalom, melyet úgy definiálhatunk, hogy az a legkisebb pozitív természetes **r** szám, melyre teljesül az $x^{\mathbf{r}} \bmod N = 1$ egyenlet (x és N pozitív egész és x < N). Ekkor azt mondjuk, hogy x rendje **r** modulo N értelemben. A Shor algoritmus ennek a rendnek a megtalálásához a Kvantum Fourier transzformációt hívja segítségül.



12. ábra- A Shor algoritmus implementációja

5.2.2. A Shor algoritmus – sebesség

A Shor algoritmus előnyét az jelenti más (klasszikus vagy kvantumos) algoritmusokéhoz képest, hogy erre van legkevésbé hatással a modulus hosszának növelése. Az algoritmus N kettes alapú logaritmusának harmadik hatványával skálázódik, mellyel nagy számok esetén jóval alulmarad vetélytársaitól.

5.3. A Shor algoritmus használata az RSA kód feltörésére

A Shor algoritmus még a Grover algoritmusénál is hatékonyabb az RSA kód feltörésének tekintetében. A feltörés a következőképpen néz ki:
Első lépésben megkeressük az elfogott kódolt üzenet (C) r rendjét modulo N értelemben. Ne felejtsük el, hogy N-nek és e-nek a birtokában vagyunk, hiszen ezek alkotják a publikus kulcsot. Ehhez a lépéshez használjuk a Shor algoritmust. Ezután kiszámítjuk e modulo r multiplikativ inverzét, jelöljük d' -vel, melyről tudjuk, hogy d'=d + k*r, ahol k egész, melynek valamilyen értékére φ(N) = k * r. Ennek a d' értéknek a birtokában már visszafejthetjük az üzenetet, csupán annyi a dolgunk, hogy a visszafejtés egyenletében az ismeretlen d értéket ezzel a d'-vel helyettesítjük.

6. Szimuláció

6.1. Grover-algoritmus szimulációja

6.1.1. Egyszerű Grover algoritmus szimulálása

Először tekintsünk egy klasszikus keresést egy adathalmazban:

```
]: import random
   def classicOracle(input):
       value = 1
       if input == value:
           return True
       else: return False

   l = []
   for i in range(1000):

       randlist = random.sample(range(0,10), 10)

       for index, value in enumerate(randlist):
           if classicOracle(value) == True:
               l.append(index+1)
               break
   print("Átlagos futási szám:", np.mean(l))
```

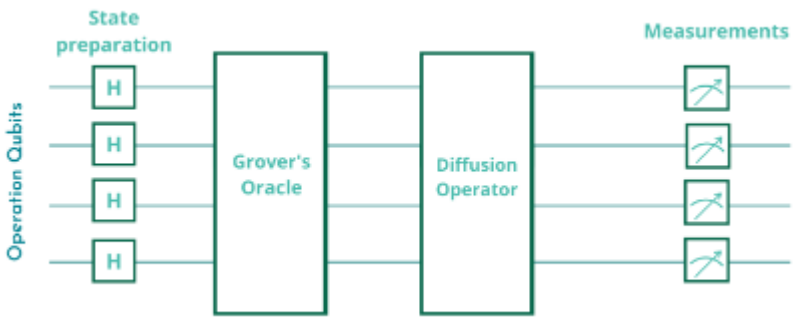
```
Átlagos futási szám: 5.471
```

13. ábra klasszikus keresés

Itt véletlenszerűen generálunk 10 számot majd ezek között keressük az 1-es értéket, amit átlagban 5,5-szer keres

Mostmár továbbléphetünk a kvantumos megközelítésre, ahol a Grover-algoritmust fogjuk használni.

A következő hálózatot kell implementálni:



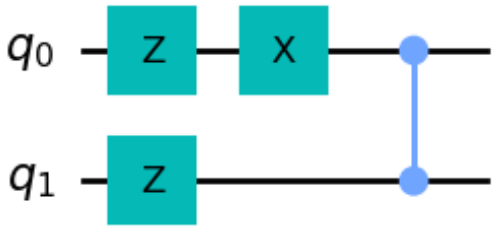
14. ábra Áramkör felépítése

6.1.1.1. Példa: 2qbites Grover

Első lépésként inicializáljuk az áramkört, majd hozzuk létre az Oracle-t:

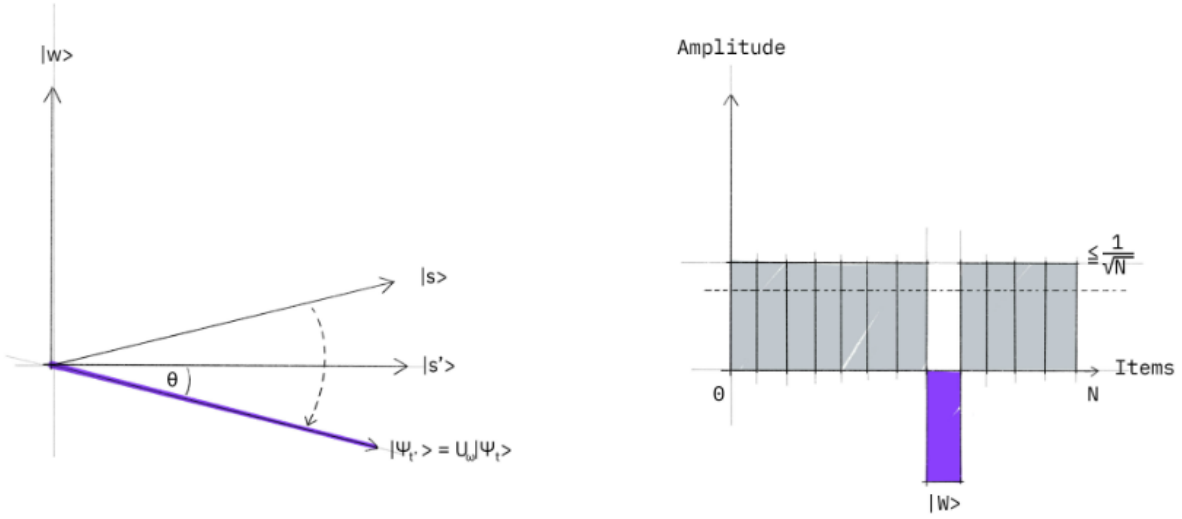
```
[18]: #Oracle for |00>
      oracle = QuantumCircuit(2,name='oracle')
      oracle.z([0,1])
      oracle.x([0])
      oracle.cz(0,1)
      oracle.to_gate()
      oracle.draw(output='mpl')
```

[18]:



15. ábra Oracle 2kvantumbites keresőhöz

Az Oracle az alábbi transzformációért lesz felelős:

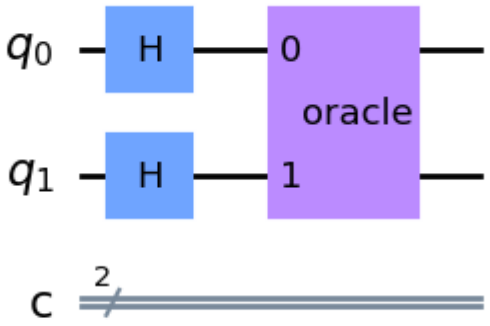


16. ábra oracle transzformáció

Hozzuk létre az áramkörünket, amiben felhasználjuk az Oracle-t.

```
[19]: grover_circ = QuantumCircuit(2,2)
      grover_circ.h([0,1])
      grover_circ.append(oracle,[0,1])
      grover_circ.draw(output='mpl')
```

[19]:



17. ábra H kapu+ Oracle

Itt érdemes megvizsgálni az oracle kimenetén lévő állapotvektorokat, ugyanis így tudjuk ellenőrizni, hogy az oracle a megfelő elem amplitúdóját invertálta-e meg.

