HEAP SORT ALGORİTMASI ANALİZİ

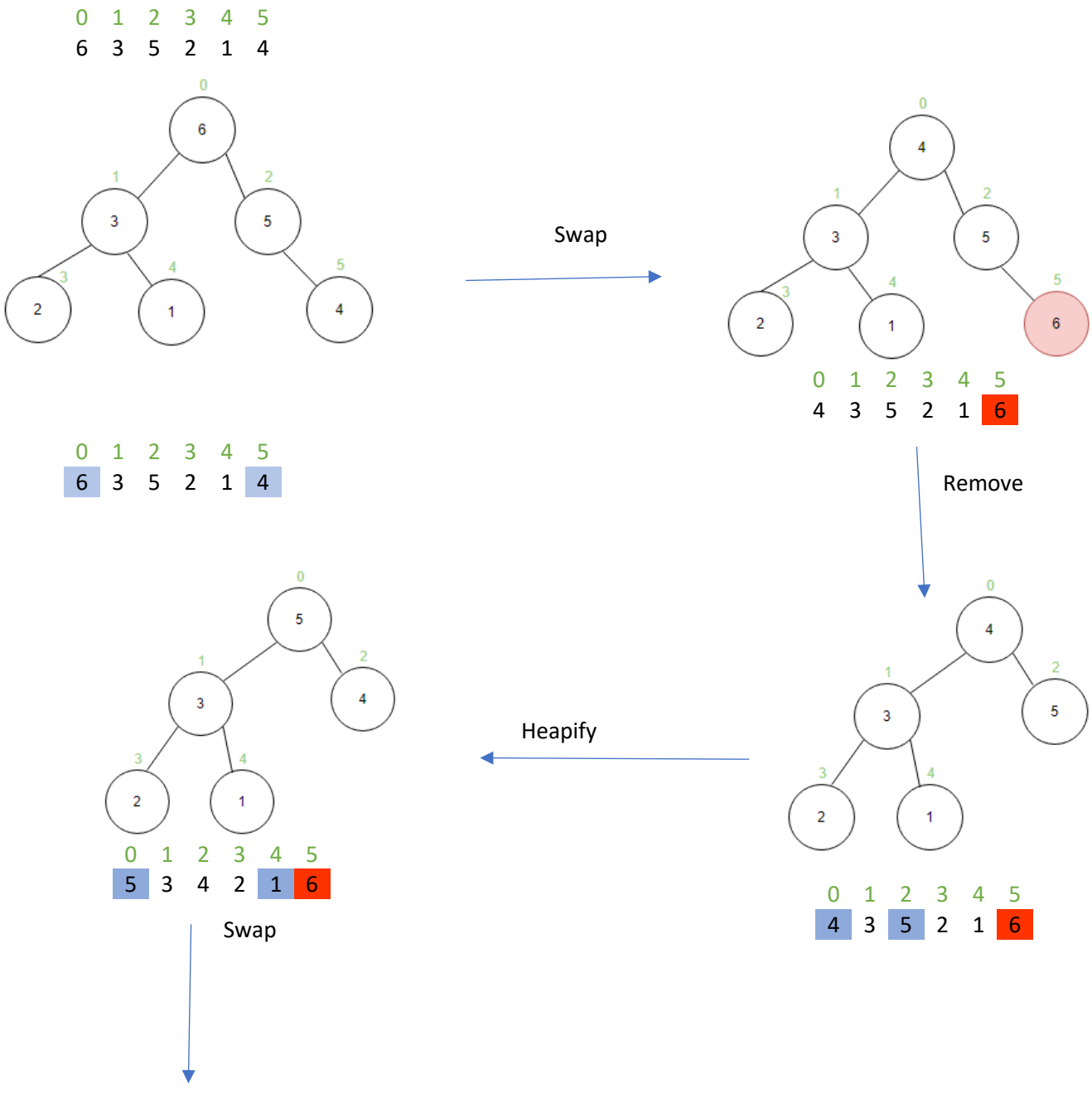
Verileri bellekte böl ve fethet yöntemi kullanarak sıralı tutmak için geliştirilmiş algoritmalardan biridir. Yığılmış sıralama, arka planda bir yığın oluşturur ve ağacın tepesindeki sayıyı alarak sıralar.

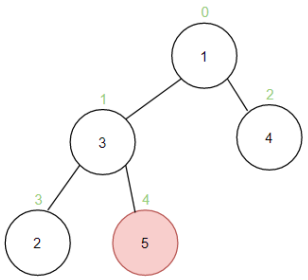
Linux çekirdeği gibi güvenlik ve gömülü sistemler, Heapsort'un çalışma zamanı üst sınırı ve ikincil depolamada sabit bir üst sınır nedeniyle heapsort kullanır.

Tam bir ikili ağaçtan başlayarak, yığının yaprak olmayan tüm öğelerinde heapify adlı bir işlevi çalıştırarak onu Max-Heap olacak şekilde değiştirir. (swap-remove-heapify)

Örnek Problem:

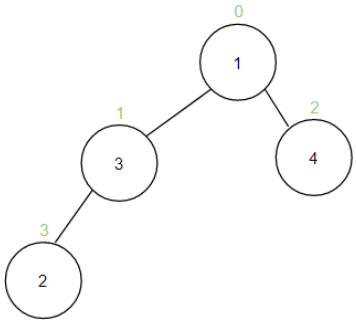
Aşağıda sıralanmamış bir dizi verilmiştir. Heap sort algoritması kullanılarak sıralanıyınız.





