**TÜRKİYE CUMHURİYETİ**

**YILDIZ TEKNİK ÜNİVERSİTESİ BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ**



# **BITAP ALGORITMASI**

Öğrenci No: 18069040

Öğrenci Adı Soyadı: Mehmet Semih BABACAN

## Öğrenci e-posta: l6918040@std.yildiz.edu.tr

Ders/Grup: BLM-1012/ Grup 2

YAPISAL PROGRAMLAMAYA GİRİŞ

PROJE ÖDEVİ

Ders Yürütücüsü

Ögr. Gör. Dr. Ahmet ELBİR

**Haziran, 2021**

# **İçindekiler**

1. Rapor Videosu

# BITAP Algoritması Nedir?

# Exact Searching İçin Bitap Algoritması

# BITAP ALGORITHM Nasıl Çalışır (Exact Searching için Bitap Algoritması)

# Muadili Diğer Algoritmalar

# Bitap Algoritmasının Uygulama Alanları Nelerdir

# Bitap Algoritmasının Kullanmanın Avantajları ve Dezavantajları

# Bitap Algoritmasının Karmaşıklığı ve Kısıtları

# Bitap Algoritmasının Çıktısı

# Bitap Algoritması Kodu

1. Kaynakça
2. **Rapor Videosu**

**VIDEO**

# **BITAP Algoritması Nedir?**

Bitap algoritması (shift-or algoritması ya da Baeza-Yates-Gonnet algoritması olarak da bilinir) yaklaşık bir dizi (approximate string) eşleştirme algoritmasıdır. Algoritma, desenin her öğesi için bir bit içeren bir dizi bit maskesini önceden hesaplayarak başlar. Daha sonra işin çoğunu son derece hızlı olan bitsel işlemlerle (bitwise operations) yapabilir.

Bitap algoritması EXACT SEARCHING ve FUZZY SEARCHING olmak üzere iki farklı şekilde uygulanıyor. Benim ele aldığım Bitap Algoritması Exact Searching için, yani aranan bir ibarenin metnin içinde olup olmadığını kontrol etmek için kullanılıyor. Fuzzy Searching için olan Bitap algoritması belirli bir metnin belirli bir desene "yaklaşık olarak eşit" olan bir alt dize içerip içermediğini söyler; burada yaklaşık eşitlik Levenshtein uzaklığı cinsinden tanımlanır. Alt dize ve desen birbirinden belirli bir k mesafesi içindeyse, algoritma onları eşit kabul eder.

\*En belirgin özelliği string işlemleri için Bitwise operasyonlarını kullanıyor olmasıdır.

# **Exact Searching İçin Bitap Algoritması**

Fuzzy Searching’tekinin (Fuzzy Searching Levensthein Distance kullanarak yaklasik uyusmalari/approximate matches hesaplar) aksine Exact Searching adından da anlaşılabileceği gibi tam karşılığı bulmaya çalışır.

Bitap algoritması bitler üzerinden işlem yapan bir algoritmadır. Her bir 1 değerindeki bit bir mismatch/uyuşmama ve her bir 0 değerindeki bit ise match/uyuşma ifade eder.

Algoritmanın bu match/uyuşmaları elde etmek için kullandığı yapı, C deki left-shifting operasyonunu kullanarak her bir seferinde sağ tarafa bir 0 değerinde bit eklemektir.

# **BITAP ALGORITHM Nasıl Çalışır (Exact Searching için Bitap Algoritması)**

Exact string matching algoritması, boyutları alfabenin uzunluğuna göre bir long [31] uzunluğu olan bir bit matrisinde eşleştirilecek deseni kodlayarak çalışır. Tüm Latin alfabesindeki harfleri kullanmak yerine, örnek vermek için 3 harflik bir alfabemiz olduğunu varsayalım. Mesela alfabemiz sırasıyla “a”, “b” ve “c” harflerinden oluşsun. Aranmasını istediğimiz bir “abcc” patterni aşağıdaki gibi kodlanacaktır.

1111111111111111111111111111110 # pattern/a  
1111111111111111111111111111101 # pattern/b  
1111111111111111111111111110011 # pattern/c

"a" satırında, karşılık gelen bit dizisine indexlendigi gibi, desendeki her "a" örneği için bir 0 vardır. Aynısı “b” ve “c” için de geçerlidir. “c” harfi desenin 3. ve 4. sıralarında yer alır, bu nedenle 3. ve 4. alt sıra bitleri 0'a ayarlanır. Geri kalan her şey 1'dir.

Şimdi bu patterni elimizdeki “aaabcca” metniyle karşılaştıralım. Bu karşılaştırmada ne kadar ileri gittiğimizi anlamak için ayrı bir bit dizisine/arrayine sahip olmamız gerekiyor çünkü bastan başlamamız gerekebilir ve elimizdeki değerler kaybolabilir. Daha doğrusu, bu bit dizisi/arrayi, bulunduğumuz iterasyondaki tüm uyuşmaların kaydını tutuyor.

İlk olarak bu bit dizisini/arrayini ~1 değerine eşitleriz (long formatı) (~ bitwise not anlamına geliyor, yani eşleniğini alıyor)

1111111111111111111111111111110 # durum/state

Şimdi “aaabcca” metnindeki karakterlerinin üzerinden iterasyonlarımızı yapalım. Bu iterasyonlarda Bitwise operasyonlar işin içerisine giriyor. Sırasıyla;

1) Metnin ilk harfini seç ve bitwise-or operasyonunu elimizdeki state/durumumuz ve yeni pattern bit matrisiyle uygula.

2) Duruma/state’e left-shift operasyonunu 1 ile uygula. (1 ile uygulamak 2^1 değeriyle çarpmak anlamına gelir)

Metin **a**aabcca  
1111111111111111111111111111110 # pattern/a  
1111111111111111111111111111110 # eski durum/state  
1111111111111111111111111111100 # yeni durum/state

Görüldüğü gibi bit seviyesinde or işlemi yapıldı ve 2 ile çarpım sonucu (left shifting) bir bit sola kaydırıldı.

