İçindekiler

Özet...................................................................................................................................2

Giriş...................................................................................................................................3

Bölüm 1:Dizi Sıralama Algoritmaları BubbleSort ve QuickSort’un karşılaştırılması.4

BubbleSort Pseudocode’u.....................................................................................4

BubbleSort Teorik Analiz.....................................................................................4

QuickSort Pseudocode’u......................................................................................4

QuickSort Teorik Analiz.......................................................................................5

Bölüm 1 Gerçekleştirimi......................................................................................5

BubbleSort Teorik ve Deney Sonuçları................................................................6

QuickSort Teorik Sonuçları ve Deney Sonuçları.................................................7

Grafikler...............................................................................................................8

Bölüm 2: String Eşleştirme Algoritmalarının Karşılaştırılması.......................................10

BruteForce String Match Pseudocode’u:.............................................................10

BruteForceStringMatch Teorik Analiz.................................................................10

Horspool Pseudocode’u.......................................................................................11

Horspool Teorik Analiz.......................................................................................11

Boyer-Moore Pseudocode’u................................................................................12

Boyer-Moore Teorik Analiz................................................................................12

Bölüm 2.nin Gerçekleştirimi................................................................................13

Bölüm 3: Matrix Chain Product Algoritmalarının Karşılaştırılması...............................14

In-order Multiplying Pseudocode’u...................................................................14

Dynamic Matrix Chain Product Pseudocode’u..................................................14

Dynamic Matrix Chain Product Teorik Analiz..................................................15

3.Bölüm Gerçekleştirimi....................................................................................15

Sonuçlar..............................................................................................................17

Kaynakça...................................................................................................... ......18

**Özet**

Üç ana kısımdan oluşan bu Algoritma Analizi projesinde, ilk olarak verilen bir dizi içerisindeki sayıları sıralama algoritmalarından olan BubbleSort ve QuickSort algoritmaları, etkinlik ve çalışma zamanları göz önünde bulundurularak ele alınmıştır.

İkinci kısımda String eşleştirme algoritması olarak Bruteforce, Horspool ve Boyer-Moore algoritmaları incelenmiştir.

Üçüncü kısımda Matrix Chain probleminin çözümüne, verilen sıra ile ve Dynamic Programming ile olmak üzere iki yaklaşım gösterilmiştir.

**Giriş**

Projenin amacı, dizi elemanlarının küçükten büyüğe veya büyükten küçüğe sıralanması, bir string içerisinde istenen karakterin veya karakter grubunun(sözcüğün) bulunması ve birden fazla matrisin ard arda çarpılması problemlerinin çözümlerinde kullanılan bazı algoritmaların etkinlik (çalışma zamanı ve karşılaştırma sayısı) olarak incelenmesi ve karşılaştırılmasıdır. Ele alınan algoritmalar:

1. Kısımda dizi sıralaması için **Bubble Sort** ve **Quick Sort**,

2. Kısımda string eşleştirme için **Bruteforce**, **Horspool** ve **Boyer-Moore**,

3. Kısımda Matrix Chain Product için **In-order** ve **Dynamic Programming Matrix Chain** algoritmalarıdır.

**1. Dizi sıralama algoritmalarının karşılaştırılması:**

Verilen n elemanlı bir dizideki sayıların küçükten büyüğe sıralanması işlemidir. Dizi sıralama algoritması seçimi, küçük dizilerde önemsizmiş gibi görünse de dizi boyutları artmaya başladığında kaynak kullanımı açısından büyük farklılıklar ortaya çıkabilmektedir. Günümüzde yazılım çözümlerinin büyük çoğunluğunda sıralama, doğrudan veya çözümlerin bir parçası olarak kullanılmaktadır. Veri boyutunun da büyüklüğü göz önüne alındığında sıralama algoritmalarının etkinliği bütün yazılımın etkinliğiyle doğrudan alakalıdır.

**2. String eşleştirme algoritmalarının karşılaştırılması:**

Verilen n uzunluktaki bir string(text) içerisinde istenen string’in(pattern) aranması ve eğer var ise yerinin(başladığı yerin) belirtilmesi işlemidir. Günümüzde veri boyutunun çok fazla artması, faydalı ve faydasız verilerin birlikte bir veri kütlesi olarak bulunması, bu büyük veri topluluğu içerisinde arama yapılırken kullanılan algoritmanın etkinliğinin önemini kritik hale getirmiştir. Ayrıca bu algoritmalar Tıp ve Adli Tıp uygulamalarında DNA eşlemesi yapılırken de kullanılmaktadır.

