



T.C. Fırat Üniversitesi
Bilgisayar Mühendisliği Bölümü

BMÜ316-Algoritma Analizi Dersi
Zaman Karmaşıklığı Analizi Ödevi Raporu

Ders Sorumlusu: Prof. Dr. Mehmet KARAKÖSE

Öğrenci İsim: Mert İNCİDELEN
Öğrenci No.: 170260101

Ödev Tanımı

Ödevde en az 8 farklı algoritma tasarım stratejisine göre yazılmış 8 ayrı algoritmanın sözde kodu ile birlikte algoritmaların karmaşıklıklarının saptanması ve derleyici üzerinde çalıştırılarak en iyi, ortalama ve en kötü senaryolar için geçen sürelerin analizinin yapılması beklenmektedir. Raporda farklı algoritma tasarım stratejileri için farklı algoritmalar örneklendirilmiş, zaman karmaşıklıkları belirtilerek Java dilinde derleyici üzerinde çalıştırarak zaman tüketimleri analiz edilmiştir. Derleyici üzerinde analizi yapılan kodlar raporun sonundaki ekte sunulmuştur.

1. Böl ve Yönet (Divide and Conquer) Algoritma Stratejisi

Böl ve yönet stratejisi, problemi daha küçük parçalara bölerek hızlı bir şekilde çözüme ulaşabilmeyi sağlar. Bu strateji için aşağıda ikili arama algoritması örneklenmiştir.

```
ikiliArama(dizi[], bas, son, aranan)
    while bas ≤ son do
        orta = (bas + son) / 2
        if dizi[orta] == aranan then
            return orta
        else if dizi[orta] < aranan then
            bas = orta + 1
        else
            son = orta - 1
        end if
    end while
```

Zaman Karmaşıklığı: $O(\log n)$

İkili arama algoritmasında, döngünün her bir adımında sıralı dizinin ortasındaki değeri kontrol ederek arama işlemini gerçekleştirmekte ve her adımdan sonra odaklanılması gereken veri sayısı yarıya inmektedir. Dolayısıyla n elemanlı sıralı dizi için zaman karmaşıklığının $O(\log n)$ olması beklenir. Algoritma Java dilinde bilgisayar üzerinde yürütülerek en iyi, ortalama ve en kötü durumda zaman tüketimi analiz edilmiştir. Boyutu 200000000 olan diziye for döngüsüyle 0-199999999 arası sayılar sıralı bir şekilde yerleştirilmiştir.

En İyi Durum: $O(1)$

Aranan verinin, veri kümesinin ortasındaki veri olması durumu ikili arama algoritması için en iyi durum senaryosudur. 0-199999999 arası sıralı sayıların bulunduğu dizinin ortasındaki 99999999 değerinin aranması ile gerçekleşen zaman tüketiminin derleyici üzerindeki görünümü aşağıdaki gibidir.

```
Output - A_IkiliArama (run) X
run:
Veri sayısı: 200000000
Aranan: 99999999 | Şu sırada bulundu: 99999999
Geçen Süre: 6800 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ortalama Durum: $O(\log n)$

Aranan verinin, veri kümesinin herhangi bir yerinde bulunması durumudur. 0-1999999999 arası sıralı sayıların bulunduğu dizide 786541 değerinin aranması ile gerçekleşen zaman tüketiminin derleyici üzerindeki görünümü aşağıdaki gibidir.

```
Output - A_IkiliArama (run) X
run:
Veri sayısı: 200000000
Aranan: 786541 | Şu sırada bulundu: 786541
Geçen Süre: 12300 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