Check the oracle output

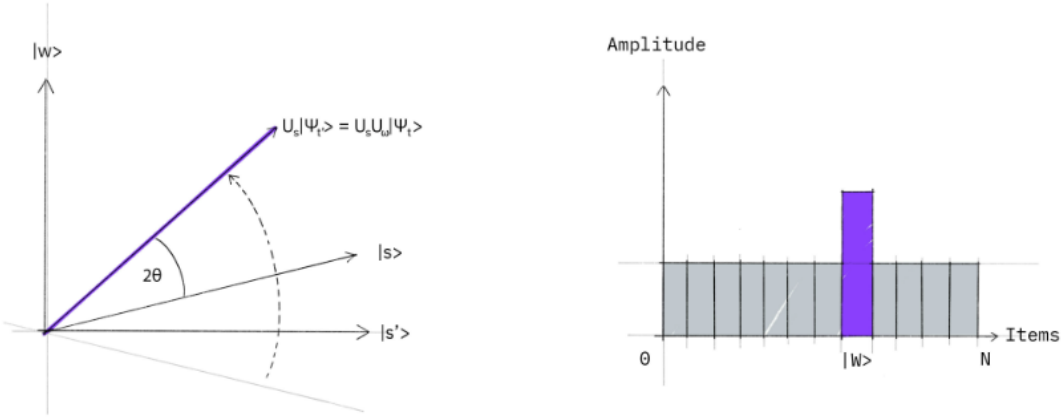
```
[20]: backend = Aer.get_backend('statevector_simulator')
      job = execute(grover_circ,backend)
      result=job.result()
      outputstate = result.get_statevector()
      print(outputstate)
```

```
Statevector([-0.5+6.123234e-17j,  0.5+0.000000e+00j,  0.5+0.000000e+00j,
             0.5+6.123234e-17j],
            dims=(2, 2))
```

18. ábra állapotvektor |00>-hoz

A kimenetről azt lehet leolvasni, hogy a |00> állapothoz tartozó együttható – ½ , míg a többi elemnél + ½ , tehát az oracle jól működik.

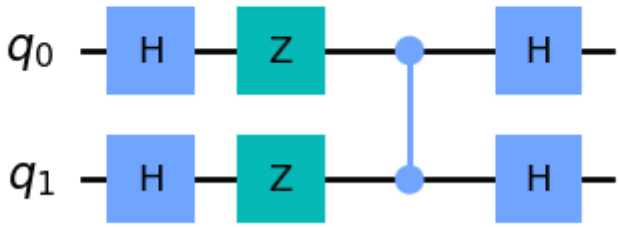
Utolsó előtti lépésként egy további reflexiót valamint amplitúdóerősítést alkalmazunk az állapoton.



19. ábra Grover operátor hatása

A reflexióért és erősítésért felelő áramkör az alábbi módon épül fel. (2 qbit esetében)

```
diffuser = QuantumCircuit(2,name = 'diffuser')
diffuser.h([0,1])
diffuser.z([0,1])
diffuser.cz(0,1)
diffuser.h([0,1])
diffuser.to_gate()
diffuser.draw(output='mpl')
```

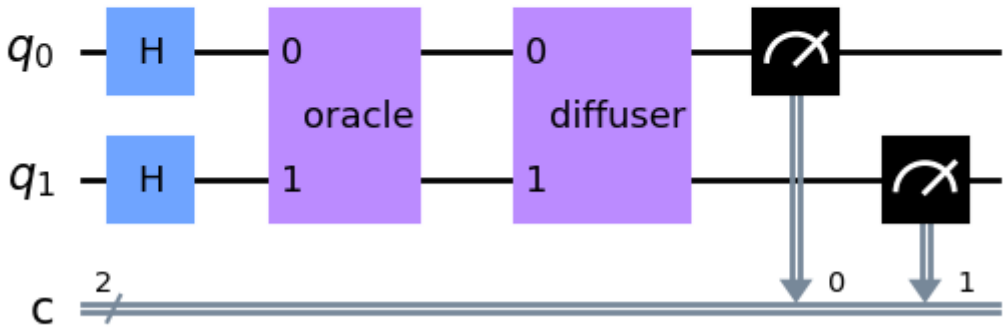


20. ábra 2qbit Diffuser

Már csak annyi maradt, hogy megmérjük a végeredményt.

A teljes áramkör pedig az alábbi elemekből épül fel:

```
grover_circ=QuantumCircuit(2,2)
grover_circ.h([0,1])
grover_circ.append(oracle,[0,1])
grover_circ.append(diffuser,[0,1])
grover_circ.measure([0,1],[0,1])
grover_circ.draw(output='mpl')
```



21. ábra Teljes áramkör az 1. példához

Következhet a szimuláció:
Először az egész áramkörre vonatkozó állapotvekort vizsgáltam meg, ami már az amplitúerősítés utáni állapotokat tartalmazza:

```
job = execute(grover_circ,backend)
result=job.result()
outputstate = result.get_statevector()
print(outputstate)
```

```
Statevector([ 1.+1.2246468e-16j, -0.+0.0000000e+00j, -0.+0.0000000e+00j,
              0.+0.0000000e+00j],
            dims=(2, 2))
```

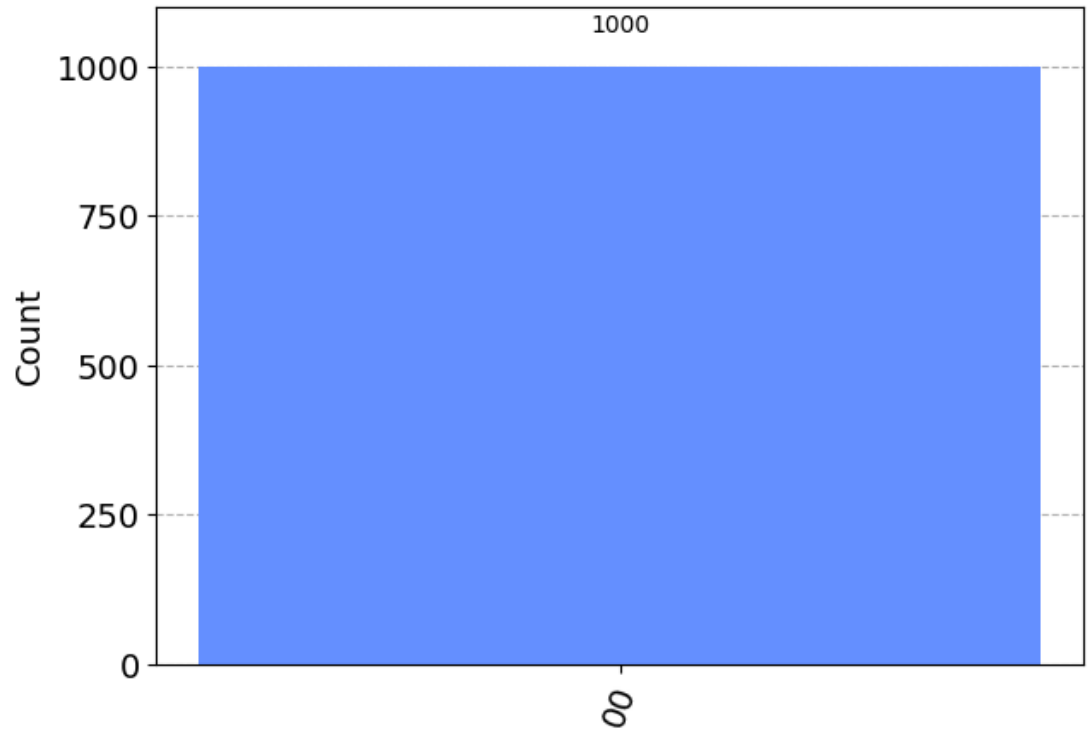
Ezen azt lehet látni, hogy a |00> kvantumállapot valószínűsége 1 lesz.

Szimuláció:

```
backend = Aer.get_backend('qasm_simulator')
transpiled_grover_circuit = transpile(grover_circ, backend, optimization_level=3)
job = backend.run(transpiled_grover_circuit)
job_monitor(job, interval=2)

job = execute(grover_circ, backend, shots = 1000)
result = job.result()
result.get_counts()

results = job.result()
answer = results.get_counts(grover_circ)
plot_histogram(answer)
```

```
Job Status: job has successfully run
```

:



22. ábra Szimuláció eredménye

Ebben az esetben az 00 kimeteket fogjuk megtalálni 100%-os pontossággal ami szorosan összefügg az állapotvektorban látott 1-szeres amplitúdóval.

Szimuláció esetében a körülmények tökéletesek, nincs zaj a rendszeren belül ezért következhet a valós kvantumszámítógéppel végzett tesztelés.