**3. Matrix Chain Problemi:**

Verilen n adet matrisin çarpımlarının hesaplanmasının en etkin şekilde(en kısa sürede) yapılabilmesi için en az çarpma sayısı olacak şekilde parantezlere ayrılması problemidir. Kullanılan algoritmaların görevi, bu parantezlerin ayrım bölgelerinin belirlenmesi ve çarpımın gerçekleştirimidir. Matris çarpımı işlemi özellikle büyük matrixlerde çok fazla çarpma sayısı ile sonuçlandığından, işlem süresini bariz olarak etkileyen etkin bir algoritmanın kullanımı daha fazla çözümün daha kısa sürede yapılabilmesi demektir. Matris çarpımı matematik alanında, grafik işlemede (OpenGL standardı bir çok data için matris kullanmaktadır) kullanılmaktadır.

Bu projede her üç problem için, özdeş örnekler üzerinde farklı algoritmalarla yapılan deneylerle hangi durumlarda bahsedilen algoritmalardan hangilerinin daha etkin olduğu gösterilecektir. Raporun geri kalan kısmında kullanılan algoritmalar ve elde edilen sonuçlar üzerinde durulacaktır.

**Rapor Kısmı**

**Bölüm 1**

**Dizi Sıralama Algoritmaları BubbleSort ve QuickSort’un karşılaştırılması**

Problem: Verilen n elemanlı bir dizideki sayıların küçükten büyüğe sıralanması.

Problemin çözümünde aynı örnekler için BubbleSort ve QuickSort algoritmaları kullanılmıştır.

**BubbleSort Pseudocode’u:**

**ALGORITHM** *BubbleSort(A*[0*..n* − 1]*)*

//Sorts a given array by bubble sort

//Input: An array *A*[0..*n* − 1] of orderable elements

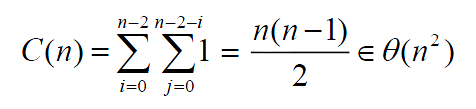
//Output: Array *A*[0..*n* − 1] sorted in nondecreasing order

**for** *i* ←0 **to** *n* − 2 **do**

**for** *j* ←0 **to** *n* − 2 − *i* **do**

**if** *A*[*j* + 1]*<A*[*j* ] swap *A*[*j* ] and *A*[*j* + 1]

**BubbleSort Teorik Analiz:** Basic operation olan karşılaştırma satırı iki for döngüsü içerisinde işletilmektedir.



Teorik olarak, BubbleSort fonksiyonu ile sıralandığında n elemanlı bir dizide **n(n-1)/2 karşılaştırma** beklenmektedir. Aynı zamanda karşılaştırma sayısı fonksiyonunun C(n) n^2 oranında büyümesi beklenmektedir.

**QuickSort Pseudocode’u:**

**ALGORITHM** *Quicksort(A*[*l..r*]*)*

//Input: Subarray of array *A*[0*..n* − 1]*,* defined by its left and right

// indices *l* and *r*

//Output: Subarray *A*[*l..r*] sorted in nondecreasing order

**if** *l < r*

*s* ←*Partition(A*[*l..r*]*)* //*s* is a split position

*Quicksort(A*[*l..s* − 1]*)*

*Quicksort(A*[*s* + 1*..r*]*)*

**ALGORITHM** *Partition(A*[*l..r*]*)*

//Partitions a subarray by Hoare’s algorithm, using the first element

// as a pivot

//Input: Subarray of array *A*[0*..n* − 1]*,* defined by its left and right

// indices *l* and *r (l<r)*

//Output: Partition of *A*[*l..r*], with the split position returned as

// this function’s value

*p*←*A*[*l*]

*i* ←*l*; *j* ←*r* + 1

**repeat**

**repeat** *i* ←*i* + 1 **until** *A*[*i*]≥ *p*

**repeat** *j* ←*j* − 1 **until** *A*[*j* ]≤ *p*

swap*(A*[*i*]*, A*[*j* ]*)*

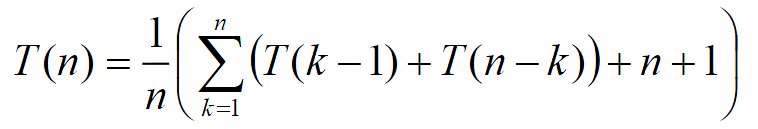
**until** *i* ≥ *j*

swap*(A*[*i*]*, A*[*j* ]*)* //undo last swap when *i* ≥ *j*

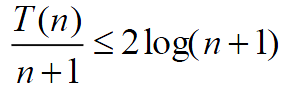
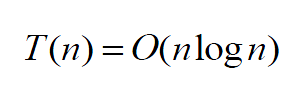
swap*(A*[*l*]*, A*[*j* ]*)*

**return** *j*

**QuickSort Teorik Analiz:** Divide and Conquer tekniğini kullanan bu sıralama algoritmasının average case zaman karmaşıklığı



bağıntısından

Olarak elde edilmektedir. T(n) değerlerinin nlogn’den daha az bir büyüklükle orantılı olarak büyüyeceği beklenmektedir.