En Kötü Durum: $O(\log n)$

Aranan verinin dizin başında ve sonunda bulunması durumu en kötü durum senaryolarıdır. 0-1999999999 arası sıralı sayıların bulunduğu dizide 0 değerinin aranması ile gerçekleşen zaman tüketiminin derleyici üzerindeki görünümü aşağıdaki gibidir.

```
Output - A_IkiliArama (run) X
run:
Veri sayısı: 200000000
Aranan: 0 | Şu sırada bulundu: 0
Geçen Süre: 18300 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

Zaman tüketimi analizinde görülebileceği üzere büyük sıralı veri kümeleri için arama zaman tüketimi minimum düzeyde kalmaktadır. 200000000 veri için en kötü durumda arama işlemi 28 adımda tamamlanmaktadır.

2. Kaba Kuvvet (Brute Force) Algoritma Stratejisi

Kaba kuvvet yaklaşımıyla tasarlanan algoritmalar çözüme ulaşmak için tek tek her yolu deneyerek işlemleri gerçekleştirirler. Bu strateji için aşağıda kabarcık sıralama algoritması örneklenmiştir. Algoritma, sırasız bir dizinin sıralanabilmesi için diziyi baştan sona dizi boyutu kadar tarayarak sıralama işlemini gerçekleştirir.

```
kabarcikSiralama(dizi[], diziUzunluk)
  for i = 0, 1, 2, ... , diziUzunluk-1 do
    for j = 0, 1, 2, ... , diziUzunluk-1 do
      if dizi[j] > dizi[j+1] then
        temp=dizi[j]
        dizi[j] = dizi[j+1]
        dizi[j+1] = temp
      end if
    end for
  end for
```

Zaman Karmaşıklığı: $O(n^2)$

Kabarcık sıralama algoritmasında dizi boyutu n kadar, n kez tarama işlemi gerçekleştirildiğinden zaman karmaşıklığı $O(n^2)$ 'dir. Veri sayısı arttıkça zaman tüketimi de karesel olarak artacaktır.

Algoritma Java dilinde bilgisayar üzerinde yürütülerek en iyi, ortalama ve en kötü durumda zaman tüketimi analiz edilmiştir. Veri sayısı arttıkça zaman tüketimi görülür biçimde karesel olarak artmaktadır. Bu nedenle 100000 adet elemanlı dizi üzerinde zaman tüketim analizi yapılmıştır. Boyutu 100000 olan diziye for döngüsüyle 0-99999 arası sayılar sıralı ve tersten sıralı bir şekilde yerleştirilmiştir. Ortalama durum analizi için ise sayılar dizinin bir başına ve bir sonuna eklenerek dağınık bir şekilde yerleştirilmiştir.

En İyi Durum: $O(n)$

Dizinin sıralı bir şekilde bulunması durumu en iyi durum senaryosudur. Bu durumda dizi bir kez n defa taranacaktır. 0-99999 arası sayıların sıralı bulunduğu dizi için zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - A_Kabarcik (run) X
run:
Dizi: 0 1 2 3 ... 99998 99999
100000 adet veri sıralı yerleştirildi
Sıralandı: 0 1 2 3 ... 99998 99999
Geçen süre: 4464694000 nanosaniye
BUILD SUCCESSFUL (total time: 4 seconds)
```

Ortalama Durum: $O(n^2)$

Dizide verilerin dağınık bir biçimde bulunma durumu ortalama durum senaryosudur. 0-99999 arası sayılar sırasıyla dizinin başına ve sonuna eklenerek elde edilen dağınık dizinin sıralanması işleminin zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - A_Kabarcik (run) X
run:
Dizi: 0 2 4 6 ... 3 1
100000 adet veri dağınık yerleştirildi
Sıralandı: 0 1 2 3 ... 99998 99999
Geçen süre: 21836471000 nanosaniye
BUILD SUCCESSFUL (total time: 22 seconds)
```

En Kötü Durum: $O(n^2)$

Dizide verilerin tersten sıralı bir biçimde bulunma durumu en kötü durum senaryosudur. 0-99999 arası sayıların tersten sıralı bir şekilde yerleştirilerek elde edilen sırasız dizinin zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - A_Kabarcik (run) X
run:
100000 adet veri tertan yerleştirildi
Sıralandı: 0 1 2 3 ... 99998 99999
Geçen süre: 24288140700 nanosaniye
BUILD SUCCESSFUL (total time: 24 seconds)
```

3. Rekürsif Algoritma Stratejisi

Rekürsif algoritmalar, kendini tekrarlayarak işlemleri gerçekleştirip çözüme kavuşturan algoritmalarlardır. Rekürsif algoritma örneği için aşağıda n. sıradaki Fibonacci Sayısını hesaplayan rekürsif algoritmanın sözde kodu yer almaktadır.