Valós kvantumszámítógép használata

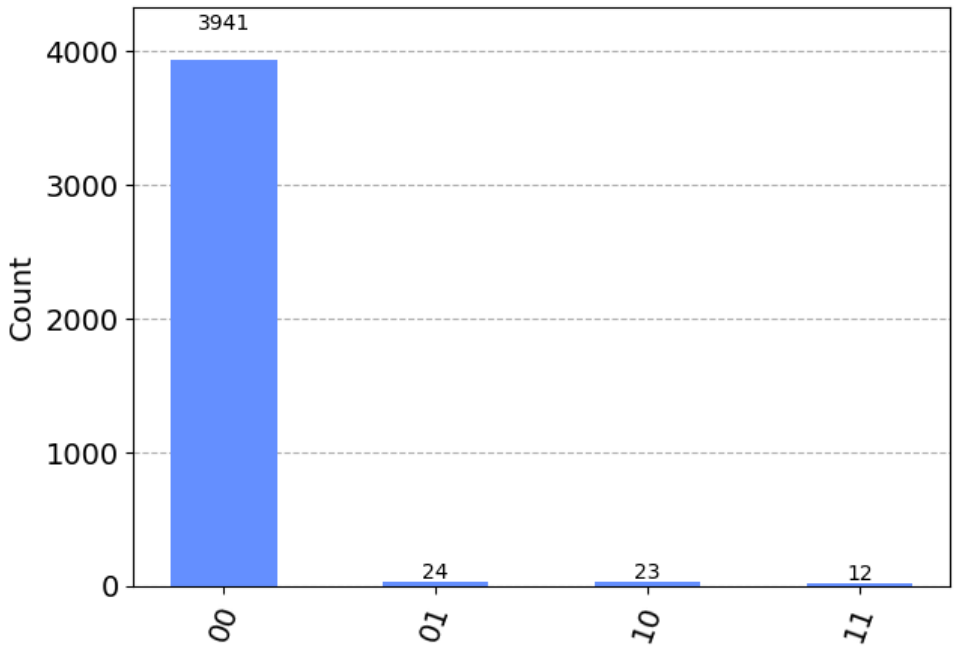
```
# Load IBM Q account and get the least busy backend device
provider = IBMQ.load_account()
provider = IBMQ.get_provider("ibm-q")
device = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= 3 and
                                    not x.configuration().simulator and x.status().operational==True))
print("Running on current least busy device: ", device)
```

```
ibmqfactory.load_account:WARNING:2023-12-05 18:13:07,363: Credentials are already in use. The existing account in the s
ced.

Running on current least busy device:  ibm_osaka
```

```
transpiled_grover_circuit = transpile(grover_circ, device, optimization_level=3)
job = device.run(transpiled_grover_circuit)
job_monitor(job, interval=2)
```

```
Job Status: job has successfully run
```

```
# Get the results from the computation
results = job.result()
answer = results.get_counts(grover_circ)
plot_histogram(answer)
```



23. ábra Eredmények valós kvantumszámítógéppel 1.

Ahogy az ábrán is látszik itt már nem sikerül elérni a 100%-os pontosságot, mint a szimulátor esetében, de a hiba mértéke elhanyagolható. Egészen pontosan 59/4000 = 1,475%.

6.1.1.2. Példa: 4qbites Grover

Az alapelv ugyanaz mint az előző példánál, az egyetlen különbség, hogy 2-nél több bit esetében már számít az iterációk száma, ami azt határozza meg, hogy hányszor kell alkalmazni a grover operátort egymás után.

Egy általános diffuser bármekkora nqubits méretű áramkörhöz:

Álatános diffuser

```
def diffuser(nqubits,qc,aux):
    # Apply transformation |s> -> |00..0> (H-gates)
    for qubit in range(nqubits):
        qc.h(qubit)
    # Apply transformation |00..0> -> |11..1> (X-gates)
    for qubit in range(nqubits):
        qc.x(qubit)
    # Do multi-controlled-Z gate
    qc.h(nqubits-1)
    qc.mct(list(range(nqubits-1)), nqubits-1,aux)  # multi-controlled-toffoli
    qc.h(nqubits-1)
    # Apply transformation |11..1> -> |00..0>
    for qubit in range(nqubits):
        qc.x(qubit)
    # Apply transformation |00..0> -> |s>
    for qubit in range(nqubits):
        qc.h(qubit)
```

24. ábra Általános diffuser

Az alábbi példában 4 kvantumbittel dolgozok:

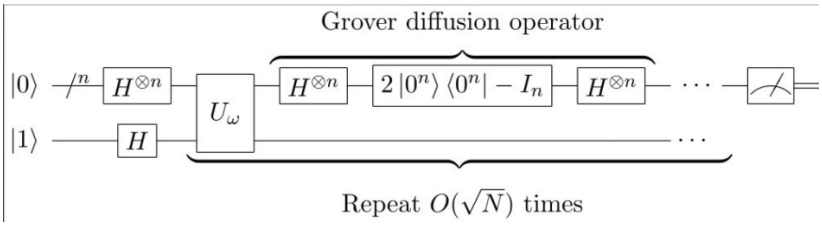
Elsőként létrehozom az áramkört, majd inicializálom azt egy Hadamard kapu segítségével.

Eddig semmi különös nincs.

```
# Number of qubits
num_qbits = 4
# Number of items we're looking for
M = 1

# Qantumregisters
qr = QuantumRegister(num_qbits)
cr = ClassicalRegister(num_qbits)
# Aux qubits if num_qubits > 3
if num_qbits > 3:
    aux = QuantumRegister(num_qbits-3)
    groverCircuit = QuantumCircuit(qr,cr, aux)
else:
    aux = None
    groverCircuit = QuantumCircuit(qr,cr)
```

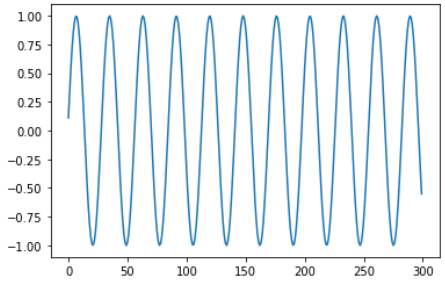
```
# Hadamard init
groverCircuit.h(qr)
```

A Grover operátor elkészítése:



25. ábra Ábra több mint 2 qbithez

Ahhoz, hogy az algoritmus a lehető legpontosabb legyen a Grover operátort $I_{opt} = \frac{\pi}{4} \cdot \sqrt{\frac{N}{M}}$ -szor egymás után fel kell venni, mert a nem optimális számú iteráció ront apontosságon:



26. ábra pontosság az iteráció függvényében

A lényegi eltérés itt tapasztalható.

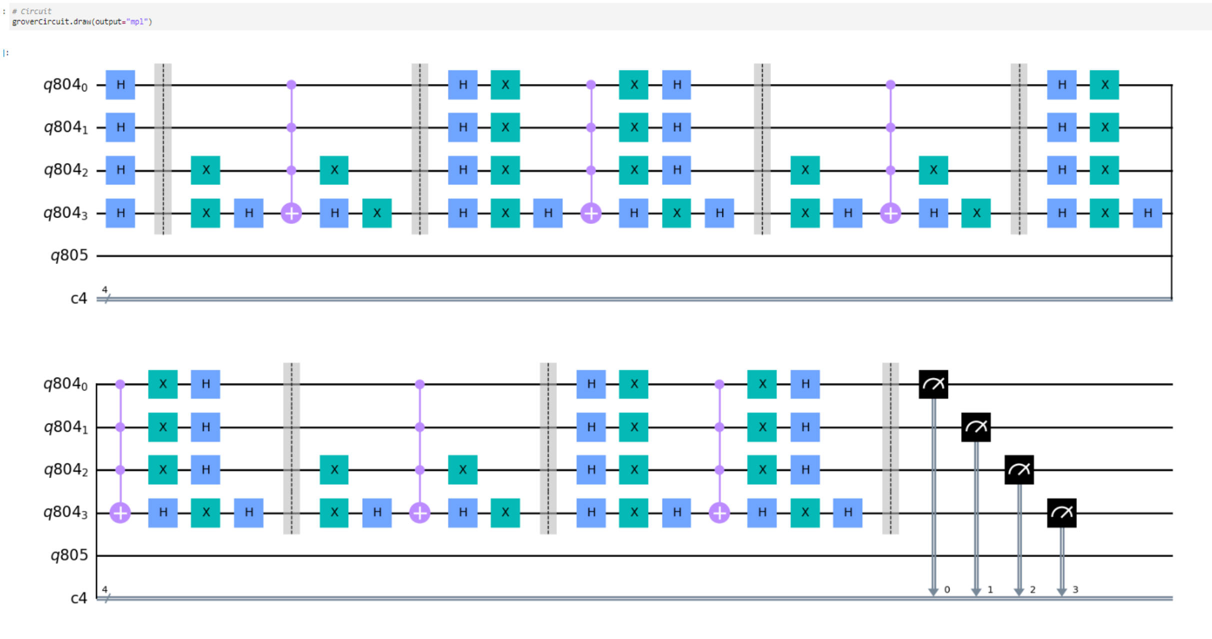
A Grover operátor:

```
result = math.pi/4*math.sqrt((2**num_qbits)/M) # Iterációk kiszámítása
iterations = round(result)
print("Num of iterations %d"%iterations)

# Grover
for i in range(iterations): # pi/4 * ((2^n)^(1/M))
    groverCircuit.barrier(qr)
    # Oracle for  0011
    groverCircuit.x([2,3])
    groverCircuit.h(qr[num_qbits-1])
    groverCircuit.mct(qr[0:num_qbits-1], qr[num_qbits-1], aux)
    groverCircuit.h(qr[num_qbits-1])
    groverCircuit.x([2,3])
    groverCircuit.barrier(qr)
    # Diffuser
    diffuser(num_qbits,groverCircuit,aux),range(num_qbits)
groverCircuit.barrier(qr)
groverCircuit.measure(qr,cr)
```

```
Num of iterations 3
```

