**Öğrenci Adı Soyadı: Engin Karataş**

**Öğrenci No: 16008118040**

**Öğrenim Gördüğü Bölüm: Bilgisayar Mühendisliği**

**Ödevin Adı: Sıralama algoritmaları ve ispatları**

**Dersin Adı: Algoritma Analizi Ve Tasarımı**

**Sıralama algoritmaları**

* 1. **Kabarcık Sıralama**

Worst complexity: n^2

Average complexity: n^2

Best complexity: n

1. baloncuk dizisi (sayı [] a)
2. {
3. işaretsiz n = a.uzunluk;
4. for (işaretsiz i = 0; i + 1 <n; i ++)
5. {
6. *// 1 (sonunda kısmen sıralanır, i = 0 için önemsiz şekilde doğrudur)*
7. *// [forall n-i <= num l && l <n:*
8. *// forall l <= k && k <n:*
9. *// a [k] <= a [l]*
10. *//]*
11. for (işaretsiz j = 0; j + 1 <n-i; j ++)
12. {
13. *// 2 (önceki tüm öğelerden daha yüksek; j = 0 için önemsiz şekilde doğrudur)*
14. *// [forall 0 <= num k && k <j:*
15. *// a [k] <= a [j]*
16. *//]*
17. eğer (a [j]> a [j + 1])
18. {
19. *// [a [j]> a [j + 1]] // 3a (if koşulundan önemsiz)*
20. a [j] <=> a [j + 1];
21. *// [a [j] <a [j + 1]] // 3b (önemsiz, takas)*
22. *// [a [j] <= a [j + 1]] // 3c (önemsiz, 3b'nin gevşemesi)*
23. }
24. else
25. {
26. *// [a [j] <= a [j + 1]] // 3d (önemsiz)*
27. }
28. *// [a [j] <= a [j + 1]] // 3 (iki öğe sıralı)*
29. *// [forall 0 <= num k && k <j + 1: a [k] <= a [j + 1]] // 4 (önceki tüm öğelerden daha yüksek + 1)*
30. }
31. *// [forall 0 <= num k && k <n-i: a [k] <= a [n-i]] // 5 (sonuncusu önceki tüm öğelerden daha yüksektir)*
32. *// [forall n-i-1 <= num l && l <n: forall l <= num k && k <n: a [k] <= a [l]] // 6 (+ 1 sonunda kısmen sıralanmış)*
33. }
34. *// [forall 0 <num l && l <n: forall l <num k && k <n: a [k] <= a [l]] // 7 (tümü sıralandı)*
35. *// "sıralı" önceden tanımlanmış bir yüklem ise [sıralı] olarak kısaltılabilir*
36. }

İspat etmeye "Takas" sonucuyla başladım daha sonra 3 e 3 ekledim, çünkü bu, 3'ün basitçe her iki if-dalına da katılarak kanıtlanabileceğini garanti eder.

İç döngünün en yüksek elemanı sonuna kadar hareket ettireceğini hatırladım, bu yüzden onu yazdım ve kanıtlamak için döngünün başlangıcına açıklama 2 ekledim. Sonra 6 yazdım, son elemanların sıralandığını. Bunu kanıtlamak için, 1 ve 5 ekledim. Bundan, 7'yi 5 ile aynı şekilde izler. Kanıtlayıcıda 5 veya 7'yi ispatlamak için matematiksel tümevarıma bile ihtiyacımız yok, sadece iç içe geçmiş döngünün son öğesi için açıklama örneğini ekleyin.Bu ek açıklamaların tamamen otomatik olarak kanıtlanabileceğine inanıyorum çünkü programdaki ifadeler hakkında önemsiz gerçeklerkontrol akış grafiğinin dalları üzerinde basit mantıksal birleşimler. Sanırım, bu ek açıklamaların çoğu otomatik olarak çıkarılabilir ve çıkarılabilir

**1.2 Kokyeyl Sıralama**

Worst complexity: n^2

Average complexity: n^2

Best complexity: n

İlk aşama, soldan sağa doğru sıralı kabarcık gibi dizi boyunca ilerler. Bitişik elemanlar karşılaştırılır ve eğer sol eleman sağ elemandan daha büyükse, o zaman bu elemanları değiştiririz. Listenin en büyük elemanı ileri geçişte dizinin sonuna yerleştirilir. İkinci aşama, en sağdaki en sıralanmamış öğeden sola doğru dizi boyunca ilerler. Bitişik elemanlar karşılaştırılır ve eğer sağ eleman sol elemandan daha küçükse, o zaman bu elemanları değiştiririz. Listenin en küçük elemanı geriye doğru geçişte dizinin başına yerleştirilir.