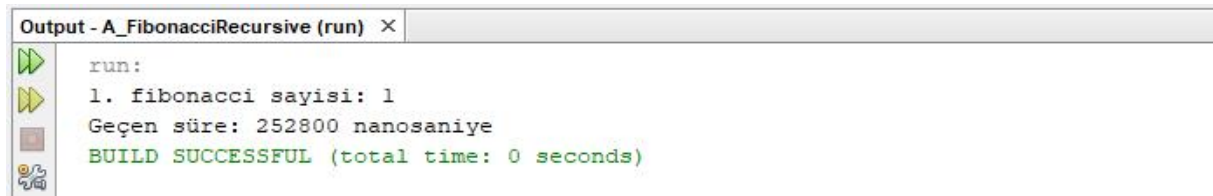
```
fibonacci(n)
    if n == 0 or n == 1 then
        return n
    else
        return fibonacci(n-1) + fibonacci(n-2)
    end if
```

Zaman Karmaşıklığı: $O(2^n)$

Bu algoritmada n. sayının hesaplanabilmesi için n-1. ve n-2. sayıların hesaplanması ve bu her iki sayı için de iki yeni hesaplamanın yapılarak bu şekilde hesaplamaların devam etmesi dolayısıyla işlemlerin 2^n defa tekrarlanması gerekmektedir. Buna göre bu algoritmanın zaman karmaşıklığı $O(2^n)$ 'dir. n değeri arttıkça hesaplama süresi 2 tabanında üssel artmaktadır.

En İyi Durum: $O(1)$

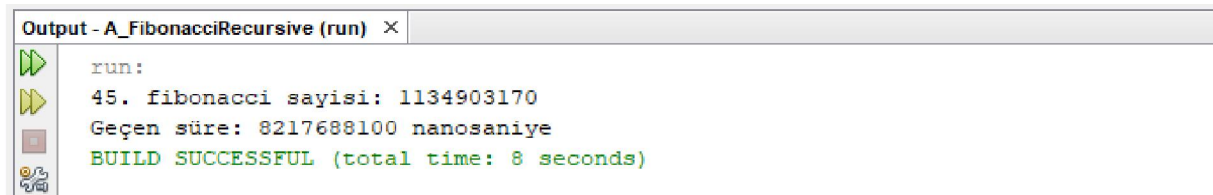
1. sıradaki fibonacci sayısının hesaplanması işlemi en iyi durum senaryosudur. 1. Sıradaki fibonacci sayısının hesaplanması işleminin zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.



```
Output - A_FibonacciRecursive (run) X
run:
1. fibonacci sayisi: 1
Geçen süre: 252800 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ortalama Durum: $O(2^n)$

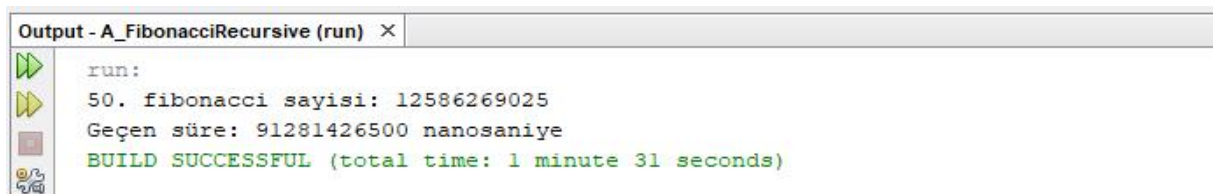
n değeri arttıkça hesaplama işlemi için gereken süre artmaktadır. n değeri 45 seçilerek zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.



```
Output - A_FibonacciRecursive (run) X
run:
45. fibonacci sayisi: 1134903170
Geçen süre: 8217688100 nanosaniye
BUILD SUCCESSFUL (total time: 8 seconds)
```

En Kötü Durum: $O(2^n)$

Aşağıda yer alan derleyici üzerindeki görüntüde n değeri önceki değere göre 5 arttırılmış, buna rağmen zaman tüketiminde büyük artış gözlemlenmiştir.