**Bölüm 1 Gerçekleştirim:**

Java dilinde ve Eclipse IDE kullanılarak yapılmıştır. Deney sistemi olarak Intel Core Duo T9600 2.80 GHZ, 4 GB RAM, Win7 64bit işletim sistemi kullanılmıştır.

***Part1.Java*** isimli implementasyonumuzda önce kullanıcıdan dizi büyüklüğü alınmaktadır. BubbleSort ve QuickSort fonksiyonları sırayla çağrılarak, her ikisi için de ayrı ayrı timer başlatılıp sonlandırılarak süreleri milisaniye cinsinden tutularak hesaplanmaktadır. Basic operation sayıları ise operasyon sayısı için counter’ı arttırma komutları kullanılarak hesaplanmıştır.

1000’den 20.000’e kadar 1000’er arttırılarak 20 ayrı dizi büyüklüğü için BubbleSort’a ilişkin teorik sonuçlar hesaplanmıştır ve deneysel sonuçlar programa hesaplattırılmıştır.

**BubbleSort Teorik ve Deney Sonuçları:**

Size(n) için ortalama karşılaştırma sayıları:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **size n** | **1000** | **2000** | **3000** | **4000** | **5000** | **6000** |
| **KeyCompBubbleSort** | **499500** | **1999000** | **4498500** | **7998000** | **12497500** | **17997000** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **size n** | **7000** | **8000** | **9000** | **10000** | **11000** | **12000** |
| **KeyCompBubbleSort** | **24496500** | **31996000** | **40495500** | **49995000** | **60494500** | **71994000** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **size n** | **13000** | **14000** | **15000** | **16000** | **17000** | **18000** |
| **KeyCompBubbleSort** | **84493500** | **97993000** | **112492500** | **127992000** | **144491500** | **161991000** |

|  |  |  |
| --- | --- | --- |
| **size n** | **19000** | **20000** |
| **KeyCompBubbleSort** | **180490500** | **199990000** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Size n** | **1000** | **2000** | **3000** | **4000** | **5000** | **6000** |
| **BubbleS Avg. runtime ms** | **11** | **13** | **21** | **30** | **40** | **57** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **7000** | **8000** | **9000** | **10000** | **11000** | **12000** | **13000** | **14000** |
| **79** | **101** | **123** | **151** | **182** | **217** | **253** | **295** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **15000** | **16000** | **17000** | **18000** | **19000** | **20000** |
| **331** | **382** | **430** | **482** | **563** | **597** |

BubbleSort algoritması, karşılaştırma sayısı olarak beklenen değerlere eşit çıkmıştır.

**QuickSort Teorik Sonuçları:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **size n** | **1000** | **2000** | **3000** | **4000** | **5000** | **6000** | **7000** |
| **Teo. Key QuickSort** | **9965,8** | **21,93155** | **34652,25** | **47863,15** | **61438,55** | **75304,5** | **89411,95** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **size n** | **8000** | **9000** | **10000** | **11000** | **12000** | **13000** | **14000** |
| **Teo. Key QuickSort** | **103726** | **118221,4** | **132877,1** | **147677,4** | **162609** | **177660,9** | **192823,9** |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **size n** | **15000** | **16000** | **17000** | **18000** | **19000** | **20000** |
| **Teo. Key QuickSort** | **208090** | **223452,5** | **238905,2** | **254442,8** | **270060,5** | **285754,2** |

**QuickSort Deney Sonuçları:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **size n** | **1000** | **2000** | **3000** | **4000** | **5000** | **6000** | **7000** |
| **Avg Key QuickSort** | **10983** | **23934** | **36702** | **50430** | **64266** | **79218** | **93342** |
| **Avg (ms) QuickSort** | **0** | **1** | **4** | **5** | **4** | **4** | **4** |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **size n** | **8000** | **9000** | **10000** | **11000** | **12000** | **13000** | **14000** |
| **Avg Key QuickSort** | **107235** | **122226** | **137124** | **150000** | **166128** | **179892** | **194106** |
| **Avg ms QuickSort** | **5** | **4** | **4** | **4** | **5** | **5** | **5** |