Hâlihazırdaki durumda bir tane 1 uyumsuzluk/mismatch belirtirken bir tane 0 biti de uyuşma/match belirtir. Şimdiye kadar bir tane “a” ile uyuşma/match bulduk ve bundan sonra herhangi bir uyuşma/match var mı, bunu da sırayla iter asyön işlemlerimize devam ederek göreceğiz.

Text a**a**abcca  
1111111111111111111111111111110 # pattern/a  
1111111111111111111111111111100 # eski durum/state  
1111111111111111111111111111100 # yeni durum/state

Hala yalnızca “a” için uyuşma/match’e sahibiz.

Text aa**a**bcca  
1111111111111111111111111111110 # pattern/a  
1111111111111111111111111111100 # eski durum/state  
1111111111111111111111111111100 # yeni durum/state

Şimdi sıradaki iterasyonda “b” harfi var.

Text aaa**b**cca  
1111111111111111111111111111101 # pattern/b  
1111111111111111111111111111100 # eski durum/state  
1111111111111111111111111111010 # yeni durum/state

Text aaab**c**ca  
1111111111111111111111111110011 # pattern/c  
1111111111111111111111111111010 # old state  
1111111111111111111111111110110 # new state

Text aaabc**c**a  
1111111111111111111111111110011 # pattern/c  
1111111111111111111111111110110 # old state  
1111111111111111111111111101110 # new state

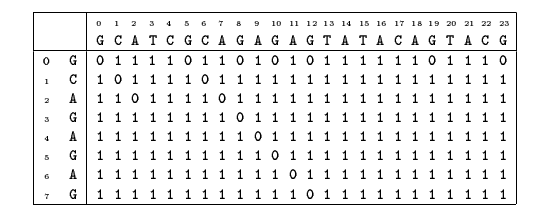
Varsayalım ki verdiğimiz patternin (aranacak ibarenin) uzunluğu n kadar olsun. Her bir iterasyonda yeni durum/state’i kontrol etsin. Bu kontrollerde acaba n+1. en değersiz bit 0 mi değil mi ile ilgili olsun. Eğer ki 0 ise bu da aradığımız pattern tamamlanmış, yani aradığımıza ulaşmış olduğumuz anlamına gelir.

ÖRNEK:

Metin**->>** GCATCGCAGAGAGTATACAGTACG

Aranacak dizi/pattern**->>** GCAGAGAG

Yukarıda bahsedilen işlemler, her bir karakter için teker teker yapılacak ve çıkan sonuç da aşağıdaki matris tablosunda incelenebileceği gibi bize (eğer ki varsa) aradığımız patterni verecektir. Art arda gelen 0’lara göre bakılırsa, indexin 5 olduğu konumda aradığımız patterni bulmuş olduk.

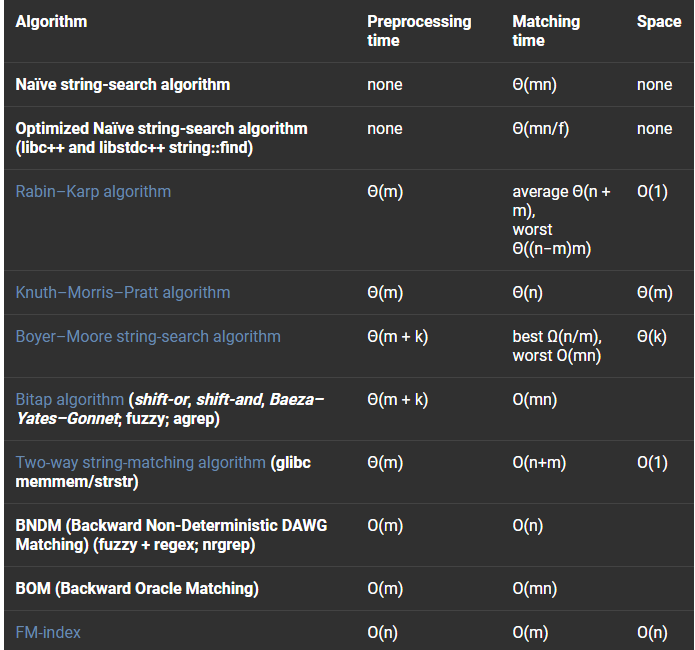


OZET OLARAK:

* Bir stringi pattern olarak girin.
* Bu string, basit bir Char Dizisine dönüştürülecek.
* Eğer ki patternin uzunluğu 0 ya da 31’i geçiyorsa, “-1” yani “metnin içinde bulunamamıştır” döndürülür.
* R değişkenine 1 atanır ve eşleniği alınır.
* “patternMask” dizisi oluşturulur ve her değerine “0” atanır, ardından her bir değer için eşleniği alınır.
* Patterndeki karakterlerin ASCII karşılıklarını patternMask icin index değeri olarak kullanarak patterndeki her bir karakter için uyusan değerler icin 0 ve uyuşmayanlar için 1 değerleri gelen bit dizileri oluşturulur.
* Bundan sonrasında diziyi metnin uzunluğu kadar bir döngü içerisinde döndürmek gerekiyor.
* Her bir adımda R ve patternMask ile birlikte bitwise or işlemi yapılır.
* Şimdi de 1UL’yi patternin uzunluğuyla left-shifting yaparak elde edilen değerle elimizdeki R değerini bitwise-and işlemi yaparak kontrol ederiz.
* Eğer ki i - len + 1, döngü bitince bulunamadıysa da -1 dondurur. İ - >> döngüde bulunduğumuz adım, len - >> pattern uzunluğu
* Çıktı metnin içinde aradığımız patternin metnin içindeki indexi yada -1’dir.

# **Muadili Diğer Algoritmalar**

Bu başlık altında Bitap algoritması yerine muadili olarak kullanabilecek exact string-matching/searching algoritmalarından bazıları ele alınacaktır ve bazıları da sıralanacaktır.



m-> metin/text uzunluğu

n-> pattern/aranan uzunluğu

1. Naïve String-Search Algoritması

Belirli bir metinde (veya metnin gövdesinde) bir kalıbın tüm oluşumlarını bulma.