```
Output - A_FibonacciRecursive (run) X
run:
50. fibonacci sayisi: 12586269025
Geçen süre: 91281426500 nanosaniye
BUILD SUCCESSFUL (total time: 1 minute 31 seconds)
```

Anlaşılabacağı üzere yüksek n değerlerine karşılık fibonacci sayılarının hesaplanabilmesi için çok uzun zamana ihtiyaç duyulacaktır. Bir sonraki başlıkta fibonacci sayılarının hesaplanması işlemi dinamik programlama stratejisi ile incelenmiştir.

4. Dinamik Programlama Stratejisi

Dinamik programlama stratejisinde, problemin alt problemlere bölünmesi ve bu alt problemlerin bir kez çözülerek çözümün sonraki tekrarda kullanılması yaklaşımı görülür. Aşağıda dinamik programlama stratejisi ile n . sıradaki Fibonacci Sayısını hesaplayan algoritmanın sözde kodu yer almaktadır.

```
fibonacci(n)
    new fib[]
    fib[0] = 1
    fib[1] = 1
    for i = 2, 3, 4, ... , n do
        fib[i] = fib[i - 1] + fib[i - 2]
    end for
    return fib[n-1]
```

Zaman Karmaşıklığı: $O(n)$

Bu algoritmada n . fibonacci sayısının hesaplanabilmesi için 1'den n 'e kadar fibonacci değerleri hesaplanıp kaydedilecek, dolayısıyla n kez bu işlemler tekrarlanacaktır. Buna göre bu algoritmanın zaman karmaşıklığı $O(n)$ 'dir. n değeri arttıkça hesaplama için gereken zaman doğrusal olarak artacaktır. Algoritma Java dilinde bilgisayar üzerinde yürütülerek en iyi, ortalama ve en kötü durum senaryoları için n değeri seçilerek zaman tüketimi analiz edilmiştir.

En İyi Durum: $O(1)$

1. sıradaki fibonacci sayısının hesaplanması işlemi en iyi durum senaryosudur. Bu işlemin zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - A_FibonacciDinamik (run) X
run:
1. fibonacci sayısı: 1
Geçen süre: 237900 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ortalama Durum: $O(n)$

Bir önceki rekürsif fibonacci algoritmasında $n=45$ değeri için gereken zaman tüketimi bu algoritma için derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - A_FibonacciDinamik (run) X
run:
45. fibonacci sayısı: 1134903170
Geçen süre: 255000 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

En Kötü Durum: $O(n)$

n değeri arttıkça zaman tüketimi doğrusal olarak artacaktır. Burada n değerinin rekürsif algortmada örneklenen değerlerden oldukça yüksek olmasına rağmen kısa sürede hesaplama işlemi sonuçlanmıştır.

```
Output - A_FibonacciDinamik (run) X
run:
5000. fibonacci sayısı: 535601498209671957
Geçen süre: 407200 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

5. İteratif Algoritma Stratejisi

İteratif algoritma stratejisi, bir problemi çözüme kavuşturmak için gerekli adımları ayrı ayrı gerçekleştirmek yerine tekrarlayan işlemleri bir döngü ile adım adım çözümlendirme yaklaşımıdır. Aşağıda n sayısının faktöriyel hesabını gerçekleştiren sözde kod yer almaktadır.

```
faktoriyel(n)
    a=1
    for i = 1, 2, 3, ... , n+1 do
        a = a * i
    end for
    return a
```

Zaman Karmaşıklığı: $O(n)$

Bu algortmada n değerinin faktöriyel hesabı için döngü n defa gerçekleşecektir. Dolayısıyla bu algortmanın karmaşıklığı $O(n)$ 'dir. n değeri arttıkça işlemler için geçen zaman doğrusal olarak artar.