27. ábra Grover operátor 4qbit esetén

Ebben az esetben az iterációk száma 3 így 3-szor fog bekerülni egymás mögé a Grover operátor, ahogy a következő ábra mutatja.



28. ábra Teljes hálózat 4qbit esetén

Az Oracle jelen esetben úgy van beállítva, hogy egy állapotot fog kiválasztani, így ha többet szeretnénk keresni, akkor több ilyen operátort kell egymás mögé elhelyezni.

Az egész rendszerre vonatkozóan az állapotvektor így néz ki:

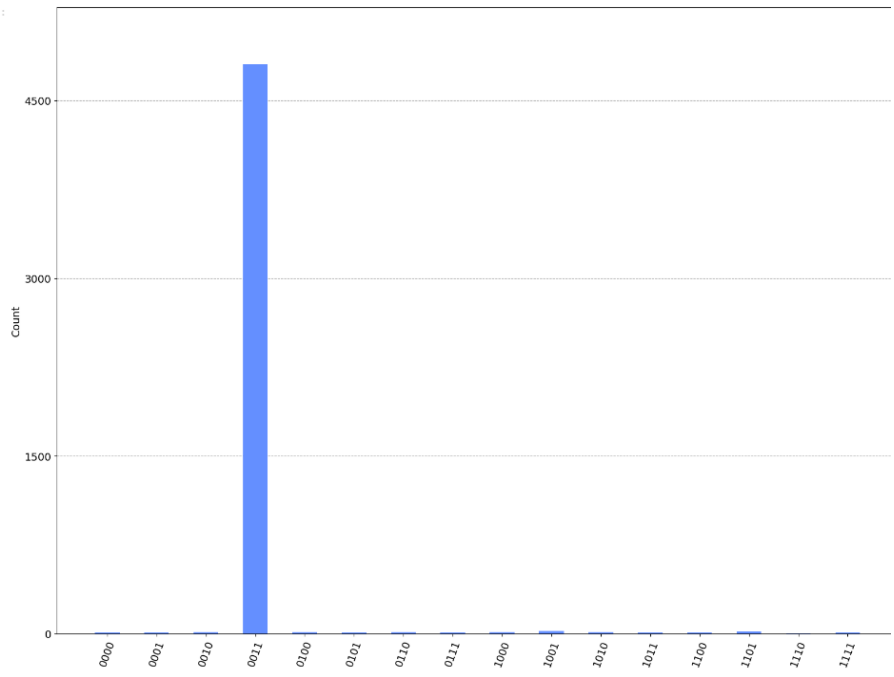
```
# Statevector
backend_sim = Aer.get_backend('statevector_simulator')
job_sim = execute(groverCircuit, backend_sim)
statevec = job_sim.result().get_statevector()
print(statevec)
```

```
Statevector([ 0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
             -1.-1.77256646e-15j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j,  0.+0.00000000e+00j,
              0.+0.00000000e+00j,  0.+0.00000000e+00j],
            dims=(2, 2, 2, 2, 2))
```

ahol a 4. elemnél látunk egy -1-szeres szorzót a többi pedig 0. Ez az elem a 0011-nek felel meg
Megjegyzés: Minket csak az első 16 elem érdekel, a többi a segédregiszter(aux) miatt került be.

Szimuláció:

```
# Simulation
backend = Aer.get_backend('qasm_simulator')
shots = 5000
results = execute(groverCircuit, backend=backend, shots=shots).result()
answer = results.get_counts()
plot_histogram(answer, figsize=[20,15], bar_labels=False)
```

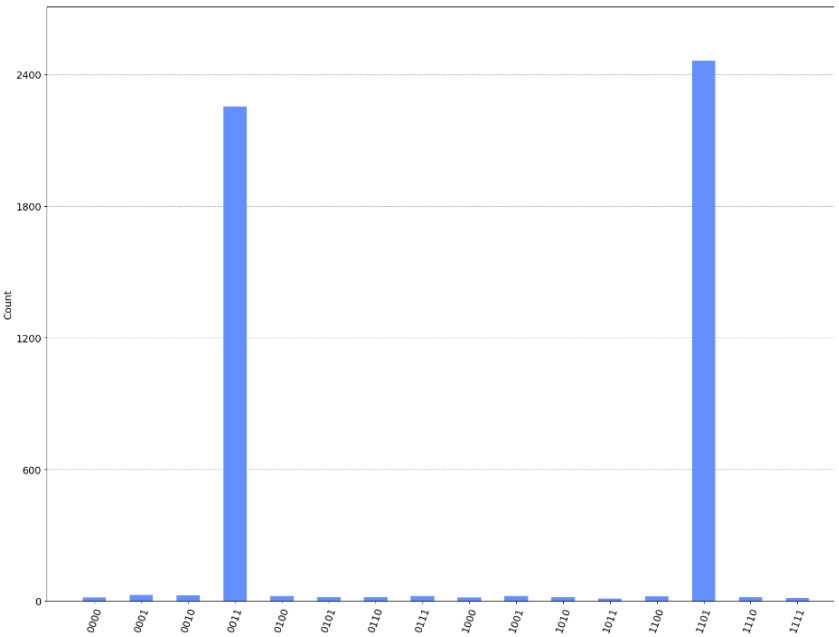


29. ábra Szimuláció |0011> esetén

A képen azt láthatjuk, hogy ismét elég jól sikerül elkülöníteni az általunk kiválasztott értéket a többitől.

|0011> és |1101> állapotokhoz tartozó oracle-t használtam:

Ebben az esetben az M értékét már 2-re kell megemelni, mert 2 különböző elemet keresek



30. ábra Szimuláció |0011> és|1101> esetén

Az első beállításhoz tartozó szimuláció valós kvantumszámítógéppel:

Valós kvantumszámítóval

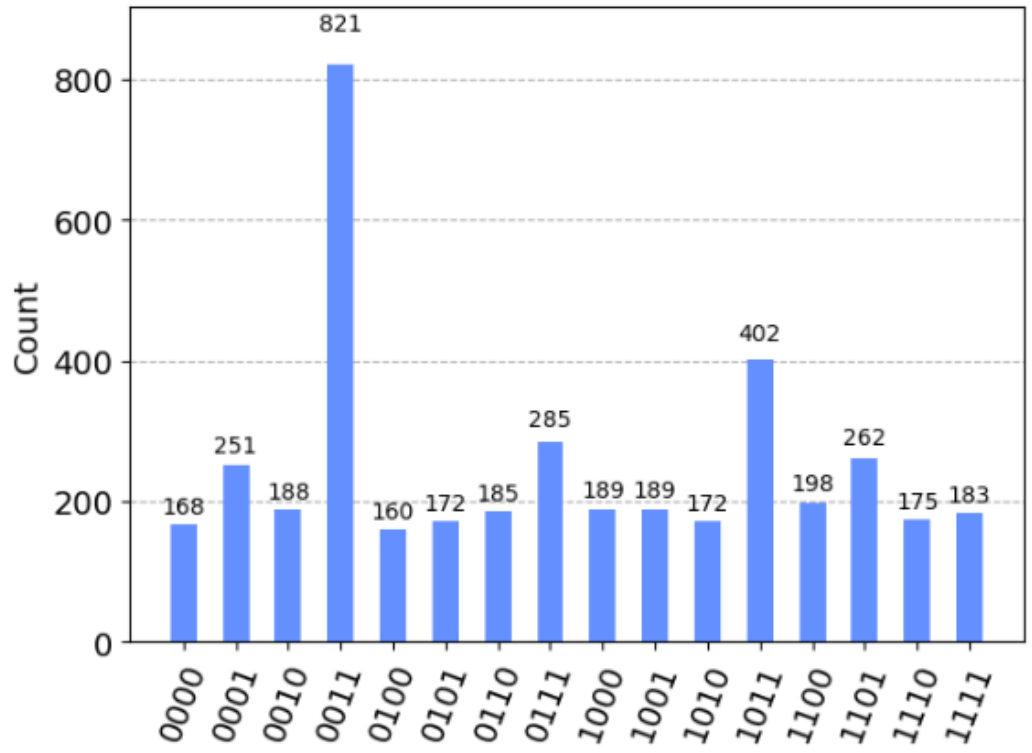
```
provider = IBMQ.load_account()
provider = IBMQ.get_provider("ibm-q")
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= 3 and
                                       not x.configuration().simulator and x.status().operational==True))
print("least busy backend: ", backend)
```

```
ibmqfactory.load_account:WARNING:2023-12-05 18:35:38,903: Credentials are already in use. The existing account

least busy backend:  ibm_osaka
```

```
transpiled_grover_circuit = transpile(groverCircuit, backend, optimization_level=3)
job = backend.run(transpiled_grover_circuit)
job_monitor(job, interval=2)
```

```
Job Status: job has successfully run
```

```
# Get the results from the computation
results = job.result()
answer = results.get_counts(groverCircuit)
plot_histogram(answer)
```