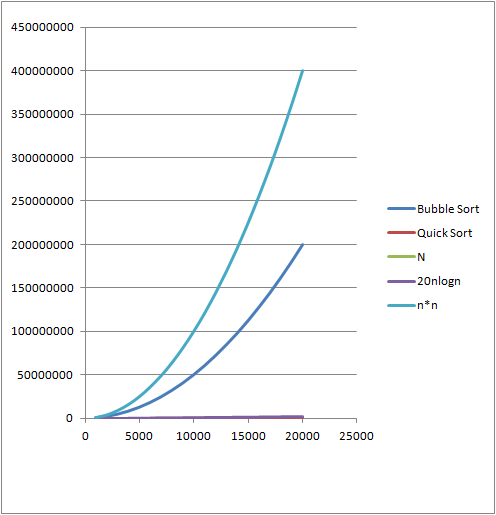
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **size n** | **15000** | **16000** | **17000** | **18000** | **19000** | **20000** |
| **Avg Key QuickSort** | **211962** | **226044** | **241575** | **256638** | **271494** | **285675** |
| **Avg Run QuickSort** | **5** | **5** | **5** | **5** | **6** | **6** |

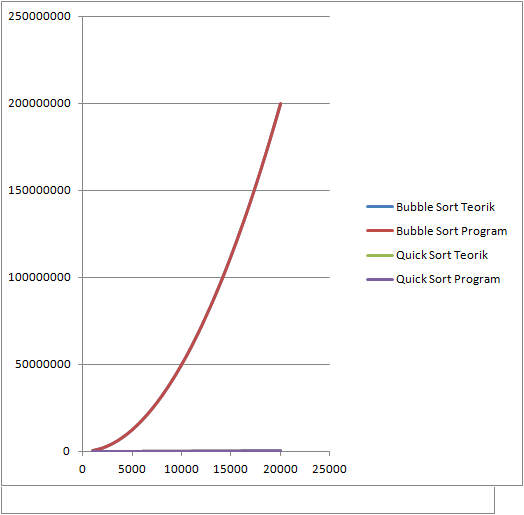
Quicksort deney sonuçları, teorik sonuçlara göre üst sınırdan biraz fazla olmakla beraber, sonuçların aynı oranda artan yakın değerler olduğu görülmüştür. (O(nlogn))

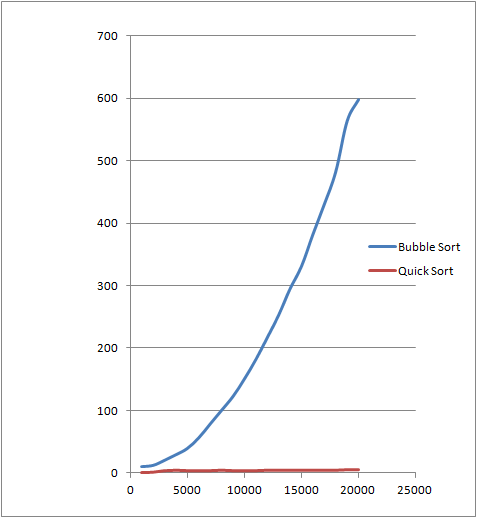
Divide and Conquer tekniğini kullanan QuickSort algoritmasının, Bruteforce-Exhaustive Search tekniğiyle arama yapan BubbleSort algoritmasına göre çok daha fazla etkin olduğu görülmüştür.

Aşağıdaki grafiklerde görüldüğü gibi BubbleSort n\*n den küçük ama ona yakın bir karmaşıklığa sahiptir.

**Plot Grafikler**







**Bölüm 2**

**String Eşleştirme Algoritmalarının Karşılaştırılması**

**(BruteForce, Horspool, BoyerMoore)**

**Problem**: Verilen n uzunluktaki bir String içerisinde aranan karakter veya karakter topluluğunun bulunması. (Başlangıç index’inin döndürülmesi)

Problemin çözümünde aynı örnekler için Bruteforce, Horspool ve BoyerMoore string matching algoritmaları kullanılmıştır.