Naïve String-Search Algoritmasi, diğer örüntü arama algoritmaları arasında en basit yöntemdir. Ana metnin tüm karakterlerini pattern ile kontrol eder.

Naïve String-Search Algoritmasi, tam dize eşleştirme (bir metindeki bir patternin bir veya tüm kesin oluşumlarını bulmak anlamına gelir) algoritmasıdır.

Bu algoritma daha küçük metinler için yararlıdır. Herhangi bir ön işleme aşamasına ihtiyaç duymaz. Dize için bir kez kontrol ederek alt diziyi bulabiliriz. Ayrıca işlemi gerçekleştirmek için fazladan yer kaplamaz.

1. Rabin-Karp Algoritması

Rabin–Karp algoritması veya Karp–Rabin algoritması, Richard M. Karp ve Michael O. Rabin (1987) tarafından oluşturulan ve bir metindeki bir pattern dizisinin tam eşleşmesini bulmak için hashing kullanan bir string-searching algoritmasıdır. Kalıpla eşleşmeyen metnin konumlarını hızla filtrelemek için yuvarlanan bir karma kullanır ve ardından kalan konumlarda bir eşleşme olup olmadığını kontrol eder. Aynı fikrin genellemeleri, tek bir kalıbın birden fazla eşleşmesini bulmak veya birden fazla kalıbın eşleşmelerini bulmak için kullanılabilir.

1. Boyer-Moore String-Search Algoritması

Boyer-Moore string-searching algoritması, pratik string arama literatürü için standart ölçüt olan verimli bir dizi arama algoritmasıdır. Robert S. Boyer ve J Strother Moore tarafından 1977 yılında geliştirilmiştir. Algoritma, aranan metni (pattern) önceden işler, ancak içinde aranan stringi (pattern) işlemez. Bu nedenle, pattern metinden çok daha kısa olduğu veya birden çok aramada kalıcı olduğu uygulamalar için çok uygundur. Boyer-Moore algoritması, metnin bölümlerini atlamak için ön işlem adımı sırasında toplanan bilgileri kullanır, bu da diğer birçok dizi arama algoritmasından daha düşük bir sabit faktörle sonuçlanır. Genel olarak, pattern uzunluğu arttıkça algoritma daha hızlı çalışır. Algoritmanın temel özellikleri, pattern başı yerine kuyruğunda eşleşmesi ve metindeki her bir karakteri aramak yerine, birden çok karakter atlamalarında metin boyunca atlamasıdır.

1. Two-Way String-Matching Algoritması

İki Two-Way String-Matching algoritması, ileriye dönük Knuth–Morris–Pratt algoritması ile geriye doğru çalışan Boyer–Moore string/dizi arama algoritmasının bir kombinasyonu olarak görülebilen verimli bir dizi arama algoritmasıdır. Maxime Crochemore ve Dominique Perrin bu algoritmayı 1991'de icat etti. En kotu durum karmaşıklığı O(n)’dir.

1. Bazı Örnek Algoritmalar: (devamı için bknz. Handbook of Exact String-Matcing Algorithms)

* Brute Force Algoritması
* Morris-Pratt Algoritması
* Knuth-Morris-Pratt Algoritması
* Simon Algoritması
* Colussi Algoritması
* Galil-Giancarlo Algoritması
* Apostolico-Crochemore Algoritması
* Not So Naïve Algoritması
* Horspool Algoritması
* Zhu-Takaoka Algoritması
* Berry-Ravindran Algoritması
* Smith Algoritması
* Raita Algoritması

# Bitap Algoritmasının Uygulama Alanları Nelerdir

1. Genome Dizilimi Analizinde Bitap Algoritmasi kullanilir: Biyoinformatikte dizi analizi, bir DNA, RNA veya peptit dizisinin özelliklerini, işlevini, yapısını anlamak için çok çeşitli analitik yöntemlerden herhangi birine tabi tutma işlemidir. Kullanılan metodolojiler, dizi hizalamayı, biyolojik veri tabanlarına karşı araştırmaları içerir.

( <https://people.inf.ethz.ch/omutlu/pub/GenASM-approximate-string-matching-framework-for-genome-analysis_ARMSummit20-long-talk.pptx>) ( <https://www.researchgate.net/figure/Example-for-the-Bitap-algorithm_fig1_344276683>)

1. Bitlerle yani düşük seviye işlem yaptığı ve hızlı olduğu için web uygulamalarında kullanılır.
2. Bitap algoritması (Fuzz Searching için olanı) Unix terminali aracı olan Grep’te bulunan algoritmalardan birisidir ve regular expressionslarla başa çıkmak için kullanılmıştır. Udi Manber, Sun Wu ve Burra Gopal tarafından yazılmıştır.
3. Bir dokumanın içinde arama yapmak için kullanılabilir.
4. İntihal (Plagiarism) belirlemede kullanılıyor. (Bitap Fuzzy String Search algoritması için de söz konusu)
5. Herhangi bir programın yapısında normal bir arama algoritması olarak kullanılabilir.

# Bitap Algoritmasının Kullanmanın Avantajları ve Dezavantajları

## Avantajları:

* Neredeyse yapılan tüm işlemler bitwise yani bit seviyesinde işlemler, bu da algoritmayı oldukça hızlı yapıyor.
* Girdi uzunluğu az olan string-matching için oldukça uygun bir algoritmadır.
* Hardware de kullanılabilecek bir algoritmadır.

## Dezavantajları:

* Girdilerin ve alfabenin karakter sayısı arttıkça algoritma daha yavaş çalışıyor, karakter sayısı yüksek olan alfabelerde alınan hız verimi düşüyor.
* Aranacak girdilerin ait olduğu alfabenin içerisindeki harflerin sayısının biliniyor olması gerekiyor.
* Aranacak patternin karakter uzunluğu bir makine Word’unun (eski bilgisayar yapıları için) içerisine girebilecek kadar kisa olmali. Fakat gündüzümdeki gelişmiş bilgisayarlar ile birlikte bu rakam 32 bytlelik ya da 64 bytelik veri tipleri dolayısıyla 32 ya da 64 karakter uzunluğuna kadar aranacak patternin uzunluğunu destekliyor. Yine de aranacak pattern 32 veyahut 64 uzunluğunu geçemez ve bu limitlerin altında olmalıdır.
* Patternin bulunduğu ilk indexi döndürür ve algoritma böylece sonlanmış olur. Birden çok aranan patterin bulunduğu metinleri için verimli bir yöntem değil. Optimize edilmesi gerekiyor.
* İçinde arama yapılacak metnin her bir değeri yeni bir dizi oluşturulur. Bu da ekstra hafıza kullanmaya neden olur.