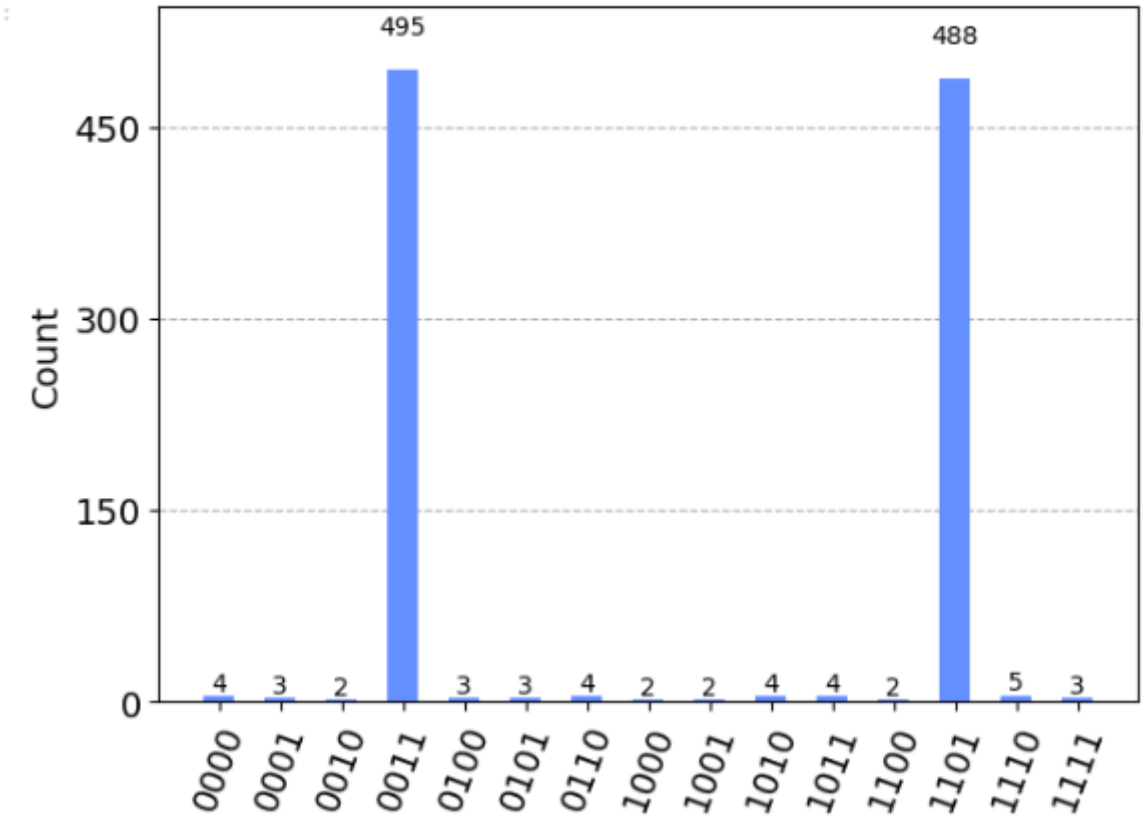


31. ábra Eredmény valós kvantumszámítógéppel 4qbit esetén 1.

Itt csak a |0011> hez tartozó Oracle volt „bekapcsolva”. A végeredményen látszik, hogy a keresett elem jól kivehető, viszont a hiba mértéke nagyon nagy, még úgy is, hogy ez a szimuláció a jobbik esetek közé tartozott és igen nagy volt az eltérés az 1. és 2. helyezett között. Átlagban ez

Második eset: 2 Oracle jelenlétével:

```
# Get the results from the computation
results = job.result()
answer = results.get_counts(groverCircuit)
plot_histogram(answer)
```



32. ábra Eredmény valós kvantumszámítógéppel 4qbit esetén 2.

Ebben az esetben jóval jobb eredményeket sikerült produkálni.

6.1.2. RSA feltörése Grover-algoritmussal

Az RSA feltöréséhez szükségünk van a publikus kulcs faktorizációjára

Ezt az alábbi Q# és python kóddal könnyedén meg tudjuk határozni nem túl nagy számokra.

Grover.qs:

```
namespace GroversTutorial {
    open Microsoft.Quantum.Canon;
    open Microsoft.Quantum.Intrinsic;
    open Microsoft.Quantum.Measurement;
    open Microsoft.Quantum.Math;
    open Microsoft.Quantum.Convert;
    open Microsoft.Quantum.Arithmetic;
    open Microsoft.Quantum.Arrays;
    open Microsoft.Quantum.Preparation;

    @EntryPoint()
    operation FactorizeWithGrovers(number : Int) : Unit {

        // Define the oracle that for the factoring problem.
        let markingOracle = MarkDivisor(number, _, _);
        let phaseOracle = ApplyMarkingOracleAsPhaseOracle(markingOracle, _);
        // Bit-size of the number to factorize.
        let size = BitSizeI(number);
        // Estimate of the number of solutions.
        let nSolutions = 4;
        // The number of iterations can be computed using the formula.
        let nIterations = Round(PI() / 4.0 * Sqrt(IntAsDouble(size) / IntAsDouble(nSolutions)));

        // Initialize the register to run the algorithm
        use (register, output) = (Qubit[size], Qubit());
        mutable isCorrect = false;
        mutable answer = 0;
        // Use a Repeat-Until-Succeed loop to iterate until the solution is valid.
        repeat {
            RunGroversSearch(register, phaseOracle, nIterations);
            let res = MultiM(register);
            set answer = BoolArrayAsInt(ResultArrayAsBoolArray(res));
            // See if the result is a solution with the oracle.
            markingOracle(register, output);
            if MResetZ(output) == One and answer != 1 and answer != number {
                set isCorrect = true;
            }
            ResetAll(register);
        } until isCorrect;

        // Print out the answer.
        Message($"The number {answer} is a factor of {number}.");

    }
```