En İyi Durum: $O(1)$

$n=0$ değeri, hesaplama için en iyi durumdur. Bu durumda gerçekleştirilen işlemin zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - iteratif (run) X
run:
0! sonucu: 1
Geçen süre: 236900 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ortalama Durum: $O(n)$

n değeri arttıkça hesaplama işlemi için gereken süre doğrusal olarak artmaktadır. n değeri 15 seçilerek zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - iteratif (run) X
run:
15! sonucu: 1307674368000
Geçen süre: 352400 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

En Kötü Durum: $O(n)$

n değeri yüksek değerler seçildiğinde tüketilen zaman doğrusal olarak artacaktır. n=25 değeri için tüketilen zamanın derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - iteratif (run) ×
run:
25! sonucu: 7034535277573963776
Geçen süre: 871400 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```

6. Azalt ve Yönet (Decrease and Conquer) Stratejisi

Azalt ve yönet stratejisi, problemin veri boyutunu azaltarak çözüme kavuşturmaya yönelik bir stratejidir. Bu strateji için aşağıda eklemeli sıralama algoritması örneklenmiştir.

```
eklemeliSiralama(dizi[], diziUzunluk)
    for i = 1, 2, 3, ... , diziUzunluk do
        temp = dizi[i]
        j = i
        while j > 0 and dizi[j - 1] > temp do
            dizi[j] = dizi[j - 1]
            j = j - 1
        end while
        dizi[j] = temp
    end for
```

Zaman Karmaşıklığı: $O(n^2)$

Eklemeli sıralama algoritması n elemanlı bir dizide 2. veri için 1 kez, 3. Veri için 2 kez ve her veri için aynı şekilde devam ederek işlemler gerçekleştirilir. Dolayısıyla algoritmanın zaman karmaşıklığı $O(n^2)$ 'dir. Veri sayısı arttıkça zaman tüketimi de karesel olarak artacaktır.

En İyi Durum: $O(n)$

Dizinin sıralı bir şekilde bulunması durumu en iyi durum senaryosudur. Bu durumda dizi bir kez n-1 defa taranacaktır. 0-99999 arası sayıların sıralı bulunduğu dizi için zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - A_InsertionSort (run) ×
run:
Dizi: 0 1 2 3 ... 99998 99999
100000 adet veri sıralı yerleştirildi
Sıralandı: 0 1 2 3 ... 99998 99999
Geçen süre: 4417000 nanosaniye
BUILD SUCCESSFUL (total time: 0 seconds)
```


Ortalama Durum: $O(n^2)$

Dizide verilerin dağınık bir biçimde bulunma durumu ortalama durum senaryosudur. 0-99999 arası sayılar sırasıyla dizinin başına ve sonuna eklenerek elde edilen dağınık dizinin sıralanması işleminin zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - A_InsertionSort (run) ×
run:
Dizi: 0 2 4 6 ... 3 1
100000 adet veri dağınık yerleştirildi
Sıralandı: 0 1 2 3 ... 99998 99999
Geçen süre: 1811552600 nanosaniye
BUILD SUCCESSFUL (total time: 2 seconds)
```