# Bitap Algoritmasının Karmaşıklığı ve Kısıtları

m – pattern uzunluğu

n – text/metin uzunluğu

Algoritmanın gerektirdiği veri yapıları nedeniyle, sabit bir uzunluktan (tipik olarak söz konusu makinenin WORD veri tipi uzunluğu) daha az patternlarda en iyi performansı gösterir ve ayrıca küçük bir alfabe yerine girdileri tercih eder. Belirli bir alfabe ve kelime uzunluğu m için bir kez uygulandıktan sonra, çalışma süresi tamamen tahmin edilebilir – metnin yapısı veya pattern ne olursa olsun O(mn) işlemlerinde çalışır.

Bu algoritmanın 31 veya daha az karakterden oluşan dizilerle sınırlandırılması nedeniyle, çalışma zamanı O(1)'de denebilir. Gereken bit düzeyinde işlem sayısı 2n'dir, burada n metnin uzunluğudur.

Sonuç: Algoritma her daim **O(mn)** suresinde çalışıyor ve **O(m + k)** kadar bir sure de preprocessing için harcanır.

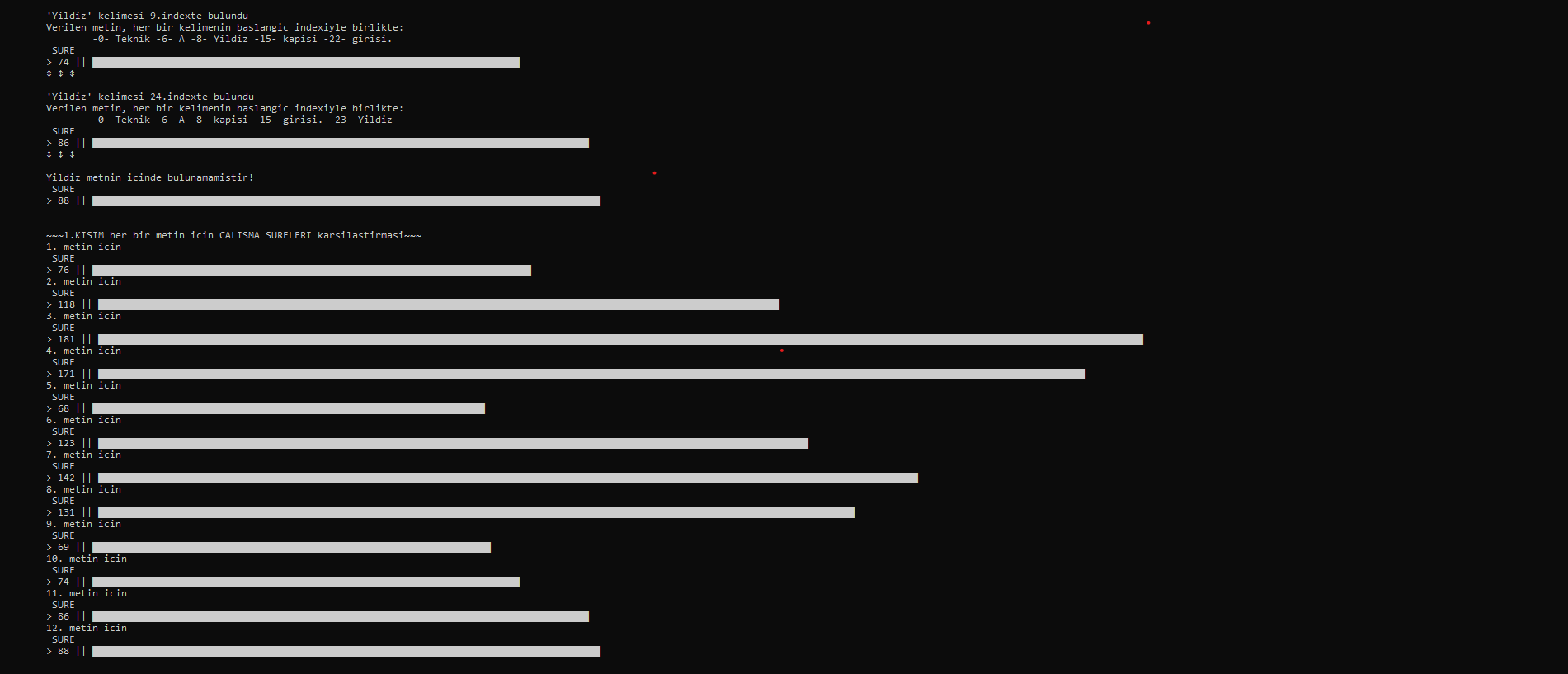
Kısıtları:Kullanılan bilgisayarın mimarisine göre 8bit, 32bit, 64bit gibi bitlerden oluşan WORD tipi veri tipleri oluşturulacağı için patternin bu durumu uzunluklardan herhangi birini geçmemesi gerekiyor çünkü aksi halde işlem yapılamaz. Aranacak metnin her daim uzunluğu ve bilgisayarın mimarisi göz önünde bulundurulması gerekiyor.