**1.3Tarak sıralama algoritması**

Worst complexity: n^2

Average complexity: n\*log(n)

Best complexity: n

Space complexity: 1

Method: Exchanging

Tarak sıralaması (veya Dobosiewicz Sırala), sırayla birden çok kez yinelenen, sıralı olmayan öğeleri değiştiren kabarcık sıralamasına benzer. Aradaki fark, tarak sıralamanın bitişik öğelere bakmaya başlamaması, bunun yerine boşluk adı verilen belirli bir mesafeye sahip öğelere bakmasıdır. Algoritma devam ettikçe bu boşluk giderek azalır.

Kokteyl sıralamasına benzer şekilde, tarak sıralaması, "kaplumbağa sorunu" (listenin sonuna yakın küçük öğeler algoritmayı yavaşlatır) ile başa çıkma yeteneği nedeniyle kabarcık sıralamasında gelişir.Ancak yine de aynı en kötü durum hesaplama karmaşıklığını korumaktadır.Ancak dikkat edilmesi gereken ilginç bir nokta, Tarak sıralamanın neredeyse Hızlı Sıralama kadar hızlı olmasıdır!Boşluk önce n'ye eşittir ve her yinelemeden sonra, 1 değerine ulaşana kadar bir küçültme faktörü (k) ile azaltılır. Bu nedenle, en sonunda, Tarak Sıralaması tam olarak Kabarcık Sıralama gibi davranır.

Küçültme faktörü, tarak ayırmanın verimliliği üzerinde önemli bir etkiye sahiptir. k = 1.3, orijinal makalenin yazarları tarafından 200.000'den fazla rastgele liste üzerinde yapılan ampirik testlerden sonra ideal bir küçültme faktörü olarak önerilmiştir. Çok küçük bir değer, gereksiz yere çok sayıda karşılaştırma yaparak algoritmayı yavaşlatır, çok büyük bir değer ise kaplumbağalarla etkili bir şekilde başa çıkmada başarısız olur ve bu da, bir boşluk büyüklüğünde birçok geçişi gerektirmesine neden olur.

1. Boşluğu ayarla = n

2. Takas gerektiğinde:

2.a. Aralığı her zaman> = 1 olacak şekilde k ile daraltın.

2.b. Boşluk ile ayrılmış her bir öğe çiftini karşılaştırın ve ters sırada iseler onları değiştirin.

2.c. Boşluk = 1 ise ve takas yapılmadıysa, durun: Bitirdik.

**1.4 Seçmeli Sıralama**

Worst complexity: n^2

Average complexity: n^2

Best complexity: n^2

Space complexity: 1

Döngü Değişmezi:

Her döngünün başlangıcından önce, A [dak], A [i..j-1] 'den küçük veya ona eşittir.

Başlatma:

Döngünün ilk yinelemesinden önce, j = i + 1. Yani dizi segmenti A [i..j-1]

gerçekten sadece nokta A [i]. Kodun 2. satırı min = i ayarladığından,

min, A [i..j-1] alt dizisindeki en küçük öğeyi (tek öğe) dizine ekler ve

dolayısıyla döngü değişmezi doğrudur.

Bakım: J'yi geçmeden önce, min'nin alt dizideki en küçük elemanı indekslediğini varsayıyoruz

A [i..j-1]. J iterasyonu sırasında iki durumumuz var: ya A [j] <A [min]

veya A [j] ≥ A [dak]. İkinci durumda, 4. satırdaki if ifadesi doğru değildir,

yani hiçbir şey yürütülmez. Ama şimdi min, A [i..j] 'nin en küçük öğesini indeksliyor.

İlk durumda, satır 5, en küçük olduğu için min'i j dizin konumuna geçirir.

Min, alt dizi A [i..j-1] 'den küçük veya ona eşit bir öğeyi dizine eklerse ve şimdi

A [j] <A [min], bu durumda A [j] şundan küçük veya eşit olmalıdır

A alt dizisindeki elemanlar [i..j-1]. Satır 5, bu yeni konumu indekslemek için min

ve dolayısıyla döngü yinelemesi bittikten sonra, min, içindeki en küçük öğeyi dizine ekler

alt dizi A [i..j].