33. ábra Grover algoritmus RSA feltöréséhez

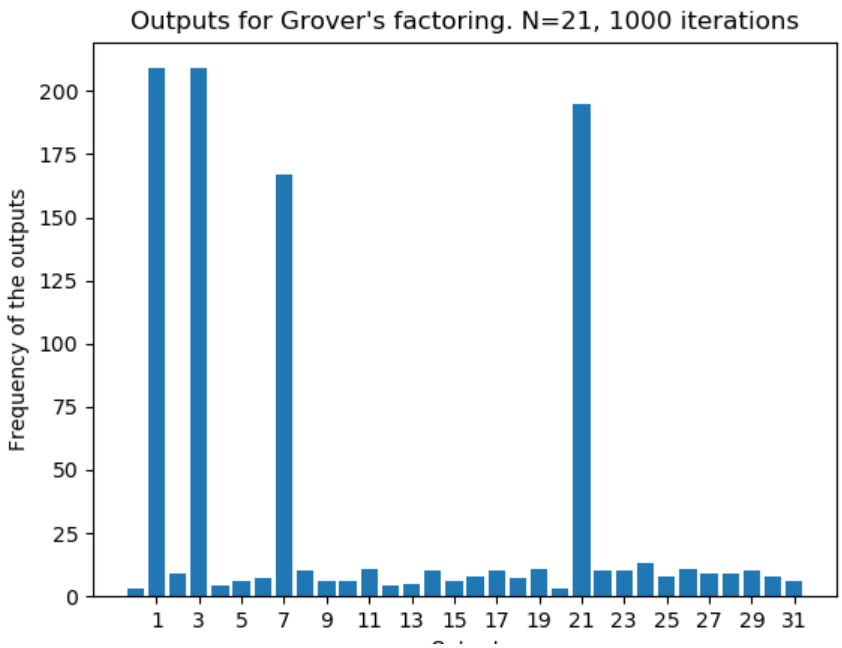
```
operation MarkDivisor (
    dividend : Int,
    divisorRegister : Qubit[],
    target : Qubit
) : Unit is Adj + Ctl {
    // Calculate the bit-size of the dividend.
    let size = BitSizeI(dividend);
    // Allocate two new qubit registers for the dividend and the result.
    use dividendQubits = Qubit[size];
    use resultQubits = Qubit[size];
    // Create new LittleEndian instances from the registers to use DivideI
    let xs = LittleEndian(dividendQubits);
    let ys = LittleEndian(divisorRegister);
    let result = LittleEndian(resultQubits);

    // Start a within-apply statement to perform the operation.
    within {
        // Encode the dividend in the register.
        ApplyXorInPlace(dividend, xs);
        // Apply the division operation.
        DivideI(xs, ys, result);
        // Flip all the qubits from the remainder.
        ApplyToEachA(X, xs!);
    } apply {
        // Apply a controlled NOT over the flipped remainder.
        Controlled X(xs!, target);
        // The target flips if and only if the remainder is 0.
    }
}
```

```
operation PrepareUniformSuperpositionOverDigits(digitReg : Qubit[]) : Unit is Adj + Ctl {
    PrepareArbitraryStateCP(ConstantArray(10, ComplexPolar(1.0, 0.0)), LittleEndian(digitReg));
}

operation ApplyMarkingOracleAsPhaseOracle(
    markingOracle : (Qubit[], Qubit) => Unit is Adj,
    register : Qubit[]
) : Unit is Adj {
    use target = Qubit();
    within {
        X(target);
        H(target);
    } apply {
        markingOracle(register, target);
    }
}

operation RunGroversSearch(register : Qubit[], phaseOracle : ((Qubit[]) => Unit is Adj), iterations : Int) : Unit {
    ApplyToEach(H, register);
    for _ in 1 .. iterations {
        phaseOracle(register);
        ReflectAboutUniform(register);
    }
}

operation ReflectAboutUniform(inputQubits : Qubit[]) : Unit {
    within {
        ApplyToEachA(H, inputQubits);
        ApplyToEachA(X, inputQubits);
    } apply {
        Controlled Z(Most(inputQubits), Tail(inputQubits));
    }
}
```

Grover.py:

```
import qsharp
qsharp.packages.add("Microsoft.Quantum.Numerics")
qsharp.reload()
from GroversTutorial import FactorizeWithGrovers2
import matplotlib.pyplot as plt
import numpy as np

def main():

    # Instantiate variables
    frequency =  {}
    N_Experiments = 1000
    results = []
    number = 21

    # Run N_Experiments times the Q# operation.
    for i in range(N_Experiments):
        print(f'Experiment: {i} of {N_Experiments}')
        results.append(FactorizeWithGrovers2.simulate(number = number))

    # Store the results in a dictionary
    for i in results:
        if i in frequency:
            frequency[i]=frequency[i]+1
        else:
            frequency[i]=1

    # Sort and print the results
    frequency = dict(reversed(sorted(frequency.items(), key=lambda item: item[1])))
    print('Output,  Frequency' )
    for k, v in frequency.items():
        print(f'{k:<8} {v}')

    # Plot an histogram with the results
    plt.bar(frequency.keys(), frequency.values())
    plt.xlabel("Output")
    plt.ylabel("Frequency of the outputs")
    plt.title("Outputs for Grover's factoring. N=21, 1000 iterations")
    plt.xticks(np.arange(1, 33, 2.0))
    plt.show()

if __name__ == "__main__":
    main()
```

Itt a number helyére a nyilvános kulcsot beírva visszakapjuk a szám osztóit, amiből kiszűrjük a nem triviális osztókat. Mivel a nyilvános kulcs egy szemi-prím szám(két prímszám szorzata), ezért így már csak 2 osztó marad, azaz az a 2 prím amit kerestünk.



34. ábra N=21 faktoralizálása

Ezek után az alábbi kódrészlet megfelelő helyeire behelyettesítve visszakapjuk a tikosított szöveget.

```
bit_length = int(input("Enter bit_length: "))

public, private = generate_keypair(2**bit_length)
print(public[0],public[1])
```

RSA átalakítás

```
msg = input("\nWrite message: ")
encrypted_msg, encryption_obj = encrypt(msg, public)

print("\nEncrypted message: " + encrypted_msg)
print("\nEncrypted object: " , encryption_obj)
```

RSA visszaalakítás

```
decrypted_msg = decrypt(encryption_obj, private)

print("\nDecrypted message using RSA Algorithm: " + decrypted_msg)
```

Shor algo

```
N_shor = public[1]
assert N_shor>0,"Input must be positive"
p,q = "Ide jönne a Shor algoritmus ami bármilyen N-re megmondja a faktorokat"
print(p,q)
phi = (p-1) * (q-1)
d_shor = mod_inverse(public[0], phi)
```

Cracked RSA with Shor's algo

```
decrypted_msg = decrypt(encryption_obj, (d_shor,N_shor))

print('\nMessage Cracked using Shors Algorithm: ' + decrypted_msg + "\n")
```

35. ábra RSA feltörése Groverrel

Ahol a kulcsgeneráláshoz szükséges kód itt található:

```
def mod_inverse(a, m):
    for x in range(1, m):
        if (a * x) % m == 1:
            return x
    return -1

def generate_keypair(keysize):
    p = rand(1, 1000)
    q = rand(1, 1000)
    nMin = 1 << (keysize - 1)
    nMax = (1 << keysize) - 1
    primes = [2]
    start = 1 << (keysize // 2 - 1)
    stop = 1 << (keysize // 2 + 1)
    if start >= stop:
        return []
    for i in range(3, stop + 1, 2):
        for p in primes:
            if i % p == 0:
                break
        else:
            primes.append(i)
    while (primes and primes[0] < start):
        del primes[0]
    while primes:
        p = random.choice(primes)
        primes.remove(p)
        q_values = [q for q in primes if nMin <= p * q <= nMax]
        if q_values:
            q = random.choice(q_values)
            break
    print(p, q)
    n = p * q
    phi = (p - 1) * (q - 1)
    e = random.randrange(1, phi)
    g = gcd(e, phi)
    while True:
        e = random.randrange(1, phi)
        g = gcd(e, phi)
        d = mod_inverse(e, phi)
        if g == 1 and e != d:
            break

    return ((e, n), (d, n))

def encrypt(msg_plaintext, package):
    e, n = package
    msg_ciphertext = [pow(ord(c), e, n) for c in msg_plaintext]
    return ''.join(map(lambda x: str(x), msg_ciphertext)), msg_ciphertext

def decrypt(msg_ciphertext, package):
    d, n = package
    msg_plaintext = [chr(pow(c, d, n)) for c in msg_ciphertext]
    return (''.join(msg_plaintext))
```

36. ábra Kiegészítő kód RSA-hoz

6.2. Shor-algoritmus szimulációja

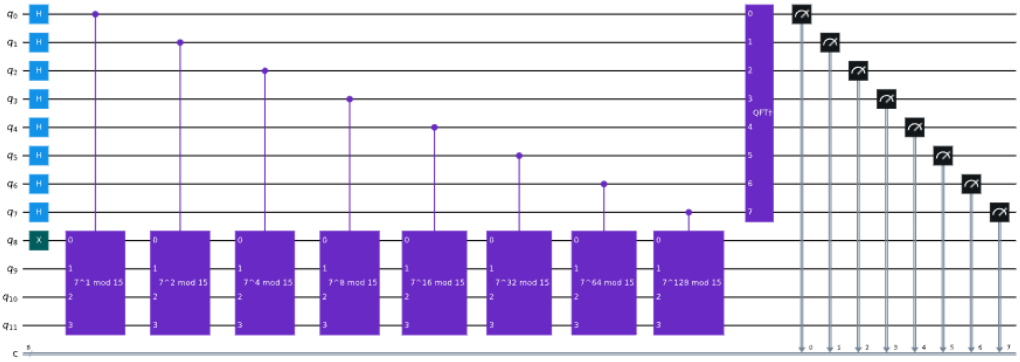
6.2.1. A Shor-algoritmus

A Shor-algoritmus továbbra is a kvantum-számítástechnika egyik legjelentősebb fejlesztése, mivel kimutatta, hogy a kvantumszámítógépek felhasználhatók a fontos, jelenleg klasszikusan nyomon követhető problémák megoldására. A Shor algoritmusa gyors módot biztosít a nagy számok kvantumszámítógép használatával történő faktorálására, amely a faktorálás nevű probléma. Számos mai kriptorendszer biztonsága azon a feltételezésen alapul, hogy nincs gyors algoritmus a faktoráláshoz. Így a Shor-algoritmus jelentős hatással volt arra, hogyan gondolunk a biztonságra egy kvantum utáni világban.