# **Bitap Algoritmasının Çıktısı**

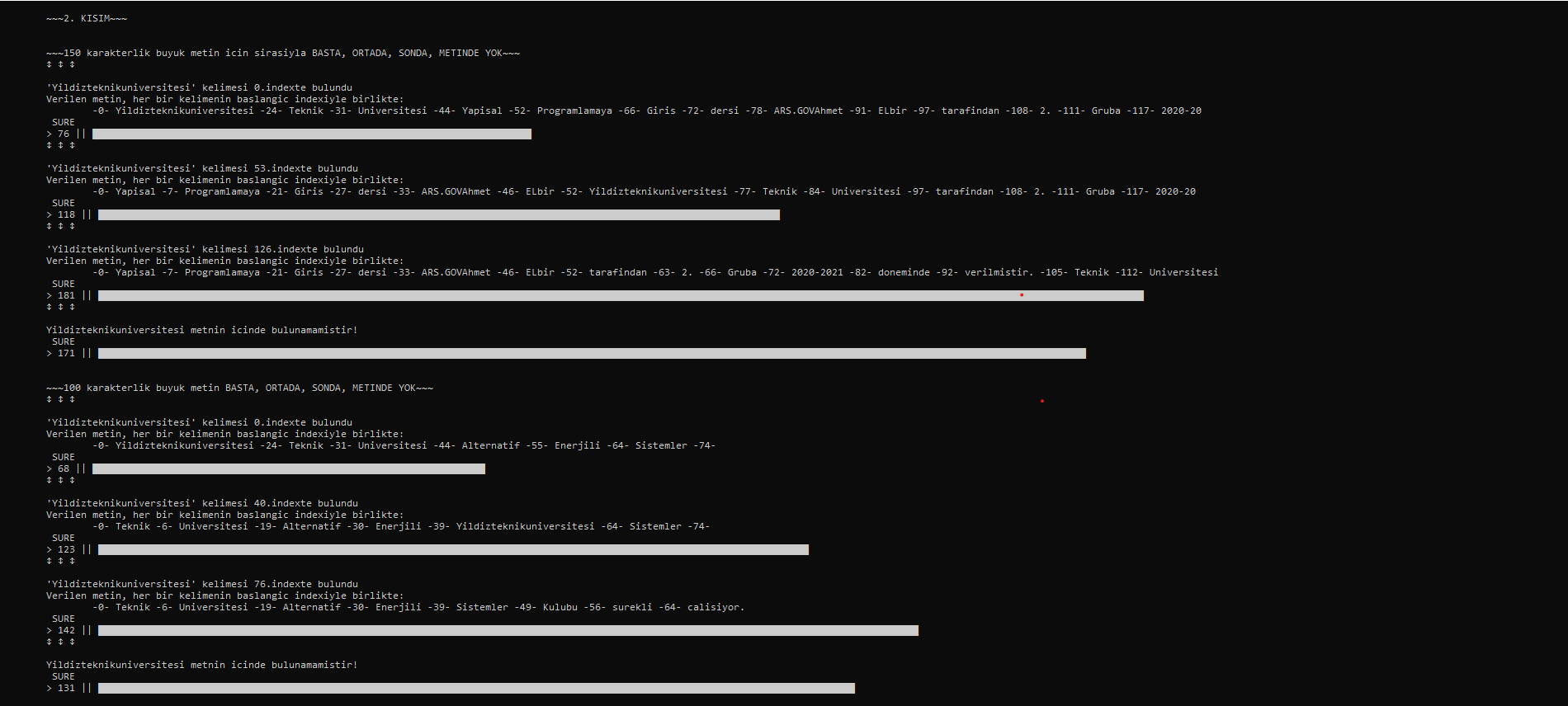
NOT: Çözünürlüğün zayıf olmasından dolayı, resimlerin daha detaylı incelenebilmesi için Google Drive linkleri de altlarında ayrıca belirtilmiştir.

# 

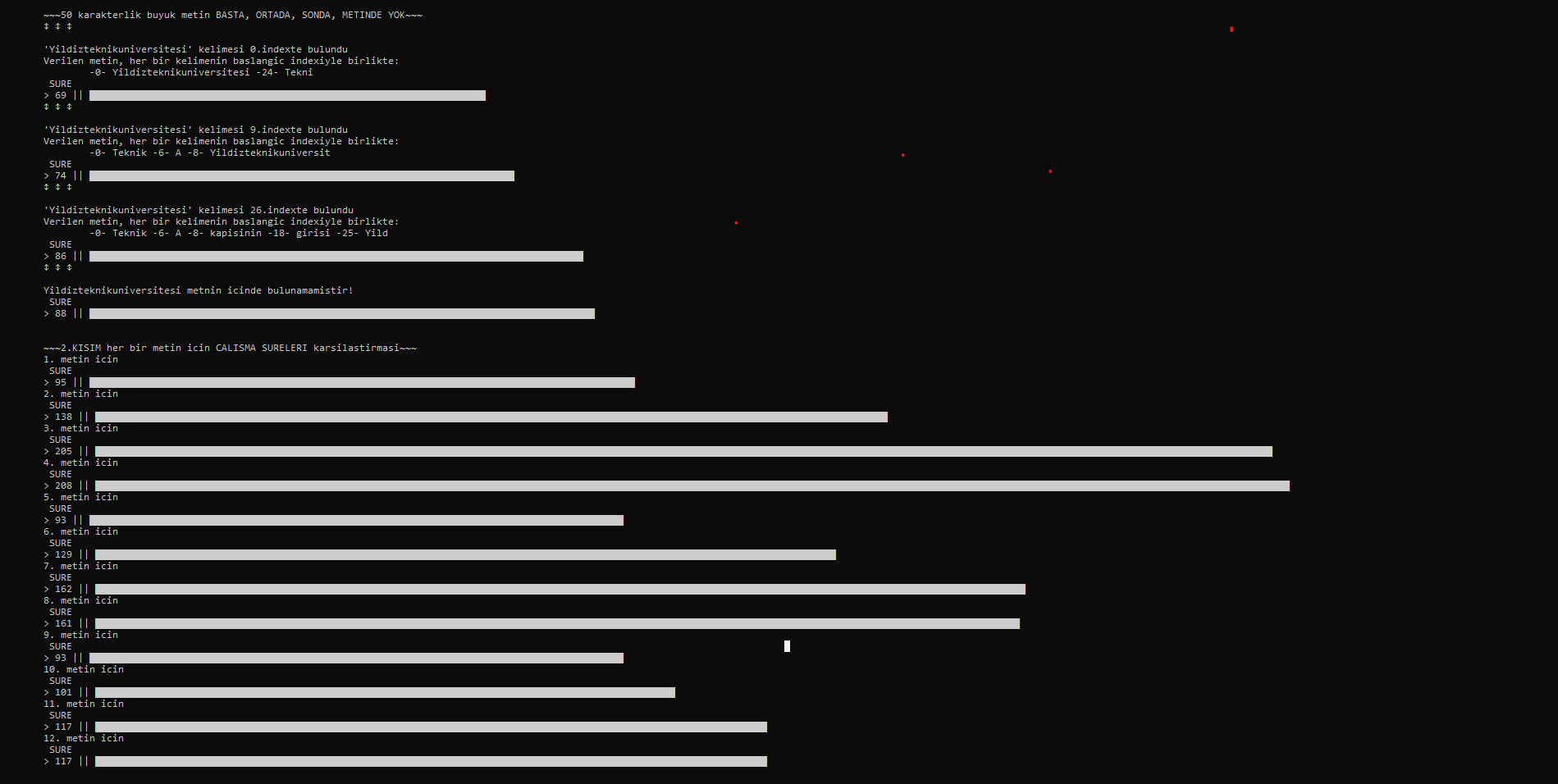
<https://drive.google.com/file/d/1aumTEEFfRbVIzzS77hWfnOfEV94O42AQ/view?usp=sharing>