Sonlandırma:İç döngünün sonlandırılmasında, min, hepsinden küçük veya hepsine eşit bir elemanı indeksler

sonlandırma üzerine j = n + 1 olduğundan, A [i..n] alt dizisindeki elemanlar. Bu bulur bu alt dizideki en küçük eleman ve dış döngüde bizim için yararlıdır çünkü biz sonraki en küçük öğeyi doğru konuma taşıyabilir.

**1.5 Eklemeli Sıralama**

Worst complexity: n^2

Average complexity: n^2

Best complexity: n

Space complexity: 1

Dış döngünün her yinelemesinin başlangıcında ( ), alt dizi orada bulunan orijinal öğelerden oluşur, ancak sıralı sırayla.

Ekleme Sıralamasının doğru olduğunu kanıtlamak için bunu üç aşamada göstereceksiniz:

Başlatma - Alt dizi, dizinin ilk öğesi ile başlar ve (belli ki) başlamak üzere sıralanır.

Bakım - Döngünün her yinelemesi alt diziyi genişletir, ancak sıralanmış özelliği korur. Bir element V diziye yalnızca solundaki öğeden daha büyük olduğunda eklenir. V Solundaki öğeler zaten sıralandığından, şu anlama gelir:solundaki tüm öğelerden daha büyük olduğundan dizi sıralı olarak kalır. ( Ekleme Sıralaması 2'de bunu, her öğe düzgün bir şekilde eklendiğinde diziyi yazdırarak gördük.)

Fesih - Kod sonra sona erecekdizideki son öğeye ulaştı, bu, sıralanan alt dizinin tüm diziyi kapsayacak şekilde genişlediği anlamına gelir. Dizi artık tam olarak sıralanmıştır.

**1.6 Kabuk sıralaması**

Worst complexity: Depends on gap sequence

Average complexity: n\*log(n)^2 or n^(3/2)

Best complexity: n

Bilgisayar bilimlerinde kullanılan sıralama algoritmalarından birisi de kabuk sıralamadır (shell sort). İsmi Türkçeye kabuk sıralaması olarak çevrilsede aslında Donald Shell isimli algoritmayı ilk bulan kişinin isminden gelmektedir. Algoritma performansı O(n2)’dir. Çalışması aşağıdaki örnek üzerinde anlatılmıştır:

Sıralayacağımız sayılar:

5,7,2,9,6,1,3

olarak verilmiş olsun. Sıralama işlemi için öncelikle bir atlama miktarı belirlenir. Atlama miktarının belirlenmesi için çok çeşitli yollar bulunmasına karşılık en basit yöntem elimizdeki sayıların yarısından başlamaktır. Yani yukarıdaki örnekte elimizde 7 sayı olduğuna göre 3 atlama miktarı ile başlanabilir.Sırasıyla her sayı kendinden 3 sonraki sayı ile karşılaştırılır ve bu sayılar kendi aralarında sıralanır. Bu sıralamayı daha rahat göstermek için aşağıdaki kolon gösterimi kullanılabilir:

5,7,2

9,6,1

3

Her kolon kendi içinde sıralanınca aşağıdaki sayılar elde edilir:

3,6,1

5,7,2

9

Yukarkıdaki örnekte de görüldüğü üzere sayıları sıralama işleminin 3te biri bitirilmiştir. Ardından atlama miktarı yarıya indirilir (bu örnek için 3/2 = 1 olur) Bu durumda bütün sayılar tek bir doğrultuda sıralanır.

3,6,1,5,7,2,9

1,2,3,5,6,7,9

Yukarıdaki sıralamada örneğin kabarcık sıralaması (bubble sort) kullanılırsa dizinin orjinal haline göre (yani kabuk sıralamasının 3 atlamalı sıralama işlemi yapılmamış haline göre) çok daha başarılı olduğu görülür.