A Shor-algoritmus hibrid algoritmusként is felfogható. A kvantumszámítógép egy számításilag kemény feladat végrehajtására szolgál, amelyet pontkeresésnek nevezünk. Az időszaki megállapítások eredményeit ezután klasszikusan feldolgozzák a tényezők becsléséhez.

6.2.2. Az RSA feltörése Shor-algoritmussal

Az alábbi függvények a következő áramkört hivatottak megvalósítani:



37. ábra Áramkör Shor algoritmushoz

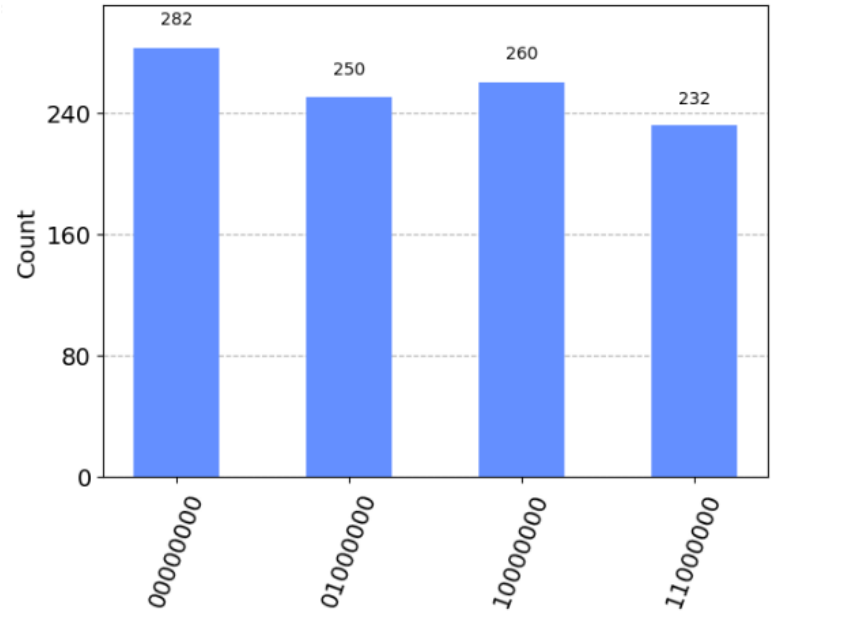
```
def c_amod15(a, power):
    """Controlled multiplication by a mod 15"""
    if a not in [2,4,7,8,11,13]:
        raise ValueError("'a' must be 2,4,7,8,11 or 13")
    U = QuantumCircuit(4)
    for _iteration in range(power):
        if a in [2,13]:
            U.swap(2,3)
            U.swap(1,2)
            U.swap(0,1)
        if a in [7,8]:
            U.swap(0,1)
            U.swap(1,2)
            U.swap(2,3)
        if a in [4, 11]:
            U.swap(1,3)
            U.swap(0,2)
        if a in [7,11,13]:
            for q in range(4):
                U.x(q)
    U = U.to_gate()
    U.name = f"{a}^{power} mod 15"
    c_U = U.control()
    return c_U
```

38. ábra a^2^j mod N létrehozása

```python
def qpe_amod15(a,N):
    """Performs quantum phase estimation on the operation a*r mod 15.
    Args:
        a (int): This is 'a' in a*r mod 15
    Returns:
        float: Estimate of the phase
    """
    N_COUNT = 8
    qc = QuantumCircuit(4+N_COUNT, N_COUNT)
    for q in range(N_COUNT):
        qc.h(q)     # Initialize counting qubits in state |+>
    qc.x(3+N_COUNT) # And auxiliary register in state |1>
    for q in range(N_COUNT): # Do controlled-U operations
        qc.append(c_amod15(a, 2**q),
                  [q] + [i+N_COUNT for i in range(4)])
    qc.append(QFT(N_COUNT).inverse(), range(N_COUNT))  # Do inverse-QFT
    qc.measure(range(N_COUNT), range(N_COUNT))
    # Simulate Results
    aer_sim = Aer.get_backend('aer_simulator')
    # `memory=True` tells the backend to save each measurement in a list
    job = aer_sim.run(transpile(qc, aer_sim), shots=100, memory=True)
    counts = job.result().get_counts()
    print(counts)
    readings = job.result().get_memory()
    print("Register Reading: " + readings[0])
    phase = int(readings[0],2)/(2**N_COUNT)
    print(f"Corresponding Phase: {phase}")
    return phase
```

39. ábra Fázis kiszámítása

A szimulátor egy ehhez hasonló eredményt fog nekünk kiadni, amiből a fázisokat úgy kaphatjuk meg, hogy ezeket az értékeket elosztjuk a $2^{qubitek\ száma}$ -val, amiben a qubitek száma jelen esetben 8, vagyis 256-tal osztjuk őket.



40. ábra Shor algoritmus szimuláció eredmények

```
rows, measured_phases = [], []
for output in counts:
    decimal = int(output, 2)  # Convert (base 2) string to decimal
    phase = decimal/(2**N_COUNT)  # Find corresponding eigenvalue
    measured_phases.append(phase)
    # Add these values to the rows in our table:
    rows.append([f"{output}(bin) = {decimal:>3}(dec)",
                 f"{decimal}/{2**N_COUNT} = {phase:.2f}"])
# Print the rows in a table
headers=["Register Output", "Phase"]
df = pd.DataFrame(rows, columns=headers)
print(df)
```

```
            Register Output           Phase
0  10000000(bin) = 128(dec)  128/256 = 0.50
1  11000000(bin) = 192(dec)  192/256 = 0.75
2  00000000(bin) =   0(dec)    0/256 = 0.00
3  01000000(bin) =  64(dec)   64/256 = 0.25
```

41. ábra Fázis eredmények Shor algoritmussal

Ezekből véletlenszerűen visszaadunk 1-et, amiből az r-t kiszámoljuk az alábbi módon és ha az r páros kiszámoljuk a hozzá tartozó gyököket, amiket úgy akpunk meg, hogy

$(a^{r/2}-1)$ és $(a^{r/2}+1)$ számoknak vesszük a modulóját.

```
def Shor(N):
    ATTEMPT = 0
    factors_array=[]
    while len(factors_array)<2:
        ATTEMPT += 1
        a=randint(0,N-1)
        print("a equals ",a)
        g=gcd(a,N)
        print(f"\nATTEMPT {ATTEMPT}:")
        if g!=1:
            continue
        phase = qpe_amod15(a,N) # Phase = s/r
        frac = Fraction(phase).limit_denominator(N)
        r = frac.denominator
        if r % 2 != 0:
                continue
        print(f"Result: r = {r}")
        if phase != 0:
            # Guesses for factors are gcd(x^{r/2} ±1 , 15)
            guesses = [gcd(a**(r//2)-1, N), gcd(a**(r//2)+1, N)]
            print(f"Guessed Factors: {guesses[0]} and {guesses[1]}")
            for guess in guesses:
                if guess not in [1,N] and (N % guess) == 0:
                    # Guess is a factor!
                    print(f'*** Non-trivial factor found: {guess} ***')
                    if guess not in factors_array:
                        factors_array.append(guess)
    return (factors_array[0],factors_array[1])
```