<https://drive.google.com/file/d/1j5hoSm-Qx2AdSlS7yB9X1aTBdqgk5mj8/view?usp=sharing>



<https://drive.google.com/file/d/1swPTh30hfNjTduqhXhMBJs5giV__mc0N/view?usp=sharing>



<https://drive.google.com/file/d/1kFX0QubTM5RwFjq2WYzalvNJSONOBoAk/view?usp=sharing>



<https://drive.google.com/file/d/1WZAUER3puS7Ymcm38rjnZn4M5AzxsmLj/view?usp=sharing>

# **Bitap Algoritması Kodu**

Dosyanın Google DrivelLinki: <https://drive.google.com/file/d/153OV0uT-rmBmU_C-itTc0uDC9iQDFunc/view?usp=sharing>

#include <stdio.h>

#include <string.h>

#include <limits.h>

#include <time.h>

#define SINIR 100000

int Bitap(const char \*text, const char \*pattern);

/\* Bitap fonksiyonu, BITAP Algoritmasi kullanarak

verilen -text- icinde, aranan -pattern- in olup olmadigini

kontrol eder.

\*/

void IndexYazdirici(const char \*text, const int max, const char \*aranan, const int index);

/\* IndexYazdirici fonksiyonu, -text- icerisindeki

her bir kelimenin baslangic indexini yazdirir,

boylece kullanici bulunan index degeriyle

metinde bulunan kelimelerin baslangic indexlerini

karsilastirabilir.

\*/

void GrafikCizici(const float miktar);

/\* GrafikCizici fonksiyonu, -miktar\_dizi- icerisinde 3 \* 4 = 12

farkli kosulu icinde tutar. Rowlarda 3 tane farkli text,

ve her bir column icinde de metnin 4 farkli varyasyonunun

Bitap fonksiyonu ile yapilan islemlerin yapilma suresini tutar.

\*/

int main(){

int i, j;

clock\_t start, end;

// 3 \* 4 = 12 tane durum icin sure degeleri, -sureler- icine atilir.

double sureler[12];

// 3 \* 4 = 12 tane durum icin index degeleri, -indexler- icine atilir.

int indexler[12];

/\* 3 tane text sirasiyla 4 farkli sekilde olusturulmustur

1. text ->> 125 karakterden olusmaktadir.

2. text ->> 75 karakterden olusmaktadir

3. text ->> 30 karakterden olusmaktadir.

3 farkli text, her biri icin 4 tane varyansi ile birlikte

12 elemanlik textler dizisinin icerisine alinmistir

\*/

char \*textler[12];

// 1. kisim /// ->> aranacak kelime 'Yildiz' 6 karakterden olusmaktadir

// 125 karakterlik buyuk boyda text

printf("\t~~~1. KISIM~~~");

textler[0] = "Yildiz Teknik Universitesi Yapisal Programlamaya Giris dersi Ahmet ELbir tarafindan 2. Gruba 2020-2021 doneminde verilmistir.";

textler[1] = "Yapisal Programlamaya Giris dersi Ahmet ELbir Yildiz Teknik Universitesi tarafindan 2. Gruba 2020-2021 doneminde verilmistir.";

textler[2] = "Yapisal Programlamaya Giris dersi Ahmet ELbir tarafindan 2. Gruba 2020-2021 doneminde verilmistir. Teknik Universitesi Yildiz";

textler[3] = "Yildim Teknik Universitesi Yapisal Programlamaya Giris dersi Ahmet ELbir tarafindan 2. Gruba 2020-2021 doneminde verilmistir.";

// 75 karakterlik orta boyda text

textler[4] = "Yildiz Teknik Universitesi Alternatif Enerjili Sistemler Kulubu calisiyor..";

textler[5] = "Teknik Universitesi Alternatif Enerjili Yildiz Sistemler Kulubu calisiyor..";

textler[6] = "Teknik Universitesi Alternatif Enerjili Sistemler Kulubu calisiyor. Yildiz.";

textler[7] = "Yildim Teknik Universitesi Alternatif Enerjili Sistemler Kulubu calisiyor..";

// 30 karakterlik kucuk boyda text

textler[8] = "Yildiz Teknik A kapisi girisi.";

textler[9] = "Teknik A Yildiz kapisi girisi.";

textler[10] = "Teknik A kapisi girisi. Yildiz";

textler[11] = "Yildim Teknik A kapisi girisi.";

/\* text icerisinde aranilmasi istenilen kisim aynidir,

4 tane aranacak metin icin ortak durumlar sirasiyla aranan:

1) basta (ilk kelime olarak)

2) neredeyse ortada (kelime baslangici olarak, 3 kelimenin 2.si gibi)

3) sonda (son kelime olarak)

4) metnin icinde yok

olacak sekilde ayarlanmistir.