**1.7 Ağaç Sıralama**

Worst complexity: n\*log(n) (balanced)

Average complexity: n\*log(n)

Best complexity: n\*log(n)

Space complexity: n

**1.8 Kütüphane sıralaması**

Worst complexity: n^2

Average complexity: n\*log(n)

Best complexity: n

Space complexity: n

**1.9 Birleştirmeli(merge)**

Worst complexity: n\*log(n)

Average complexity: n\*log(n)

Best complexity: n\*log(n)

Space complexity: n

İlk varsayım: kullandığınız birleştirme yordamı, sıralanmış iki diziyi sıralı bir dizi halinde birleştirir. İkinci varsayım: birleştirme rutini sona erer

Temel durum: n = 1, 1 elemanlı dizi her zaman sıralanır

Endüktif hipotez: n = 1,2, ..., k için birleştirme sıralaması çalışır

Endüktif adım: n = k + 1

Şimdi endüktif adımın doğru olduğunu kanıtlamamız gerekiyor.

Birleştirme sıralaması, diziyi L = [1, n / 2] ve R = [n / 2 + 1, n] olmak üzere iki alt diziye böler. Yukarıdaki gerçeklere göre tavan (n / 2) k'den küçüktür. Tümevarımlı hipotezimize göre, L ve R için birleştirme sıralamanın her iki sonucu da doğru şekilde sıralanır ([1, k] aralığında oldukları için). Dahası, varsayımımızdan, birleştirme yordamı, onları tüm öğeleri içeren sıralı bir dizide birleştirir çünkü boyut (L) + boyut (R) = n, yani bu, n = k + 1 boyutundaki bir diziyi doğru şekilde sıraladığı anlamına gelir.

@Düzenle: üzgünüm, yanlış okundu. Birleştirme kısmı için:

Burada çok boyutlu bir tümevarıma sahip olacağız.

Varsayım: X, Y girdi dizileri zaten sıralanmıştır

Temel durum: size (X) == 0 && size (Y)> = 0 => return Y || size (Y) == 0 && size (X)> = 0 => X, bu doğrudur çünkü hem X hem de Y sıralanır ve sıralı bir diziyi boş bir diziyle birleştirmek bize aynı boş olmayan diziyi verir

X üzerinde indüktif hipotez: birleştirme, n = 1,2,3, ..., k && boyut (Y)> = 0 olduğunda birleştirme için çalışır (n, boyut (Y))

Y üzerine indüktif hipotez: birleştirme, m = 1,2,3, ..., l && boyut (X)> = 0 olduğunda birleştirme için çalışır (boyut (X), m)

X üzerinden endüktif adım: n = k + 1

Y üzerinden endüktif adım: m = l + 1

X'e göre ilk tümevarım adımı için, temel durumun yanı sıra 2 durumumuz var:

X [1] <Y [1] => X [1] ⊕ birleştirme (kuyruk (X), Y) => bu, X üzerindeki hipotezimize göre birleştirme (k, boyut (Y)) doğru olduğu için doğrudur ve biz daha küçük olan öğeyi öne koyuyoruz, böylece düzeni sağlıyoruz

X [1]> = Y [1] => Y [1] ⊕ birleştirme (X, kuyruk (Y)) => burada iki seçeneğimiz var:

boyut (kuyruk (Y)) = 0 => bir temel duruma çarptık, bu nedenle bu durumun doğru olduğu kanıtlandı

boyut (kuyruk (Y))> 0 => nihayet temel duruma vurarak veya boyut (kuyruk (X)) = k => indüktif hipotezimiz tarafından kanıtlandığı yerde birleştirerek (kuyruk (X), alt dizi (Y)) tekrar tekrar tekrar ederiz

Benzer şekilde Y üzerinden indüksiyon adımı için:X [1]> = Y [1] => Y [1] ⊕ birleştirme (X, kuyruk (Y)) => bu, hipotez birleştirme (boyut (X), l) `tarafından doğru ve biz öndeki daha küçük eleman X [1] <Y [1] => X [1] ⊕ birleştirme (kuyruk (X), Y) => burada iki seçeneğimiz var: boyut (kuyruk (X)) = 0 => bir temel duruma çarptık, bu nedenle bu durumun doğru olduğu kanıtlandı

boyut (kuyruk (X))> 0 => nihayet temel duruma ulaşarak veya boyut (kuyruk (Y)) = l => indüktif hipotezimiz tarafından kanıtlandığı yerde birleştirerek (alt dizi (X), kuyruk (Y)) tekrar tekrar tekrar ederiz. Algoritma, her adımda dizilerden birini 1 eleman küçülttüğümüz için sona erer, dolayısıyla bunlardan biri sonunda temel durumumuza ulaşacaktır.

**2.1 Yığın sıralaması**

Worst complexity: n\*log(n)

Average complexity: n\*log(n)

Best complexity: n\*log(n)

Yığın sırasının bir büyük dezavantajı, her zaman Θ (n lg n) ile çalışmasıdır. Bunun nedeni, maks-yığının yapısıyla ilgilidir. Yığın sıralamasında örtük ikili yığın oluşturduğumuzda, yığının en büyük öğesi her zaman dizinin en sol tarafındadır, ancak sonuçta öğe sağ tarafta olmalıdır. Max öğesini uygun konumuna taşımak için, onu yığının esasen rastgele bir öğesi ile değiştirmeli ve ardından öbeği yeniden dengelemek için bir baloncuk azaltma işlemi yapmalıyız. Bu baloncuk azaltma işlemi Ω (lg n) zaman alır ve genel Ω (n lg n) çalışma süresine katkıda bulunur. Sahip olunması gereken bariz bir içgörü şudur: Ya bir maksimum yığın oluşturursak ancak maksimum öğeyi dizinin sağında saklarsak? Bu şekilde, maksimum elemanı dizinin sonundaki bir sonraki açık noktaya taşımak istediğimizde, zaten işimiz bitmiştir - maksimum eleman artık doğru konumda olacaktır. Şimdi tek yapmamız gereken, geri kalan unsurları yeniden dengelemek. Burada oldukça güzel bir mülkle karşılaşıyoruz. Yığının kökünü çıkardığımızda ve kök altında yaşayan iki maksimum yığını ortaya çıkarmak için "yığını kırdığımızda". Bunların her biri maksimum yığınların kökleri olduğu için, ilgili maksimum yığınlarının tüm öğelerinden daha büyüktür ve bu nedenle bu iki öğeden biri, kalanların en büyük öğesidir. Ancak şimdi sorun şu ki, güzel maks-yığınımızı iki farklı maksimum yığın olarak ayırdık ve onları yeniden dengelemeyi bilmemizin tek yolu bir yaprağı değiştirip onu aşağı doğru balonlamaktır. Bu katil adımdır, çünkü neredeyse kesinlikle Ω (lg n) zaman alacak ve çalışma zamanımızı Θ (n lg n) olmaya zorlayacak. En iyi durumda O (n) elde etmeyi hedefliyoruz ve bu yüzden bu işe yaramayacak. İşte düzgün sıralamayı mümkün kılan anahtar fikir - ya tek bir maksimum yığın yerine, bir maksimum yığın ormanımız varsa? Yani, tek bir maks yığınına sahip olmak yerine, diziye gömülü bir maksimum yığın dizisi tutacağız. Bu şekilde, bir yığını tekrar bir araya getirmeden birden çok parçaya ayırırsak sorun olmaz. Herhangi bir zamanda çok fazla yığına sahip olmamak koşuluyla (örneğin, O (lg n)), geriye kalanların en büyük öğesini verimli bir şekilde bulabiliriz. Yüksek düzeyde, düzgün sıralama aşağıdaki gibi çalışır. İlk olarak, girdi dizisi üzerinde doğrusal bir tarama yaparız ve bunu bir örtülü maks-yığın dizisine dönüştürürüz. Bu yığınlar ikili yığınlar olmayacak, bunun yerine aşağıda Leonardo yığını adı verilen alışılmadık bir yığın türü olacaktır. Bunu yaparken, yığınların üst öğelerinin artan sırada olması özelliğini koruyarak, en sağdaki öbeği kalan öğelerin maksimumunu tutmaya zorlarız. Bunu yaptıktan sonra, en sağdaki maksimum yığının üst öğesi, en sağdaki doldurulmamış noktada olduğu için doğru konumda olan sürekli olarak kuyruktan çıkarılacaktır. Daha sonra yığınları ve sıralanmış özelliği yeniden oluşturmak için bazı manipülasyonlar yapacağız. Bu garantiler ve biraz akıllı matematik, algoritmanın sıralı diziler üzerinde hızlı çalışmasını garanti eder. Smoothsort'un ilk uygulaması, mükemmel çalışma zamanı garantilerine, ancak yüksek bellek kullanımına (O (n)) sahip olacak. Daha sonra, bunu O (lg n) değerine sıkıştıran bir optimizasyon ve son olarak, alan gereksinimini O (1) 'e düşüren teorik olarak şüpheli bir hile göreceğiz.

**2.2 Rahat sıralama**

Worstcomplexity: n\*log(n)

Average complexity: n\*log(n)

Best complexity: n

Space complexity: 1

**2.3 Hızlı Sıralama**

Worst complexity: n^2

Average complexity: n\*log(n)

Best complexity: n\*log(n)

Verinin hafızada sıralı tutulması için geliştirilen sıralama algoritmalarından (sorting algorithms) bir tanesidir. Basitçe sıralanacak olan dizideki orta noktada (mean) bulunan bir sayıyı seçerek diğer bütün sayıları bu orta sayıdan büyük veya küçük diye sınıflayarak sıralama yapmayı hedeflemektedir. Bu açıdan bir parçala fethet (divide and conquere) yaklaşımıdır. Ayrıca bu seçilen orta noktaya eksen (pivot) adı da verilir çünkü bütün diğer sayılar bu sayının ekseninde sıralanacaktır.

Sıralanmak istenen verimiz:

5,7,2,9,6,1,4,7

olsun. Bu verilerin bir oluşumun(composition) belirleyici alanları olduğunu düşünebiliriz. Yani örneğin vatandaşlık numarası veya öğrenci numarası gibi. Dolayısıyla örneğin öğrencilerin numaralarına göre sıralanması durumunda kullanılabilir.

Yukarıda verilen bu örnek dizinin sıralanması adım adım aşağıda anlatılmıştır:

1. adım, dizinin ortasındaki sayı bulunur. Bu örnekte 8 sayı olduğu için ortadaki sayı 4. elemandır. Bu elemanın değeri de 9’dur. Bu durum aslında biraz bahtsız bir durumdur çünkü tesadüfen dizideki en büyük sayıdır. Bu mesele 2. adımda ortaya çıkacaktır.

2.adım: Diziden 1.adımda seçilen sayıya göre dizideki bütün elemanları küçük veya büyük diye sınıflandır. Tabi 1. adımda şanssız bir şekilde en büyük sayı seçildiği için bütün sayılar 9’dan küçük olarak sınıflandırılacaktır.

5,7,2,6,1,4,7 (9)

3. adım: Sınıflandırılana küçük ve büyük dizileri tekrar hızlı sıralamaya ver. Yani 5,7,2,6,1,4,7 dizisini aynı adımlarla tekrar sıralıyoruz.

4. adım: eleman sayımız 7 ve ortadaki eleman 3. eleman olan 6 olur. Dizideki sayılar 6’dan büyük ve 6’dan küçük diye sınıflandırılırsa:

5,2,1,4 (6) 7,7

olarak iki grup elde edilir. Bu grupları da sıralamak üzere tekrar hızlı sıralama algoritmasına veririz. Dolayısıyla 5,2,1,4 sayıları ayrı ve 7,7 sayıları ayrı ayrı sıralanacaktır.

5. adım 5,2,1,4 sayılarının orta değeri 2’dir ve sınıflandırılırsa:

1 (2) 4,5 bulunur. Aynı zamanda diğer dizi olan 7,7 sıralanırsa sonuç değişmez ve 7,7 bulunur.

6. adım 1 sıralanırsa 1 ve 4,5 sıralanırsa 4,5 bulunur.

7.adım. Bu adımdan sonra artık birleştirme işlemine geçilebilir. Buna göre 6. adımda sıralanan değerleri birleştirirsek :

1,2,4,5 değerleri elde edilir.

8.adım: 4. adımdaki sayılar birleştirilirse 1,2,4,5,(6),7,7 sayıları elde edilir.

9.adım: 2. adımdaki sayılar birleştirilirse 1,2,4,5,6,7,7 (9) olarak dizinin sıralanmış hali elde edilir.

**2.4 İçgözleme Sıralama**

Worst complexity: n\*log(n)

Average complexity: n\*log(n)

Best complexity: n\*log(n)

Space complexity: log(n)

**2.5 Sabır sıralama**

Worst complexity: n\*log(n)

Best complexity: n

Space complexity: n

**2.6 İplik Sıralama**

Worst complexity: n^2

Average complexity: n^2

Best complexity: n

***Ödev: Engin Karataş***