En Kötü Durum: $O(n^2)$

Dizide verilerin tersten sıralı bir biçimde bulunma durumu en kötü durum senaryosudur. 0-99999 arası sayıların tersten sıralı bir şekilde yerleştirilerek elde edilen sırasız dizinin zaman tüketiminin derleyici üzerindeki görüntüsü aşağıdaki gibidir.

```
Output - A_InsertionSort (run) ×
run:
100000 adet veri tersen yerleştirildi
Sıralandı: 0 1 2 3 ... 99998 99999
Geçen süre: 3379553600 nanosaniye
BUILD SUCCESSFUL (total time: 3 seconds)
```

```
1  //***** 1.BOL VE YONET *****
2
3
4
5  package a_ikiliarama;
6
7  class BinarySearch {
8
9      int binarySearch(int dizi[], int bas, int son, int aranan) {
10
11          while (bas < son || bas == son) {
12              int orta = (bas + son) / 2;
13              if (dizi[orta] == aranan) {
14                  return orta;
15              } else if (dizi[orta] < aranan) {
16                  bas = orta + 1;
17              } else {
18                  son = orta - 1;
19              }
20          }
21          return -1;
22      }
23
24      public static void main(String args[]) {
25          BinarySearch bs = new BinarySearch();
26
27          //Siralı diziyi oluşturma
28          int[] dizi = new int[200000000];
29          for (int i = 0; i < 200000000; i++) {
30              dizi[i] = i;
31          }
32
33          //      int aranan = 99999999; //EN IYI
34          //      int aranan = 786541; //ORTALAMA
35          int aranan = 0; //EN KOTU
36
37          long baslama = System.nanoTime();
38          int sonuc = bs.binarySearch(dizi, 0, dizi.length - 1, aranan);
39          long bitis = System.nanoTime();
40          long sure = bitis - baslama;
41
42          System.out.println("Veri sayısı: " + dizi.length);
43          System.out.println("Aranan: " + aranan + " | Şu sırada bulundu: " + sonuc);
44          System.out.println("Geçen Süre: " + sure + " nanosaniye");
45      }
46  }
47
```

```

1  //***** 2.KABA KUVVET *****
2
3
4
5  package a_kabarcik;
6
7  public class A_Kabarcik {
8
9      public static void main(String[] args) {
10         int[] dizi = new int[100000];
11
12         ///-----EN IYI DURUM-----
13         for (int i = 0; i < 100000; i++) {
14             dizi[i]=i;
15         }
16         System.out.println("Dizi: "+dizi[0]+" "+dizi[1]+" "+dizi[2]+" "+dizi[3]+"
17         ... "+dizi[99998]+" "+dizi[99999]);
18         System.out.println("100000 adet veri sıralı yerleştirildi");
19         ///-----ORTALAMA DURUM-----
20         int a=0;
21         int b=99999;
22         boolean durum=true;
23         for (int i = 0; i < 100000; i++) {
24             if(durum==true){
25                 dizi[a]=i;
26                 a++;
27                 durum=false;
28             }
29             else{
30                 dizi[b]=i;
31                 b--;
32                 durum=true;
33             }
34         }
35         System.out.println("Dizi: "+dizi[0]+" "+dizi[1]+" "+dizi[2]+" "+dizi[3]+"
36         ... "+dizi[99998]+" "+dizi[99999]);
37         System.out.println("100000 adet veri dağınık yerleştirildi");
38         ///-----EN KOTU DURUM-----
39         int a = 99999;
40         for (int i = 0; i < 100000; i++) {
41             dizi[i] = a;
42             a--;
43         }
44         System.out.println("100000 adet veri tertem yerleştirildi");
45
46         int diziUzunluk = dizi.length;
47         int temp;
48
49         long baslamaZamani = System.nanoTime();
50
51         for (int i = 0; i < diziUzunluk - 1; i++) {
52             for (int j = 0; j < diziUzunluk - 1; j++) {
53                 if (dizi[j] > dizi[j + 1]) {
54                     temp = dizi[j];
55                     dizi[j] = dizi[j + 1];
56                     dizi[j + 1] = temp;
57                 }
58             }
59         }
60
61         long bitisZamani = System.nanoTime();
62         long geczenZaman = bitisZamani - baslamaZamani;
63
64         System.out.println("Sıralandı: " + dizi[0] + " " + dizi[1] + " " + dizi[2] +
65         " " + dizi[3] + " ... " + dizi[99998] + " " + dizi[99999]);
66         System.out.println("Geçen süre: " + geczenZaman + " nanosaniye");
67     }
68 }

