Böl ve Yönet Algoritmaları

Öğrenci Adı: Burak Atalay

Öğrenci Numarası: 22011641

Dersin Eğitmeni: M. Elif Karslıgil

Video Linki: https://youtu.be/xGo712tYRt8

1- Problemin Çözümü:

Problem için iki şekilde çözüm yapıldı. Önce yollanacak çözüm sonra ikinci çözüm anlatılacak.

Birinci Çözüm:

Alınan dizi mergeSortKWay fonksiyonunda k parçaya bölünüp her bir parça için çağrı yapılır. Base case olarak n'nin k'dan küçük olduğu durum vardır çünkü dizi daha fazla bölünemez. n = 1 ise dizi zaten sıralıdır ve doğrudan return olunur ancak değilse ve k'dan küçükse sıralamak için insertion sort kullanıldı. Çağrılardan k sıralı parça olarak dönüldükten sonra birleştirme işlemi için merge fonksiyonu çağrılır. Merge fonksiyonunda her bir parçanın başlangıç indisleri ve son indisleri birer dizide tutulur. Burada birleştirme işlemindeki yaklaşım geçen dönem veri yapıları üçüncü ödevindeki k adet heapten max elemanı seçme kısmına benzer şekildedir. Burada da aslında küçükten büyüğe sıralı diziler min heap yapısına uyuyor ve root'u yani min elemanı çıkardıktan sonra indisi ilerletmek heap yapısını bozmuyor. Bu sebeple her bir parçanın başlangıç indislerini tutan dizi yardımıyla parçaların o anki min elemanları arasında linear şekilde k adımda gezerek min elemanı bulunup n uzunluğundaki yardımcı diziye eklenir. Sonrasında yardımcı dizi gerçek diziye kopyalanır.

İkinci Çözüm:

İkinci çözümün birincisinden tek farkı parçalardan min eleman seçilirken O(k) karmaşıklıkta linear arama yapmak bir heap kullanarak bunu logk karmaşıklıkta yapmaktır. Parçaların başlangıç indislerindeki elemanlar struct yapısındaki bir heap'e atılıp buildHeap yapılır. Struct'ta iki değişken vardır: eleman ve elemanın ait olduğu parça. Sonrasında her iterasyonda min eleman seçilir ve elemanı seçilen parçadan heap'e yeni bir insert yapılır. Hangi parçanın elemanın seçildiği struct'ta tutulan değişken yardımıyla bilinir. Yine parçaların min elemanlarını takip için birinci çözümde kullanılan başlangıç ve bitiş indislerini tutan dizilerden faydalanılır.

2- Karşılaşılan Sorunlar:

Ödevde ilk aşamada birinci çözüm yapıldı ve sonrasında performansı artırmak adına ikinci çözüm implemente edildi. Ancak k 2'den 10'a kadarki performans karşılaştırıldığında birinci yöntemin k = 7'ye kadar daha iyi performans gösterdiği görülünce birinci çözümde karar kılındı. Buradaki sebep olarak k küçükken ikinci çözümdeki heap'li yaklaşımın heap kullanımından ötürü overheadinin çok olması söylenebilir çünkü ikinci çözüm grafiğinde k artarken düzenli bir düşüş görülürken birinci çözümde k = 5'e kadarki düşüşten sonra zigzaglı bir yükseliş söz konusudur. Bunun için de sebep olarak birinci çözümde k*n karmaşıklıkta bir merge kısmı olması söylenebilir. İki çözüm için de ortak olarak k = 5'e kadarki düşüş k'dan kaynaklı olarak recursion derinliğinin azalmasıdır. Birinci çözümde merge kısmındaki karmaşıklık recursion derinliğinin azalması devam etmesine rağmen bir yerdeden sonra daha etkin olmaya başlamıştır. Sonuç olarak ödevde verilen k aralığı için en iyi çözümün birinci çözüm olduğuna karar verilmiştir. Ancak verilecek grafikte de görüleceği üzere k çok fazla arttıkça performans farkı çok büyük boyutlara ulaşıyor. Bu sebeple k büyük olsaydı ikinci çözüm tercih edilecekti.

3- Karmaşıklık Analizi:

Birinci Çözüm Karmaşıklık Analizi:

mergeSortKWay()

@brief divides array into k parts make a call for each part until the base case where n < k and at the end merge the k parts.

@param arr: array which will be sorted.

@param k: it determines how many parts the array will be divided.

@param left: start index of the portion of the array.