**2. Kısım Algoritmaları:**

**BruteForce String Match Pseudocode’u:**

**ALGORITHM** *BruteForceStringMatch(T* [0*..n* − 1]*, P*[0*..m* − 1]*)*

//Implements brute-force string matching

//Input: An array *T* [0*..n* − 1] of *n* characters representing a text and

// an array *P*[0*..m* − 1] of *m* characters representing a pattern

//Output: The index of the first character in the text that starts a

// matching substring or −1 if the search is unsuccessful

**for** *i* ←0 **to** *n* − *m* **do**

*j* ←0

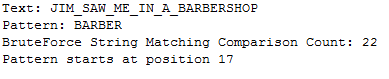
**while** *j <m* **and** *P*[*j* ]= *T* [*i* + *j* ] **do**

*j* ←*j* + 1

**if** *j* = *m* **return** *i*

**return** −1

**BruteForceStringMatch Teorik Analiz:** En kötü durumda, m uzunluğundaki bir dizi için m(n-m+1) karşılaştırma yapılması gerekmektedir. Bu da O(nm) sınıfında olduğu anlamına gelmektedir.



Görüldüğü üzere text imiz 26 karakter uzunluğundadır. İçerisinde 6 karakter uzunluğundaki BARBER sözcüğünü brute force yöntemiyle ararken karşılaştırma sayımız 22 ve sözcüğümüz bulunduğu için başladığı konum olan 17 out put olarak gösterilmiştir.

**Horspool Pseudocode’u:**

**ALGORITHM** *HorspoolMatching(P* [0*..m* − 1]*, T* [0*..n* − 1]*)*

//Implements Horspool’s algorithm for string matching

//Input: Pattern *P*[0*..m* − 1] and text *T* [0*..n* − 1]

//Output: The index of the left end of the first matching substring

// or −1 if there are no matches

*ShiftTable(P* [0*..m* − 1]*)* //generate *Table* of shifts

*i* ←*m* − 1 //position of the pattern’s right end

**while** *i* ≤ *n* − 1 **do**

*k*←0 //number of matched characters

**while** *k* ≤ *m* − 1 **and** *P*[*m* − 1− *k*]= *T* [*i* − *k*] **do**

*k*←*k* + 1

**if** *k* = *m*

**return** *i* − *m* + 1

**else** *i* ←*i* + *Table*[*T* [*i*]]

**return** −1

**ALGORITHM** *ShiftTable(P* [0*..m* − 1]*)*

//Fills the shift table used by Horspool’s and Boyer-Moore algorithms

//Input: Pattern *P*[0*..m* − 1] and an alphabet of possible characters

//Output: *Table*[0*..size* − 1] indexed by the alphabet’s characters and

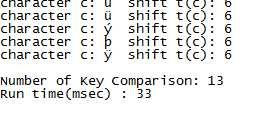
// filled with shift sizes computed by formula (7.1)

**for** *i* ←0 **to** *size* − 1 **do** *Table*[*i*]←*m*

**for** *j* ←0 **to** *m* − 2 **do** *Table*[*P*[*j* ]]←*m* − 1− *j*

**return** *Table*

**Horspool Teorik Analiz:** Aranan pattern uzunluğu M, içinde arama yapılan string uzunluğu N kabul edilirse, random text üzerinde O(N) average case, O(MN) worst case zaman karmaşıklığına sahiptir.



Burada brute force da olduğu gibi aynı text ve pattern kullanılmıştır. Shift table bütün ASCII karakterleri için oluşturulmuş olup 256 karakter içermektedir. Aradığımız pattern i bulmak için burada 13 karşılaştırma yapmışız ve programımızın run time ı da 33 msec sürdü.

**Boyer-Moore Pseudocode’u:**

**Algoritm** BMMatch(T, P):

*//last*(ch) is the index of the last occurence of ch in P, if ch is in P, and otherwise, //*last*(ch) =-1

//match(j) is the smallest s >= 1 such that one of the following is satisfied:

//P[j+1-s...m-s-1] is a suffix of P[j+1...m-1] (or j=m-1) andP[j] <> P(j-s], with //s<= j (so all previously matched text characters will match the shifted pattern //and there is a chance that P[j-s] may match in the text the character that mis //mismatched with P[j]).

//P[0...m-s-1] is a suffix of P[j+1...m-1], with j<s<m ( so as many of the //previously matched text characters will mact a prefix of P as possible, //because we are shifting P past the character that mismatched with P[j]).

//s = m (in which case we would shift the pattern completly past all the characters of the text were previously compared with P[j...m-1]).

**Input**: String T (text with n characters and P (pattern) with m characters **Output**: Starting index offirst substring of T matching p, or an indication that P is not a substring of T

i <-- m - 1  
j <-- m - 1

**repeat**

**if** P[j] = T[i] **then**

**if** j = 0 **then**

**return** i {a match!}

**else** {check next character}

i <-- i - 1  
j <-- j - 1

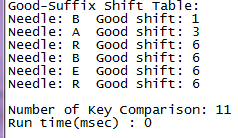
**else** { P[j] <> T[i] move the pattern}

i <-- i + m - j - 1 {reset i to where it began in most-recent test}  
i <-- i + max(j - last(T[i]), match(j)) {shift P relative to T}  
{note that even if j-last(T[i]) is negative, we will still perform apositive shift, because match (j) is always at least 1.}  
j <-- m-1

**until** i > n - 1

**return** -1

**Boyer-Moore Teorik Analiz:** Aranan pattern uzunluğu M, içinde arama yapılan string uzunluğu N kabul edilirse O(N+M) zaman karmaşıklığına sahiptir.



Yukarıda programın çalışma esnasında alınmış bir ekran görüntüsü görünmektedir. Burada boyer-moore algoritmasına göre oluşturmuş olduğumuz good-suffix table ımızı görüyoruz. Ayrıca run time ve programın sonuca ulaşması için gereken karşılaştırma sayıları da ekran görüntüsünde yer alıyor.

**Bölüm 2.nin Gerçekleştirimi:**

Java dilinde ve Eclipse IDE kullanılarak yapılmıştır. Deney sistemi olarak Intel Core Duo T9600 2.80 GHZ, 4 GB RAM, Win7 64bit işletim sistemi kullanılmıştır.

part2\_infile.txt dosyası girdi olarak kullanılmaktadır. İlk satır içinde arama yapılacak olan text’i, ikinci satır aranacak olan pattern’ı içerir. Örnek part2\_infile.txt:

ARAMA

MA

***Part2.Java*** isimli class’ta, aranacak pattern ve arama yapılacak string alındıktan sonra önce Bruteforce sonra Horspool algoritmaları kullanılarak, her biri için arama süreleri ve karşılaştırma sayıları tutulup yazdırılmaktadır. Ayrıca Horspool algoritmasında shift table yazdırılmaktadır.

Bruteforce string arama fonksiyonu, aranan pattern’ın ilk harfini string içerisine aramakta, eğer bulursa kelimenin geri kalanını bu harfin devamında aramaktadır.

Horspool string arama fonksiyonu için önce shift table oluşturulmaktadır. Shift table için her karakter için soldan sağa doğru arama yapılırken kaç karakter kaydırma yapılacağı bilgisi hesaplanmaktadır. Shift table oluşturulduktan sonra arama yapılırken pattern’ın son karakteriyle, sıradaki karakter karşılaştırılıyor. Eğer doğruysa pattern’ın harfleri sola doğru aranmaya başlıyor. Eğer yanlış ise en sağda karşılaştırma yapılan karakterin Shift değeri kadar, pattern shift ediliyor.

**BoyerMoore.Java** class’ı, **IstringSearch** interface’ini implement ediyor.

BoyerMoore algoritması, bad-symbol table ve good-suffix table olmak üzere iki tablo oluşturuyor.

Bad-symbol table, Horspool algoritmasındaki Shift table ile aynı mantıkla oluşturulmaktadır.

Good suffix table ise, pattern’ın son karakteri, son iki karakteri, son... karakteri şeklinde giderek, pattern üzerinde sağdan sola doğru giderken kaç kaydırma yapılacağı bilgisi ile oluşturulmaktadır.

Pattern’ın en sağındaki karakter, text’in sıradaki karakteriyle eşleşiyorsa pattern sola doğru doğru kelime mi diye taranır.

Eğer doğru kelime değilse, pattern’ın sağa doğru kaydırma miktarı bulunmalıdır. Eşleşmeyen ilk karakterin Bad Shift değerinden bu karakterin sağa olan uzaklığı(pattern bitimine olan uzaklığı) çıkarılır. Bu değerin ve eşleşen kısmın maximumu, bir sonraki kaydırma miktarıdır.

Buna göre arama yapıldıktan sonra her 3 algoritma için işlem süreleri, karşılaştırma sayıları ve Horspool Shift Table’ı ile Boyer-Moore Bad Shift table’ı ve Good Suffix Table’ı part2\_outfile.txt dosyasına yazdırılıyor.

**Bölüm 3**

**Matrix Chain Product Algoritmalarının Karşılaştırılması**

**(Sıra ile çarpma ve Dinamik programlama ile çarpma)**

**Problem:** Verilen n adet matrisin çarpımlarının hesaplanmasının en etkin şekilde(en kısa sürede) yapılabilmesi için en az çarpma sayısı olacak şekilde parantezlere ayrılması.

Problemin çözümünde aynı örnek için Bruteforce(in-order) ve Dynamic programming yaklaşımları uygulanmıştır. In-order multiplying kısmında sadece çarpma işlemi yapılırken, Dynamic programming yaklaşımında tablo oluşturma, en uygun şekilde parantezlere ayırma ve çarpma işlemleri gerçekleşmektedir.

**3.Kısım Algoritmaları:**

**In-order Multiplying Pseudocode’u:**

**ALGORITHM InorderMultiply(A,B)**

if columns[A] <> rows [B]

then ERROR "Incompatible Dimensions"

else for i = 1 to rows[A]

do for j = 1 to columns[B]

do C[i,j] = 0

for k = 1 to columns[B]

do C[i,j] = C[i,j] + A[i,k] \* B[k,j]

return C

**In-order Multiplying Teorik Analiz:** T(n) = 8T(n/2) + O(n^2 ) = O (n3)

**Dynamic Matrix Chain Product Pseudocode’u:**

PRINT OPTIMAL PARENS(i, j)

if i = j

then print(’Ai’)

return

print(’(’)

k s[i, j]

PRINT OPTIMAL PARENS(i, k)

PRINT OPTIMAL PARENS(k + 1, j)

print(’)’)

return

**Dynamic Matrix Chain Product Teorik Analiz:**

T(n) = T(s[1, n]) + T(n − s[1, n]) + 2

= 3s[1, n] − 2 + 3(n − s[1, n]) − 2 + 2

= 3n – 2

T(n)=O(n)

T(n) lineer zamanda çalışmaktadır.

**3.Bölüm Gerçekleştirimi:**

MatrixChain.Java dosyasında yapılmıştır. Kullanıcıdan alınan matris sayısına göre, random matrisler oluşturulup, önce bruteforce yöntemle, sonra matrix chain dinamik programlama yöntemiyle çarpılmıştır. Ayrı ayrı zaman sayaçları ve çarpma sayıları hesaplanmıştır.

**Matris Çarpımı Deney Sonuçları:** (Dynamic programming kısmına tabloları oluşturma ve dosyaya yazdırma süreleri de dahildir. Bu yüzden in-order çarpma 0 ms’de gerçekleşirken dinamik programlama ile çarpma

girdi büyüklüğüne göre yaklaşık 5-10 ms’lik gecikmelerle sonuçlanmaktadır)

input: 15 matris

Matrix Sizes: A1 3 x 2 ; A2 2 x 2 ; A3 2 x 12 ; A4 12 x 9 ; A5 9 x 7 ; A6 7 x 2 ; A7 2 x 8 ; A8 8 x 1 ; A9 1 x 12 ; A10 12 x 7 ; A11 7 x 14 ; A12 14 x 15 ; A13 15 x 8 ; A14 8 x 8 ; A15 8 x 1