```

```
1  //***** 3.REKURSIF *****
2
3
4
5  package a_fibonaccirecursive;
6
7  public class A_FibonacciRecursive {
8
9      public static long fib(long n) {
10         if ((n == 0) || (n == 1)) {
11             return n;
12         } else {
13             return fib(n - 1) + fib(n - 2);
14         }
15     }
16
17     public static void main(String[] args) {
18
19         // int n = 1; //EN IYI
20         // int n = 45; //ORTALAMA
21         int n = 50; //EN KOTU
22         long baslamaZamani = System.nanoTime();
23         System.out.println(n + ". fibonacci sayisi: " + fib(n));
24         long bitisZamani = System.nanoTime();
25         long gecenZaman = bitisZamani - baslamaZamani;
26         System.out.println("Geçen süre: " + gecenZaman + " nanosaniye");
27     }
28
29 }
30
```

```
1  //***** 4.DINAMIK *****
2
3
4
5  package a_fibonacci;
6
7  public class A_Fibonacci {
8
9      long fib(int a) {
10         long[] fib = new long[a];
11         fib[0] = 1;
12         fib[1] = 1;
13         for (int i = 2; i < a; i++) {
14             fib[i] = fib[i - 1] + fib[i - 2];
15         }
16         return fib[a - 1];
17     }
18
19     public static void main(String[] args) {
20
21         A_Fibonacci a = new A_Fibonacci();
22         // int n=1; //EN IYI
23         // int n=45; //ORTALAMA
24         int n = 5000; //EN KOTU
25         long baslamaZamani = System.nanoTime();
26         System.out.println(n + ". fibonacci sayısı: " + a.fib(n));
27         long bitisZamani = System.nanoTime();
28         long gecenZaman = bitisZamani - baslamaZamani;
29         System.out.println("Geçen süre: " + gecenZaman + " nanosaniye");
30
31     }
32
33 }
34
```

```
1  //***** 5.ITERATIF *****
2
3
4
5  package iteratif;
6
7  public class Iteratif {
8
9      long faktoriyel (int n){
10         long a=1;
11         for (int i = 1; i < n+1; i++) {
12             a=a*i;
13         }
14         return a;
15     }
16     public static void main(String[] args) {
17         Iteratif a= new Iteratif();
18         int n=0;
19         long baslamaZamani = System.nanoTime();
20         System.out.println(n+"! sonucu: "+a.faktoriyel(n));
21         long bitisZamani = System.nanoTime();
22         long gecenZaman = bitisZamani - baslamaZamani;
23         System.out.println("Geçen süre: "+gecenZaman+" nanosaniye");
24     }
25
26 }
27
```

```

1 //***** 6.AZALT VE YONET *****
2
3
4
5 package a_insertionsort;
6
7 public class A_InsertionSort {
8
9     int[] insertionSort(int[] dizi) {
10         int i, j, temp;
11         for (i = 1; i < dizi.length; i++) {
12             temp = dizi[i];
13             j = i;
14             while (j > 0 && dizi[j - 1] > temp) {
15                 dizi[j] = dizi[j - 1];
16                 j--;
17             }
18             dizi[j] = temp;
19         }
20         return dizi;
21     }
22
23     public static void main(String[] args) {
24         A_InsertionSort is = new A_InsertionSort();
25         int[] dizi = new int[100000];
26
27         ///-----EN IYI DURUM-----
28         for (int i = 0; i < 100000; i++) {
29             dizi[i]=i;
30         }
31         System.out.println("Dizi: "+dizi[0]+" "+dizi[1]+" "+dizi[2]+" "+dizi[3]+"
32         ... "+dizi[99998]+" "+dizi[99999]);
33         System.out.println("100000 adet veri sıralı yerleştirildi");
34         ///-----ORTALAMA DURUM-----
35         int a=0;
36         int b=99999;
37         boolean durum=true;
38         for (int i = 0; i < 100000; i++) {
39             if(durum==true){
40                 dizi[a]=i;
41                 a++;
42                 durum=false;
43             }
44             else{
45                 dizi[b]=i;
46                 b--;
47                 durum=true;
48             }
49         }
50         System.out.println("Dizi: "+dizi[0]+" "+dizi[1]+" "+dizi[2]+" "+dizi[3]+"
51         ... "+dizi[99998]+" "+dizi[99999]);
52         System.out.println("100000 adet veri dağınık yerleştirildi");
53         ///-----EN KOTU DURUM-----
54         int a = 99999;
55         for (int i = 0; i < 100000; i++) {
56             dizi[i] = a;
57             a--;
58         }
59         System.out.println("100000 adet veri tersen yerleştirildi");
60
61         long baslamaZamani = System.nanoTime();
62         int[] sonuc = is.insertionSort(dizi);
63         System.out.println("Sıralandı: " + sonuc[0] + " " + sonuc[1] + " " +
64         sonuc[2] + " " + sonuc[3] + " ... " + sonuc[99998] + " " + sonuc[99999]);
65         long bitisZamani = System.nanoTime();
66         long geczenZaman = bitisZamani - baslamaZamani;
67         System.out.println("Geçen süre: " + geczenZaman + " nanosaniye");
68     }
69 }

```