@param right: end index of the portion of the array

@return

Base Case Kısmı -> k <= 10 olduğundan ve n değerleri 100'den büyük eşit olduğundan base case karmaşıklığını sabit kabul edebiliriz. Çünkü en fazla 10 boyutundaki kısımlar insertion sort'a sokulacak. (Ayrıca n 10'un katları olduğundan k = 10 iken dizi tam bölündüğünden hiç insertion sort kullanılan base case'e girilmeyecek.)

```
int n = right - left + 1;
if(n == 1) return; //base case when we have one element, it is already sorted
if(n < k){ //base case when n < k, we can divide further so we need to sort it in naive way
    insertionSort(arr, left, right); //used insertion sort to sort the array in the base case
    return;
}
```

Divide kısmı -> n / k genişliğindek k çağrı yapıldığından T(n) = k * T(n / k) + O(n^d) bağıntısı kurabiliriz.

```
//divide
int partSize = n / k, i, j = 0; //calculating the size of each part except the last part by integer division

for(i=0;i<k-1;i++){
    mergeSortKWay(arr, k, left + j, left + j + partSize - 1); //making call for each part with size n / k except last part
    j += partSize;
}
mergeSortKWay(arr, k, left + j, right); //last part may be bigger than n / k so we make the call after for loop
```

merge()

```
//merge
merge(arr, k, left, right, partSize); //merging the k parts
```

@brief creates an auxilary array with size n and selects the min element out of all the k parts in O(k) complexity in each step and

saves it in the aux array. at the end copy the elements to the original array.

@param arr: array some portion of which will be sorted by merging k parts.

@param k: the number of parts which will be merged.

@param left: start index of the portion of the array.

@param right: end index of the portion of the array

@param partSize: the size of each part except the last part (we can find the last part's size by partSize + n % k)

@return

Merge Kısmı -> merge fonksiyonun domine eden kısmı aşağıda verilen kısımdır. Görüleceği üzere O(k*n)'lik bir karmaşıklık var.

```
i = 0;
while(i < n){ //iterates until all the n element is correctly placed in the helper a
    min = INT_MAX; //will be the min element out of all the k parts
    for(j=0;j<k;j++){ //iterating all the parts' current indices
        if(indices[j] <= ends[j] && min > arr[indices[j]]){ //if the current part is
            min = arr[indices[j]];
            index = j; //stores which part's element we took
        }
    }
    indices[index]++; //moving forward in the part where we took the element
    helperArr[i++] = min; //adding the element
}
```

Birinci Çözüm için Total Karmaşıklık -> T(n) = k * T(n / k) + O(n * k) bağıntısı kurulabilir. Buradan Master Theorem kullanarak T(n) = O(n * k * $\log_k$(n)) sonucuna varabiliriz. Logaritmadaki k tabanını silemeyiz çünkü k recursion derinliğini doğrudan etkilediği ve dolayısıyla çalışma süresini etkiliyor. Aynı mantıkla k çarpanını da kaldıramayız çünkü merge kısmından kaynaklı k çarpanından ötürü k arttıkça grafiklerde de görüleceği üzere recursion derinliği azalmasına rağmen k = 5'ten itibaren çalışma süresinde zigzaglı bir artış meydana geliyor.

İkinci Çözüm Karmaşıklık Analizi:

Tek farklı kısım merge kısmı olduğundan oraya odaklanılacak.

merge kısmı -> Görüleceği üzere her iterasyonda extractMin işlemi ve insert işlemi var. İkisi de logk karmaşıklıktadır. Sonuç olarak O(n * logk) karmaşıklık elde ediliyor.