42. ábra Faktorok Shor-algoritmussal

Egyszerű dekódolófüggvény, valamint kulcspárgenerátor

RSA feltörése Shor algoritmussal

```
]: def mod_inverse(a, m):
       for x in range(1, m):
           if (a * x) % m == 1:
               return x
       return -1

   def generate_keypair(p,q):
       n = p * q
       phi = (p - 1) * (q - 1)
       e = random.randrange(1, phi)
       g = gcd(e, phi)
       while True:
           e = random.randrange(1, phi)
           g = gcd(e, phi)
           d = mod_inverse(e, phi)
           # e should not be equal d but I need to do this with n=15
           if g == 1:
               break

       return ((e, n), (d, n))

   def dec(code, key):
       D, N = key
       return "".join([chr(((d**D) % N) + ord('A'))
                       for d in [int(d) for d in str(code)]])
```

43. ábra Kiegészítő kód RSA-hoz 2.

```
: public, private = generate_keypair(3,5)
  print(public[0],public[1])
  print(private[0],private[1])
```

```
5 15
5 15
```

44. ábra Kulcsgenerálás

```
: encrypted_msg = 1843
  decrypted_msg = dec(encrypted_msg, private)

  print("\nDecrypted message using RSA Algorithm: " + decrypted_msg)
```

```
Decrypted message using RSA Algorithm: BIED
```

Ezt az üzenetet csak akkor lehet visszafejteni pontosan, ha ismerjük a két faktort és meg tudjuk belőle határozni phi-t, ami d kiszámításához kell.

```
: N_shor = public[1]
  assert N_shor>0,"Input must be positive"
  p,q = Shor(N_shor)
  print(p,q)
  phi = (p-1) * (q-1)
  d_shor = mod_inverse(public[0], phi)

  print("Value of d:",d_shor)
```

```
a equals  2

ATTEMPT 1:
{'10000000': 17, '11000000': 33, '01000000': 25, '00000000': 25}
Register Reading: 00000000
Corresponding Phase: 0.0
a equals  8

ATTEMPT 2:
{'10000000': 27, '01000000': 22, '00000000': 14, '11000000': 37}
Register Reading: 11000000
Corresponding Phase: 0.75
Result: r = 4
Guessed Factors: 3 and 5
*** Non-trivial factor found: 3 ***
*** Non-trivial factor found: 5 ***
3 5
Value of d: 5
```

45. ábra Faktorizálás és privát kulcs meghatározása

Látható, hogy sikeresen megkaptuk d értékét, ami a private[0] tagja. Igaz, hogy ez nem egy ideális eset, mert mind e-vel mind pedig q-val megegyezik.

```
decrypted_msg = dec(encrypted_msg,(d_shor,N_shor))

print('\nMessage Cracked using Shors Algorithm: ' + decrypted_msg + "\n")
```

```
Message Cracked using Shors Algorithm: BIED
```

46. ábra RSA feltörés ellenőrzése 1.

Sikerült visszakapni az eredeti szöveget.
Ha például elrontottam volna a számolást és nem kaptam volna meg a megfelelő d értékét, akkor nem ugyanazt a szöveget kapnám vissza.

```
decrypted_msg = dec(encrypted_msg,(7,N_shor))

print('\nMessage Cracked using Shors Algorithm: ' + decrypted_msg + "\n")
```

```
Message Cracked using Shors Algorithm: BCEM
```

47. ábra RSA feltörés ellenőrzése 2.

Habár a periodicitás miatt az is előfordulhat, hogy mégis sikerül eltalálnom egy olyan számot amivel az eredit kapom meg, de nagy számoknál erre nagyon kicsi az esély.

Pl. jelen esetben az 1 +4*i értékekre a BIED sztringet kapom vissza.

7. Összefoglalás – kitekintés a jövőre

Ahogy azt az előző fejezetekben megmutattuk, jelen pillanatban az aszimmetrikus kulcsú titkosítások bizonyulnak biztonságosabbnak, hiszen a szimmetrikus kulcsú titkosítással szemben, itt nincsen szükség a kulcs megosztására valamilyen csatornán a kommunikáció megkezdése előtt. Azonban így sem lehetünk nyugodtak a titkosításaink biztonságát illetően. Bár az RSA a jelek szerint biztonságos eljárás, azonban ahogy azt említettük jelenleg nincs bizonyítva, hogy nem létezik olyan klasszikus algoritmus, mellyel polinomiális időben feltörhető lenne. Láttuk továbbá azt is, hogy hiába derülne is ki, hogy ilyen algoritmus nem létezik, a kvantumalgoritmusok mindenképpen fenyegetik az RSA biztonságát. Jelenleg nem rendelkezünk olyan nagy bitszámú kvantumszámítógéppel, mellyel a Shor algoritmust megfelelően lehetne futtatni, azonban ez idővel változni fog. Előbb-utóbb tehát szükséges lesz új, titkosítási eljárások után néznie a szakembereknek. A megoldás valószínűleg a kvantum kulcsszétosztás lesz, mellyel visszatérünk a szimmetrikus titkosítások mezejére, azzal a különbséggel, hogy a másolhatatlansági tételt miatt, nem kell aggódnunk a közös kulcs megosztása miatt sem. Ezzel elegánsan válaszolva a kvantumos fenyegetésre egy kvantumos védelemmel.

Irodalomjegyzék

Boneh-Durfee Attack. (2023. 11. 23.). Forrás: CryptoBook: https://cryptohack.gitbook.io/cryptobook/untitled/low-private-component-attacks/boneh-durfee-attack

Gábor, D. K. (2014 - 2015). *Titkosítás, RSA algoritmus.* Dr. Kallós Gábor, Széchenyi István egyetem.

Imre Sándor, & Balázs Ferenc. (2004). *Quantum Computing and Communications: An Engineering Approach.*

math.bme.hu. (2023. 11. 23.). Forrás: Euler–Fermat-tétel, számelméleti algoritmusok: https://math.bme.hu/~vkitti/bsz1_2016_12.pdf

Péter, S. (211. 12. 15.). *GROVER-algoritmus.* Sinkovicz Péter, ELTE, Pest, Magyarország.

RSA-eljárás. (2023. 11. 23.). Forrás: Wikipédia: https://hu.wikipedia.org/wiki/RSA-elj%C3%A1r%C3%A1s

Tamás, D. (2023. 11. 23.). *Új eredmények az RSA kulcsok megfejtéséhez*. Forrás: titoktan.hu: http://www.titoktan.hu/_raktar/_e_vilagi_gondolatok/HTRSA.htm

Wikipédia. (2023. 11. 23.). *Enigma (gép)*. Forrás: Wikipédia: https://hu.wikipedia.org/wiki/Enigma_(g%C3%A9p)

Wikipédia. (2023. 11. 23.). *Nyilvános kulcsú rejtjelezés*. Forrás: Wikipédia: https://hu.wikipedia.org/wiki/Nyilv%C3%A1nos_kulcs%C3%BA_rejtjelez%C3%A9s

Wikipédia. (2023. 11. 23.). *Prímfelbontás*. Forrás: Wikipédia: https://hu.wikipedia.org/wiki/Pr%C3%ADmfelbont%C3%A1s

Wikipédia. (2023. 11. 23.). *Shor's algorithm*. Forrás: Wikipédia: https://en.wikipedia.org/wiki/Shor%27s_algorithm

Wikipédia. (2023. 11. 23.). *Szimmetrikus kulcsú rejtjelezés*. Forrás: Wikipédia: https://hu.wikipedia.org/wiki/RSA-elj%C3%A1r%C3%A1s

wikiwand. (2023. 11. 23.). *Multiplikatív rend*. Forrás: wikiwand: https://www.wikiwand.com/hu/Multiplikat%C3%ADv_rend

Quantum Learning :

https://learning.quantum.ibm.com/course/fundamentals-of-quantum-algorithms/phase-estimation-and-factoring

Qiskit Github:

https://github.com/Qiskit/textbook/tree/main/notebooks/ch-algorithms

Qiskit youtube Channel:

https://www.youtube.com/watch?v=0RPFWZj7Jm0

Github link:

https://github.com/7enTropy7/Shor-s-Algorithm_Quantum

Microsoft Learn:
https://learn.microsoft.com/hu-hu/azure/quantum/tutorial-qdk-grovers-search?tabs=tabid-visualstudio

https://learn.microsoft.com/hu-hu/azure/quantum/concepts-grovers
https://learn.microsoft.com/hu-hu/azure/quantum/user-guide/libraries/standard/applications#shors-algorithm

Ábrajegyzék: