

Gebze Technical University
Computer Engineering

CSE 222
2017 Spring

HOMEWORK 2 REPORT

FÜRKAN YILDIZ
141044031

Course Assistant : NUR BANU ALBAYRAK

Q1)

1)

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 3 = \sum_{i=1}^{n-1} 3 = \sum_{i=0}^{n-2} 3(n-1-i) = -3/2 n^2 + 9/2 n - 3$$

En büyük derece n^2 olduğundan ötürü fonksiyonun çalışma süresi $O(n^2)$ dir.

2)

Bu fonksiyonu best ve worst case olarak ikiye ayırarak incelemeliyiz çünkü recursive if koşuluna bağlı olarak değişik şekilde çalışabilir. Örneğin string boş veya null ise sadece 1 kez çalışacak diğer durumlarda sonuna kadar çalışacak.

(best case) $T(1) = 1$

Worst case için ise fonksiyon base caseye gelene kadar çağırılmalı.

Şu şekilde inceliyebiliriz,

Methoddaki else kısmını inceleyecek olursak, recursive çağırımdan sonra 1 ekleme olayı constant ve bu işlemin ardından return etme işlemide constant olduğundan bunları $1+1$ den $= 2$ şeklinde hesaplayacak olursak,

$$\begin{aligned} T(n) &= 2+T(n-1) \\ &= 2+2+T(n-2) \\ &= 2+2+2+T(n-3) \\ &= 2+2+2+...+2+T(1) \\ &= 2(n-1)+\theta(1)=\theta(n) \end{aligned}$$

Sonuç olarak

(best case) $T(n) = \theta(1)$

(worst case) $T(n) = \theta(n)$

Q2)

1)

Fonksiyon verilen bir diziyi küçükten büyüğe doğru sıralar. Bunu iç içe 2 for ile yapar. Dıştaki for sırasıyla arraydeki elemanın indexini tutacak içteki for ise o tutulan indexin devamında, tutulan indexdeki değerden daha küçük bir değer var mı diye if conditionu ile test edecek. Eğer var ise bu ikisini yer değiştirecek.

2)

Best case durumu arraydeki tüm elemanların sıralı olması durumudur. Worst case durumu ise tüm elemanların büyükten küçüğe (algoritmanın yapmak istediğinin tam tersi) olması durumudur. Yazılan bu algoritmaya göre, sıralanmak istenilen dizinin durumunun (zaten sıralı mı ya da sırasız mı) hiçbir önemi yoktur. Algoritma her koşulda yazılan iki adet for döngüsünede girecektir. Çünkü dışarıdaki fordan sonra içerideki for için hiçbir koşul yoktur. Bu sebeple bu algoritmanın best casesinden ve worst casesinden bahsedemeyiz. Her türlü aynı sürede çalışacaktır. Çalışma süresini ise şu formül ile hesaplayabiliriz.

$$\sum_{j=1}^{n-2} \sum_{i=j+1}^{n-1} k, \text{ k: constant bir tam sayı. (karşılaştırma ve Exchange işlemlerinin yapılma süresi)}$$

Bu formülü açtığımızda ise karşımıza n^2 li bir ifade gelecektir. (küçük terimlerin ve 'nin katsayısının önemi yok)

Buna göre bu algoritmanın çalışma süresi $\theta(n^2)$ dir.

Q3)

Bir algoritmanın çalışma süresini $T(n)$ olarak ifade edecek olursak, $T(n) = \theta(n)$ dir. $0 \leq c_1 g(n) \leq T(n) \leq c_2 g(n)$ $n \geq n_0$ koşulunu sağlayan n değerleri için.

$0 \leq c_1 g(n) \leq T(n)$ ifadesini $n \geq n_0$ şartıyla sağlayan her n değeri için $T(n) = \Omega(g(n))$ diyebiliriz. $T(n)$ 'nin lower bound 'u (best-case çalışma süresi) $\Omega(n)$ dir.

$T(n) \leq c_2 g(n)$ ifadesini $n \geq n_0$ şartıyla sağlayan her n değeri için ise $T(n) = O(g(n))$ diyebiliriz. $T(n)$ 'nin upper bound 'u (worst-case çalışma süresi) $O(n)$ dir.

Q4)

1)

```
public static void insSort(int array[], int size){
    if (size>1) {
        insSort(array, size-1);
        int key = array[size-1];
        int i=size-2;
        while (array[i] > key && i>=0 ) {
            array[i+1] = array[i];
            --i;
        }
        array[i+1] = key;
    }
}
```

2)

Yazdığımız algoritmaya göre, metoda gelen arrayin size'ı birden büyük ise metod tekrar çağırılacak ta ki size 1 olana (array sıralanana) kadar ve çağırma işlemleri bittikten sonra arrayin işlem görülen indexindeki eleman ile, o indexden önce gelen elemanlar karşılaştırılacak, eğer daha önce gelenler büyük iseler bir yer değiştirme işlemi yapılacaktır.

Buna göre, bu methodun n girdili bir işlemi $T(n)$ sürede yaptığını varsayar isek, Her recursive method çağırıldığında n değeri bir azalacağından k . kez çağırıldığında bu süre $T(n-k)$ olur, (örneğin 2. Sefer fonksiyon çağırıldığında $T(n-1)$ sürede çalışır.)

Recursive çağırmadan sonra, çağırılan indexin elemanı ile önceki indexlerdeki elemanlar karşılaştırılıyor, bu işlemde $(n-1)$ sürede gerçekleşir.

Tüm bunlara bakacak olursak,

Reccurence relation: $T(n) = T(n-1) + (n-1)$ 'dir.

3)

$$\begin{aligned}T(n) &= T(n-1) + (n-1) \\&= T(n-2) + (n-2) + (n-1) \\&\vdots \\&= T(2) + 2 + 3 + \dots + (n-2) + (n-1) \\&= T(1) + 1 + 2 + 3 + \dots + (n-2) + (n-1) \\&= 1 + 2 + 3 + \dots + (n-2) + (n-1) \\&= \theta(n^2)\end{aligned}$$

Method 2. kez çağırıldığında, size bir azalacağından $T(n-2)$ olur çalışma süresi ve karşılaştırma yapmak için oluşturulan while döngüsü ise yine size bir azalacağından $(n-2)$ süre alır. Yani 2. kez method çağırıldığında sadece 2. methodun çalışma süresi $T(n-2) + (n-2)$ olur. Aynı şekilde 3. Kez method çağırıldığında onun çalışma süresi $T(n-3) + (n-3)$ olacaktır. Bu şekilde method çağırılırken verilen girdi azalacak ve en son size ikiye indiğinde ise tek bir karşılaştırma yapılacak arrayin ilk elemanı ile ikincisi karşılaştırılacak ve bu constant time sürecektir. $T(2) = 1$, size 1 e düşünce ise hiçbir şey yapılmayacak $T(1) = 0$

Bu şekilde denklemimizi oluşturduğumuzda ve tüm süreyi yukarıdaki gibi hesapladığımızda algoritmamız toplam $\frac{(n-1) \cdot n}{2}$ sürede çalışacak, buradan da n^2 'li bir ifade gelecek katsayıları ve düşük dereceli çarpanları işleme katmadığımız için buradan bu algoritmanın çalışma süresinin $\theta(n^2)$ olduğunu söyleyebiliriz.

Q5)

3 soruda da,

$T(N) \leq cf(N)$ when $N \geq n_0$ O notasyonu için

$T(N) \geq c f(N)$ when $N \geq n_0$ Ω notasyonu için

Ve eğer her ikisi de doğru ise θ notasyonunu kullanacağız.

Bu tanımlardan yola çıkarak c ve n_0 değerleri bularak sorular aşağıdaki gibi çözüldü.

1)

$\sqrt[10]{n} \geq C \log^{10} n$ $n > n_0$ için bu şartı sağlayıp sağlamadığını kanıtlarsak omega ile gösterip gösteremeyeceğimizi kanıtlamış oluruz. $n_0 = 10^{1000}$ $c = 1$ seçersek. Bu şartın sağlandığını görebiliriz. $10^{1000} \geq (1000)^{10}$ ifadesi doğru bir ifadedir. Bu sebeple $f(n) = \Omega(g(n))$ ifadesi doğru bir ifadedir.

$\sqrt[10]{n} \leq C \log^{10} n$ $n > n_0$ değerinin doğruluğunu ise herhangi bir n_0 değeri için gösteremeyiz çünkü sayıyı eğer pozitif seçersek n_0 değerleri küçükken bu ifade doğru olsa bile n_0 değeri büyüdükçe bu ifade yanlış olacak. Negatif n_0 değerleri için ise bu ifade tanımsız çünkü logaritma negatif değerler alamaz. Bu yüzden $f(n) = O(g(n))$ ifadesi yanlış bir ifadedir.

$f(n) = O(g(n))$ ifadesi yanlış iken $f(n) = \Omega(g(n))$ ifadesi doğru olduğu için $f(n) = \Theta(g(n))$ ifadeside yanlış bir ifadedir.

Sonuç olarak bu ifadeyi $f(n) = \Omega(g(n))$ ile gösterebiliriz.

2)

$n! \geq 2^n$ $n > n_0$ için bu şartı sağlayıp sağlamadığını kanıtlarsak omega ile gösterip gösteremeyeceğimizi kanıtlamış oluruz. $n = 10$ için $n!$ her zaman 2^n 'den büyüktür. Faktoriyelli ifade sonsuza gittikçe çok daha hızlı bir şekilde büyüyecektir. Bu yüzden $f(n) = \Omega(g(n))$ ifadesi doğru bir ifadedir.

Negatif sayılar faktöriyelde tanımsız olduğundan dolayı ise sadece pozitif sayılar ile karşılaştırma yapabiliyoruz onunda sonsuza gittikçe $n!$ in büyük olduğunu gördük.

Sonuç olarak bu ifadeyi $f(n) = \Omega(g(n))$ ile gösterebiliriz.

3)

$n_0 = 2^{100}$ $c = 1$ değerlerini seçersek, n değeri sonsuza gittikçe $2^{(\log_2 n)^2}$ ifadesi daha büyük değere ulaşacağı için $(\log n)^{\log n} \leq 2^{(\log_2 n)^2}$ dir. Bu yüzden $f(n) = O(g(n))$ ifadesi doğru bir ifadedir.

Ω yı test etmek için eşitliğin tam tersini kanıtlamalıyız fakat n_0 değerine negatif tamsayılar veremiyoruz logaritma tanımsız olduğundan dolayı o aralıkta. Dolayısıyla tek çözüm $O(g(n))$ dir.

Sonuç olarak bu ifadeyi $f(n) = O(g(n))$ ile gösterebiliriz.

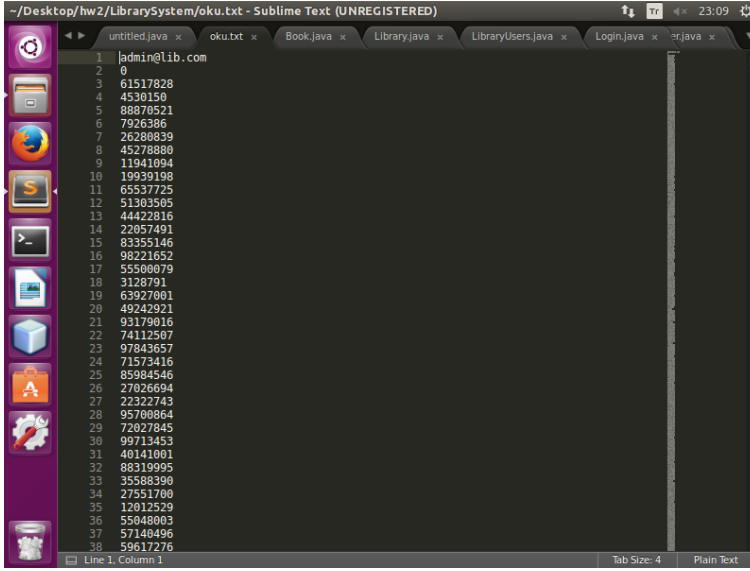
Q6)

Problem Solutions Approach

Kitap ekleme methodu üzerinden linked list ve array list i karşılaştırmaya çalıştım. (Neden bu method çünkü en geniş kapsamlı olan method bu, hem eklenecek kitap için staff'tan 4 adet input alıyor, hem de eklenmek istenile kitabın id'sinin daha önce olup olmadığını daha önceki kitaplar üzerinde search işlemi yaparak bulamaz ise ekliyor.) n değil dosyadan almasını sağladım. Dosyadan inputları alırken önce staff olarak giriş yapıyor daha sonra kitap ekleme seçeneğini seçiyor ve ekleyeceği kitabın id'sini adını ve yazarını okuyor dosyadan ve kitabı listeye ekliyor. Bu şekilde 5000,15000,20000 ve 25000 adet kitap ekleme kodu yazdım ve bunu arraylist ve linkedlist için ayrı ayrı test ettim. Arraylist ile array'in sonucu ise hemen hemen çok yakın olacak çünkü düzgün impliment edildiklerinde hiçbir farkları olmayacak birbirlerinden.

2. Test Cases

Yazdığımız programda stafflar 3, userlar 2 adet operasyona sahip, kodun en iyi şekilde arraylist ve linkedlist ile karşılaştırılmasını incelemek için en detaylı method olan kitap ekleme methodunu kullandım. Bu methoda random olarak oluşturulan dosyadaki inputları okuttum ve 5000-15000-20000-25000 adet kitap ekledim databaseye.



Inputları dosyadan aldirmek için şu şekilde bir random sayılarla dolu dosya oluşturdum dosyadan kitabın id'sini, adını ve yazarını random olarak alıyor. Deneme yaptığımız için kitap adı ve yazarını string oluşturmanın bir anlamını görmedim. Sayıları büyük seçtiğim için string boyutundan farklı olmadı durum.

3. Running and Results

Eklenen kitap sayısı	Linked List ile ekleme süresi	Array List ile ekleme süresi
5000	16.921 s	11.683 s
15000	2 dakika 8.500 s	1 dakika 19.408 s
20000	2 dakika 14.7308 s	3 dakika 37.024 s
25000	3 dakika 20.846 s	6 dakika 54.156 s

Önce 5000 kitap eklemek için programın buradan inputları alacağı random bir dosya hazırlandı. Bu dosyadan inputlar alınarak, Linked list ve array list kullanılarak sadece kitap ekleme operasyonu gerçekleştirildi ve buna göre programın toplam çalışma süresi ayrı ayrı hesaplandı. Daha sonra aynı işlem 15000, 20000 ve 25000 adet kitap ekleme yapılarak test edildi ve süreler kaydedildi.

Arraylist için aldığım bu sonuçların array içinde geçerli olduğunu düşünüyorum.

Sonuç olarak benim yazdığım programa göre, data küçükken array list daha hızlı iken, data büyüdükçe linked list daha hızlı hale geliyor.