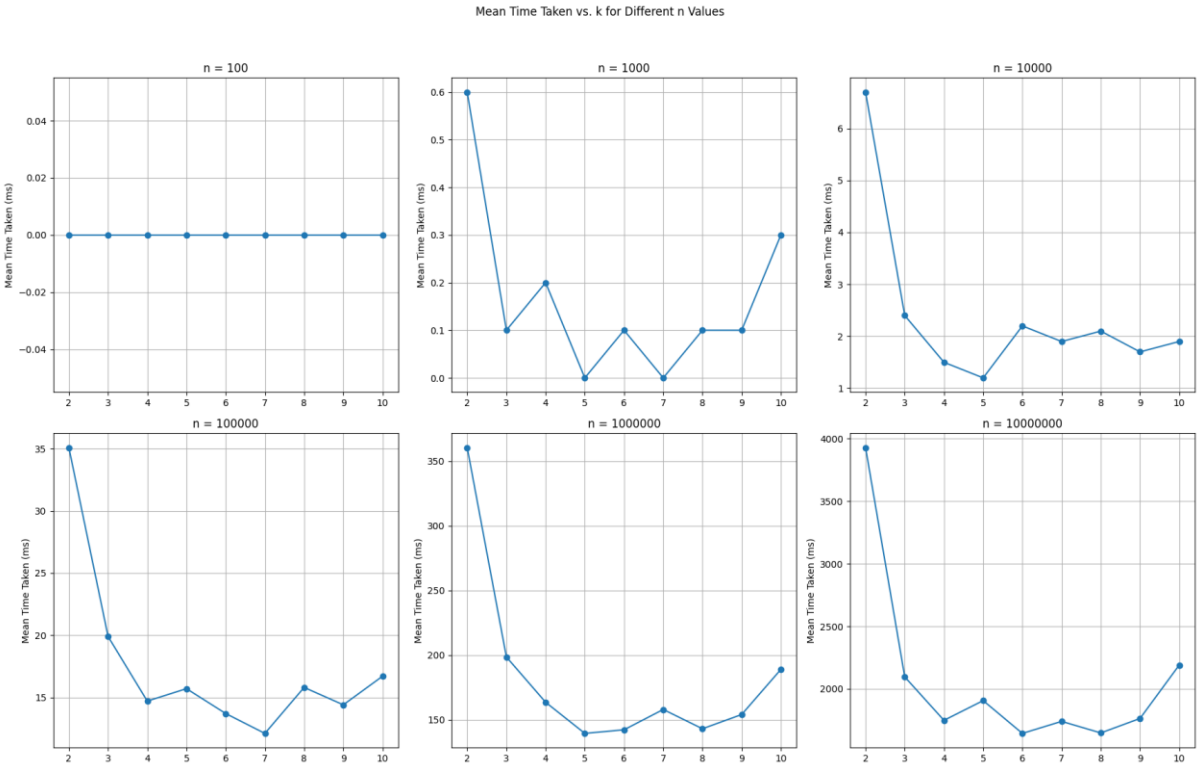
```
i = 0;
while(i < n){
    currMin = extractMin(heap, &heapSize);
    helperArr[i++] = currMin.key;

    if(indices[currMin.index] <= ends[currMin.index]){
        insert(heap, &heapSize, arr[indices[currMin.index]], currMin.index);
        indices[currMin.index]++;
    }
}
```

İkinci Çözüm için Total Karmaşıklık -> T(n) = k * T(n / k) + O(n * logk) bağıntısı kurulabilir. Master Theorem'den yararlanılarak T(n) = O(logk * n * $\log_k$(n)) sonucuna varılabilir. Buradan da ayrıca $\log_k$(n) logn / logk olarak yazılabileceğinden O(nlogn) sonucuna varılabilir. Zaten grafiklerde de görüleceği üzere ikinci çözüm için n sabitken belli bir k değerinden (k = 15) sonra k'dan bağımsız bir çalışma süresi elde ediliyor. Yani grafik sabit bir fonksiyon gibi görülüyor. Belli k değerine kadar alçalan bir grafik elde edilmesi heap kullanmanın overheadinin ancak o değere kadar dengelenmesidir denebilir.

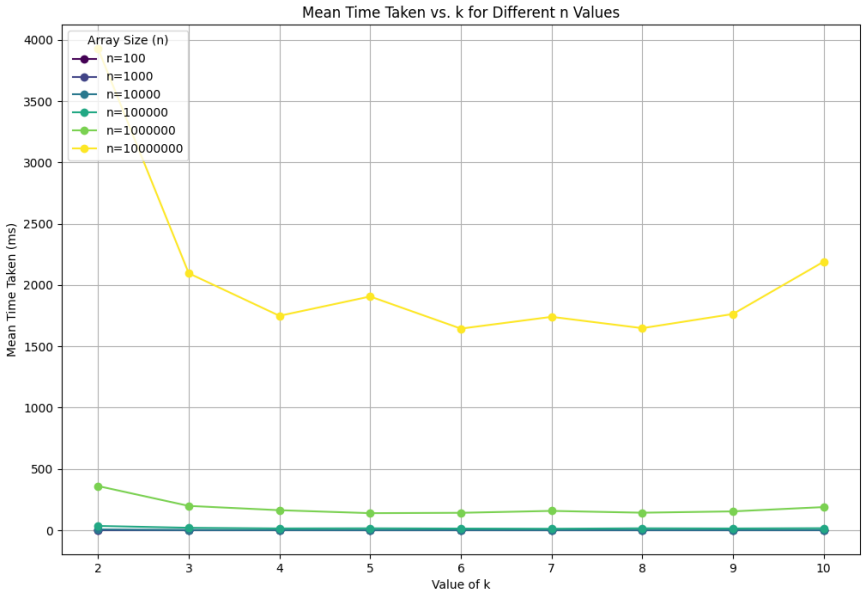
4-Ekran Çıktıları:

Şekil 1- k değerlerine göre her bir n için süre grafikleri

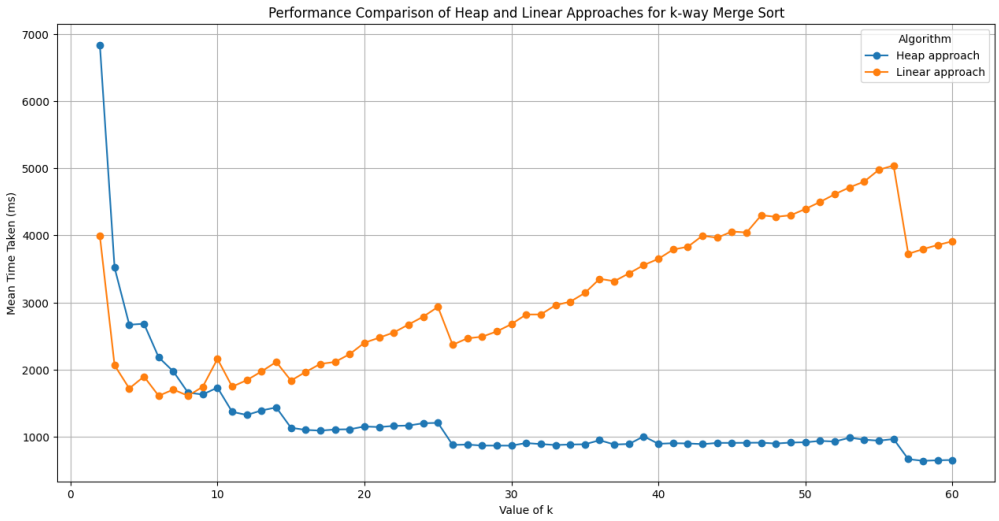


Görüldüğü üzere k arttıkça 5 ve 6'ya kadar düzenli bir artış var çünkü recursion derinliği yarıya hatta yarıdan fazlasına iniyor ve merge kısmındaki k çarpanı çok küçük kalıyor. Ancak sonrasında görüleceği üzere zigzaglı bir yükseliş var. Bu zigzag'ın sebebi olarak k değerine göre insertion sort'un kullanım oranı, yoğunluğu düşünülebilir. Mesela k = 8 iken ve n = 100 iken ilk parçalar son parça hariç 12 uzunluğunda ikinci seviyedeki çağrılarda ise son parça hariç 1 uzunluğundalar yani doğrudan insertion sort'a girmeden parçaların çoğunluğu O(1) zamanda return olacak ve merge edilecekler. Ancak k = 7 iken ilk seviyedeki çağrılarda parçalar sonuncu hariç 14 uzunluğunda, ikinci seviyede ise herbiri 2'şer uzunlukta olacak yani insetion sort maliyeti yoğunlaşacak. Bu olmasaydı daha düzgün bir artış görebilirdik diye düşünüyorum. Zaten test sayısı artınca

Şekil 2- k değerlerine göre n değerlerinin birlikte olduğu süre grafiği



Şekil 3- Birinci ve ikinci çözümün karşılaştırılması



Görüleceği üzere birinci çözüm güzel bir başlangıç yapıyor ancak sonrasında süre artmaya başlıyor. Bunda daha öncesinde denildiği gibi merge kısmındaki k çarpanının etkisi yüksek. İkinci çözüm ise görüldüğü üzere belli bir k değerinden sonra daha düzenli sabit bir grafik elde ediliyor. Başlangıçtaki yüksek başlangıç ve düşüş recursion derinliğindeki ani düşüşten kaynaklı diye düşünüyorum. K = 2 için çok büyük fark oluşması ise heap overhead'inin

etkisinin çok büyük olması olarak düşünülebilir. Ayrıca fark edileceği üzere birinci çözümde fark edilecek zigzagların ikinci çözümde de bulunduğu ancak süre daha düzenli bir aşamaya geçtiğinden çok farkedilmediği belli oluyor. Ben bunun recursion derinliğinin k ikinin tam katı olduğundaki ani değişiminden kaynaklı olduğunu ayrıca farklı k değerlerinin diziyi farklı böldüklerinden insertion maliyetini artırıp azaltmalarından kaynaklı olduğunu düşünüyorum.