In-order multiplication

Number of multiplication: 2499

Run time(msec) : 0

Multiplication with Dynamic Programming

Table for Dynamic Programming

0 12 84 282 396 390 438 235 271 340 459 672 771 835 820

0 0 48 252 370 378 410 229 253 327 439 651 757 821 814

0 0 0 216 342 370 402 225 249 323 435 647 753 817 810

0 0 0 0 756 342 534 201 345 369 551 773 809 873 786

0 0 0 0 0 126 270 93 201 240 401 620 677 741 678

0 0 0 0 0 0 112 30 114 163 310 527 598 662 615

0 0 0 0 0 0 0 16 40 114 226 438 544 608 601

0 0 0 0 0 0 0 0 96 140 294 512 576 640 591

0 0 0 0 0 0 0 0 0 84 182 392 512 576 583

0 0 0 0 0 0 0 0 0 0 1176 2730 2982 3430 576

0 0 0 0 0 0 0 0 0 0 0 1470 2310 2758 492

0 0 0 0 0 0 0 0 0 0 0 0 1680 2576 394

0 0 0 0 0 0 0 0 0 0 0 0 0 960 184

0 0 0 0 0 0 0 0 0 0 0 0 0 0 64

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 1 1 1 0 5 0 7 7 7 7 7 7 0

0 0 1 1 1 1 5 1 7 7 7 7 7 7 1

0 0 0 2 3 4 5 2 7 7 7 7 7 7 2

0 0 0 0 3 3 5 3 7 7 7 7 7 7 3

0 0 0 0 0 4 5 4 7 7 7 7 7 7 4

0 0 0 0 0 0 5 5 7 7 7 7 7 7 5

0 0 0 0 0 0 0 6 7 7 7 7 7 7 7

0 0 0 0 0 0 0 0 7 7 7 7 7 7 7

0 0 0 0 0 0 0 0 0 8 9 10 11 12 9

0 0 0 0 0 0 0 0 0 0 9 9 9 9 9

0 0 0 0 0 0 0 0 0 0 0 10 11 12 10

0 0 0 0 0 0 0 0 0 0 0 0 11 12 11

0 0 0 0 0 0 0 0 0 0 0 0 0 12 12

0 0 0 0 0 0 0 0 0 0 0 0 0 0 13

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

**Matrices with paranthesis**: (p[0](p[1](p[2](p[3](p[4](p[5]((p[6]p[7])((p[8]p[9])(p[10](p[11](p[12](p[13]p[14]))))))))))))

**Number of multiplication:** 820

**Run time(msec)** : 0

---------------------------

input: 250 matris

In-order multiplication

Number of multiplication: 301370

Run time(msec) : 0

Number of multiplication: 30193

Run time(msec) : 31 (Tablo oluşturma ve dosya yazma işlemleri)

**Sonuçlar**

BubbleSort ve QuickSort karşılaştırmasında 1000’e kadar olan büyüklüklerde dahi 7ms’lik farklar olmuştur. Zaman karmaşıklığının önemli olduğu herhangi bir pratik uygulamada bu iki algoritma arasından kesinlikle divide and conquer bir algoritma olan QuickSort’un seçilmesi gerektiği görülmektedir.

Metin içerisinde herhangi bir pattern ararken hangi algoritmanın daha iyi olduğunu incelemeye çalıştık. Tabiki brute force aralarında en etkin olmayandı çünkü tek tek karşılaştırma işlemi yapıyordu. Diğer iki algoritmamızdan ise Boyer-Moore un Horspool'a göre daha etkin olduğunu söyleyebiliriz.

Matris çarpımları öncesi random sayılarla matris oluşturumu fazla süreyi alan işlemdir. (15.000 matris için ~5 saniye)

Random oluşturulan 15.000 matrisin çarpım sayısının hesaplanması 0 ms sürede gerçekleşmiştir. (20832 çarpma)

Random oluşturulan 15.000 matris için MatrixChainProduct dinamik programlama algoritması kullanarak optimal parantez ayrımının yapılması 10344 çarpma işlemi ile 5 milisaniyede gerçekleşmektedir.

Bu projede bir çok şey öğrendik.3 aşamada önce sıralama algoritmalarını uygulayarak aralarında farkı gözle gördük. 2. bölümde pattern bulmak için hangi algoritmaları kullanabileceğimizi öğrendik. Son bölümde ise matrisleri doğru sırayla çarpmanın ne kadar önemli olduğunu gördük.

**Kaynakça**

[**http://www.albany.edu/~csi503/pdfs/handout\_13.1.pdf**](http://www.albany.edu/~csi503/pdfs/handout_13.1.pdf)

[**http://www.schweriner.com/cse465.html**](http://www.schweriner.com/cse465.html)

[**http://www.dei.unipd.it/~geppo/DA2/DOCS/dynprog.pdf**](http://www.dei.unipd.it/~geppo/DA2/DOCS/dynprog.pdf)

http://bhaus.bh.funpic.de/UNI/Info2/Blatt4/

**http://bitstogather.blogspot.com/2011/04/brute-force-pattern-matching.html**

**http://www.vogella.com/articles/JavaAlgorithmsQuicksort/article.html**

[**http://orion.lcg.ufrj.br/Dr.Dobbs/books/book5/chap10.htm**](http://orion.lcg.ufrj.br/Dr.Dobbs/books/book5/chap10.htm)

[**http://en.wikipedia.org**](http://en.wikipedia.org)

**http://www.people.westminstercollege.edu/faculty/ggagne/fall2012/306/labs/lab8/index.html**

**Introduction to the Design and Analysis of Algorithms 3rd Edition, Levitin, Anany**

[**http://efe.ege.edu.tr/~bulut/courses/451/451.html**](http://efe.ege.edu.tr/~bulut/courses/451/451.html)

**http://www.codeforge.com/read/29266/Matrix\_Chain.java\_\_html**