\*/

// aranacak ibare word degiskeni icerisindedir icerisindedir.

char \*word = "Yildiz";

/// buradan sonra algoritma calisacak

for (i = 0; i < 12; i++){

// hesaplanan sureler, -sureler-in icerisine

// hesaplanan indexler, -indexler-in icerisine atiliyor

start = clock();

/\* kodun burada SINIR=100000 kere calismasinin sebebi

kodun cok hizli calismasi ve gozle gorulur bir deger elde

edilememesidir. Dongu sonucu bir deger elde edilebilmektedir.

\*/

for (j = 0; j < SINIR; j++){

indexler[i] = Bitap(textler[i], word);

}

end = clock();

sureler[i] = (float)(end - start);

}

int value;

for (i = 0; i < 12; i++){

if (i == 0){

printf("\n\n\n\t~~~125 karakterlik buyuk metin icin sirasiyla BASTA, ORTADA, SONDA, METINDE YOK~~~");

value = 125;

}

if (i == 4){

printf("\n\n\n\t~~~75 karakterlik buyuk metin BASTA, ORTADA, SONDA, METINDE YOK~~~");

value = 75;

}

if (i == 8){

printf("\n\n\n\t~~~30 karakterlik buyuk metin BASTA, ORTADA, SONDA, METINDE YOK~~~");

value = 30;

}

IndexYazdirici(textler[i], value, word, indexler[i]);

GrafikCizici(sureler[i]);

}

printf("\n\n\n\t~~~1.KISIM her bir metin icin CALISMA SURELERI karsilastirmasi~~~");

for(i = 0; i < 12; i++){

printf("\n\t%d. metin icin", i + 1);

GrafikCizici(sureler[i]);

}

// 2. kisim /// ->> aranacak kelime 'Yildizteknikuniversitesi' 24 karakterden olusmaktadir

char \*textler2[12];

int sureler2[12];

int indexler2[12];

printf("\n\n\n\t~~~2. KISIM~~~");

/\* 3 tane text sirasiyla 4 farkli sekilde olusturulmustur

1. text ->> 150 karakterden olusmaktadir.

2. text ->> 100 karakterden olusmaktadir

3. text ->> 50 karakterden olusmaktadir.

3 farkli text, her biri icin 4 tane varyansi ile birlikte

12 elemanlik textler dizisinin icerisine alinmistir

\*/

// 150 karakterlik buyuk boyda text

textler2[0] = "Yildizteknikuniversitesi Teknik Universitesi Yapisal Programlamaya Giris dersi ARS.GOVAhmet ELbir tarafindan 2. Gruba 2020-2021 doneminde verilmistir.";

textler2[1] = "Yapisal Programlamaya Giris dersi ARS.GOVAhmet ELbir Yildizteknikuniversitesi Teknik Universitesi tarafindan 2. Gruba 2020-2021 doneminde verilmistir.";

textler2[2] = "Yapisal Programlamaya Giris dersi ARS.GOVAhmet ELbir tarafindan 2. Gruba 2020-2021 doneminde verilmistir. Teknik Universitesi Yildizteknikuniversitesi";

textler2[3] = "Yildimteknikuniversitesi Teknik Universitesi Yapisal Programlamaya Giris dersi ARS.GOVAhmet ELbir tarafindan 2. Gruba 2020-2021 doneminde verilmistir.";

// 100 karakterlik orta boyda text

textler2[4] = "Yildizteknikuniversitesi Teknik Universitesi Alternatif Enerjili Sistemler Kulubu surekli calisiyor.";

textler2[5] = "Teknik Universitesi Alternatif Enerjili Yildizteknikuniversitesi Sistemler Kulubu surekli calisiyor.";

textler2[6] = "Teknik Universitesi Alternatif Enerjili Sistemler Kulubu surekli calisiyor. Yildizteknikuniversitesi";

textler2[7] = "Yildimteknikuniversitesi Teknik Universitesi Alternatif Enerjili Sistemler Kulubu surekli calisiyor.";

// 50 karakterlik kucuk boyda text

textler2[8] = "Yildizteknikuniversitesi Teknik A kapisinin girisi";

textler2[9] = "Teknik A Yildizteknikuniversitesi kapisinin girisi";

textler2[10] = "Teknik A kapisinin girisi Yildizteknikuniversitesi";

textler2[11] = "Yildimteknikuniversitesi Teknik A kapisinin girisi";

/\* text icerisinde aranilmasi istenilen kisim aynidir,

4 tane aranacak metin icin ortak durumlar sirasiyla aranan:

1) basta (ilk kelime olarak)

2) neredeyse ortada (kelime baslangici olarak, 3 kelimenin 2.si gibi)

3) sonda (son kelime olarak)

4) metnin icinde yok

olacak sekilde ayarlanmistir.

\*/

// aranacak ibare word degiskeni icerisindedir icerisindedir.

char \*word2 = "Yildizteknikuniversitesi";

/// buradan sonra algoritma calisacak

for (i = 0; i < 12; i++){

// hesaplanan sureler, -sureler-in icerisine

// hesaplanan indexler, -indexler-in icerisine atiliyor

start = clock();

/\* kodun burada SINIR=100000 kere calismasinin sebebi

kodun cok hizli calismasi ve gozle gorulur bir deger elde

edilememesidir. Dongu sonucu bir deger elde edilebilmektedir.

\*/

for (j = 0; j < SINIR; j++){

indexler2[i] = Bitap(textler2[i], word2);

}

end = clock();

sureler2[i] = (float)(end - start);

}

for (i = 0; i < 12; i++){

if (i == 0){

printf("\n\n\n\t~~~150 karakterlik buyuk metin icin sirasiyla BASTA, ORTADA, SONDA, METINDE YOK~~~");

value = 125;

}

if (i == 4){

printf("\n\n\n\t~~~100 karakterlik buyuk metin BASTA, ORTADA, SONDA, METINDE YOK~~~");

value = 75;

}

if (i == 8){

printf("\n\n\n\t~~~50 karakterlik buyuk metin BASTA, ORTADA, SONDA, METINDE YOK~~~");

value = 30;

}

IndexYazdirici(textler2[i], value, word2, indexler2[i]);

GrafikCizici(sureler[i]);

}

printf("\n\n\n\t~~~2.KISIM her bir metin icin CALISMA SURELERI karsilastirmasi~~~");

for(i = 0; i < 12; i++){

printf("\n\t%d. metin icin", i + 1);

GrafikCizici(sureler2[i]);

}

printf("\n\n\n\t~~~1. ve 2.KISIM her bir metin icin CALISMA SURELERI karsilastirmasi~~~");

for(i = 0; i < 2; i++){

if (i == 0){

printf("\n\n\t1.KISIM");

for (j = 0; j < 12; j++){

printf("\n\t%d. metin icin", j + 1);

GrafikCizici(sureler[j]);

}

}

else{

printf("\n\n\n\t2.KISIM");

for (j = 0; j < 12; j++){

printf("\n\t%d. metin icin", j + 1);

GrafikCizici(sureler2[j]);

}

}

}

return 0;

}

int Bitap(const char \*text, const char \*pattern)

{

/\*

return degeri bir -index- ise, aranan bulundu anlamina gelir

return degeri -1 ise aranan bulunmadi anlamina gelir

\*/

int n = strlen(pattern);

unsigned long R;

unsigned long patternMask[CHAR\_MAX + 1]; // 128 e kadar olan ascii karakterler dahil

int i;

if (pattern[0] == '\0')

return 0;

//HATA: aranan pattern'in boyu 32 bitten fazla olamaz, yani 0 ile 31!

if (n > 31)

return -1;

// sayinin eslenigini alir ve isaret degistirir

// unsigned icin ilk bit haric diger tum bitler

// 1 olmus olur, deger ->> 4,294,967,294 (2^32 - 2)

R = ~1;

/\* tum degerler eslenikleri alinakarak

unsigned long icin max degere esitlendi

4,294,967,295 (2^32 - 1)

\*/

for (i = 0; i <= CHAR\_MAX; ++i)

patternMask[i] = ~0;

/\*

1UL -> yalnizca ilk biti 1 olan unsigned long ifade eder

x << y -> bitwise left shift, yani x \* (2^y) islemi yapilir

&= -> bitwise and, x &= y isleminde bit seviyesinde -and- yapilir

|= -> bitwise or, x |= y isleminde bit seviyesinde -or- yapilir

\*/

for (i = 0; i < n; ++i)

patternMask[pattern[i]] &= ~(1UL << i);

// i kadar 1UL sola oteleniyor

// her seferinde 1 oteleme miktari artiyor, 2 kere 3 kere gibi

// her bir karakter icin kaydedilen ve tekrar hesaplanirsa guncellenen -pattern- yapisi patternMask'a atiliyor

for (i = 0; text[i] != '\0'; ++i){

//R hala ~1'e esit

// her seferinde R'in degeri guncelleniyor bitwise or islemi ile

R |= patternMask[text[i]];

// ve bir sola shift-left ile kaydiriliyor, yani en anlamsiz bite bir 0 ekleniyor

R <<= 1;

if (0 == (R & (1UL << n))) // her seferinde n kadar bitin eklendigi 1UL ile mathch yani 0 larin uyusup uyusmadigi kontrol ediliyor.

// uyustugu vakit bu degerimizi return edebiliriz anlamina geliyor ve indexi bulmus oluyoruz.

return (i - n) + 1;

}

//pattern bulunmaz ise -1 return ediliyor, bu da arananin bulunmadigi anlamina geliyor

return -1;

}

void IndexYazdirici(const char \*text, const int max, const char \*aranan, const int index){

/\*

verilen indexe gore, -aranan- in bulunup bulunmadigini soyler

-aranan- bulundu ise her bir kelimeyi baslangic indexi

ile birlikte yazdirir.

\*/

int i;

printf("\n\t%c %c %c", 18, 18, 18);

if (index != -1){

printf("\n\n\t'%s' kelimesi %d.indexte bulundu", aranan, index);

printf("\n\tVerilen metin, her bir kelimenin baslangic indexiyle birlikte: ");

printf("\n\t\t-0- ");

for (i = 0; i < max; i++){

if (text[i] == ' ')

printf(" -%d- ", i);

else

printf("%c", text[i]);

}

}

else{

printf("\n\n\t%s metnin icinde bulunamamistir!", aranan);

}

}

void GrafikCizici(const float miktar){

/\*

verilen miktar kadar 219 kodlu karakteri yazdirir ve bir bar-plotting yapar.

\*/

int i;

int int\_miktar = miktar;

printf("\n\t SURE\n");

printf("\t> %d || ", int\_miktar);

for (i = 0; i < int\_miktar; i++){

printf("%c", 219);

}

}

1. **) KAYNAKCA**

* <https://en.wikipedia.org/wiki/String-searching_algorithm>
* <https://en.wikipedia.org/wiki/Bitap_algorithm#:~:text=The%20algorithm%20tells%20whether%20a,the%20algorithm%20considers%20them%20equal>.
* <https://www.cs.auckland.ac.nz/courses/compsci369s1c/lectures/GG-notes/CS369-StringAlgs.pdf>
* <https://en.wikipedia.org/wiki/Bitap_algorithm#:~:text=The%20algorithm%20tells%20whether%20a,the%20algorithm%20considers%20them%20equal>.
* <https://www.slideshare.net/SwapanShakhari/string-matching-algorithmspattern-matching>
* <https://www.wikizero.com/en/Bitap_algorithm>
* <https://memim.com/bitap-algorithm.html>
* <http://www.wucathy.com/blog/?p=680>
* <https://www-igm.univ-mlv.fr/~lecroq/string/node6.html>
* <https://www.cs.bgu.ac.il/~sa201/wiki.files/lec16.pdf>
* Handbook of Exact String-Matcing Algorithms by Christian Charras and Thierry Lecroq