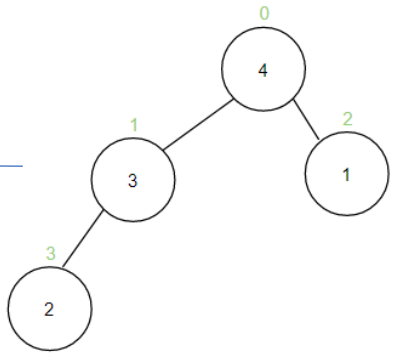
0	1	2	3	4	5
1	3	4	2	5	6

remove



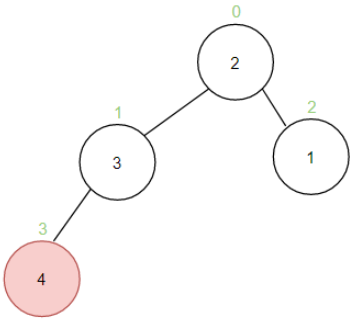
0	1	2	3	4	5
1	3	4	2	5	6

Heapify



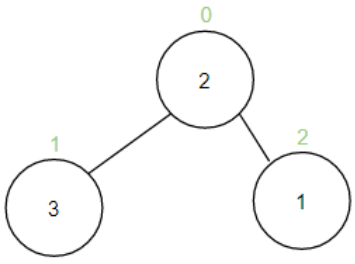
0	1	2	3	4	5
4	3	1	2	5	6

Swap



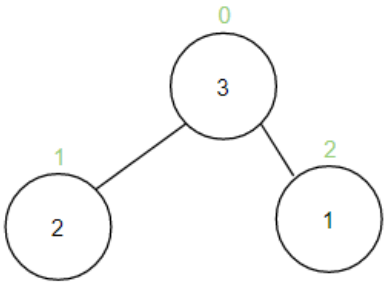
0	1	2	3	4	5
2	3	1	4	5	6

Remove



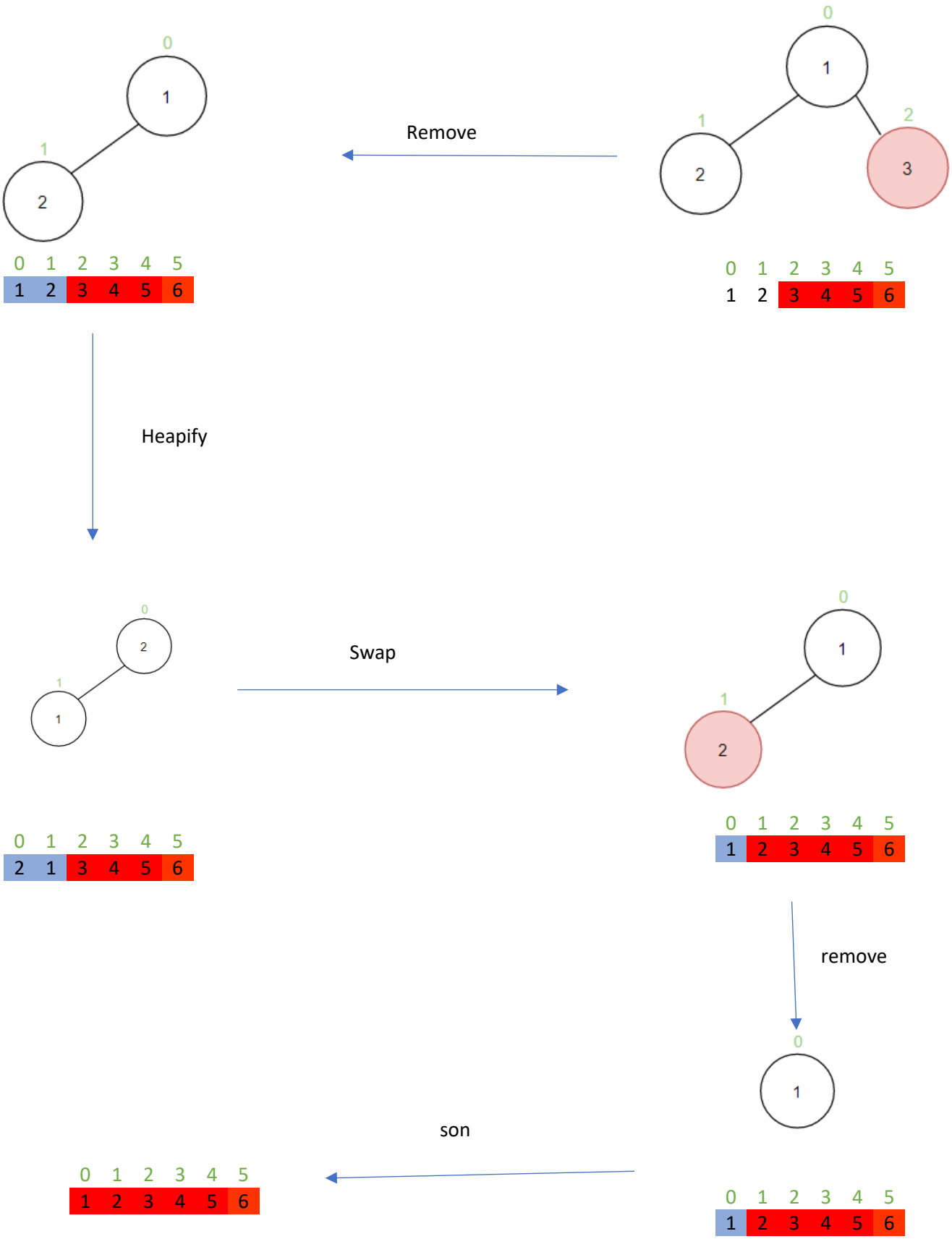
0	1	2	3	4	5
2	3	1	4	5	6

Heapify

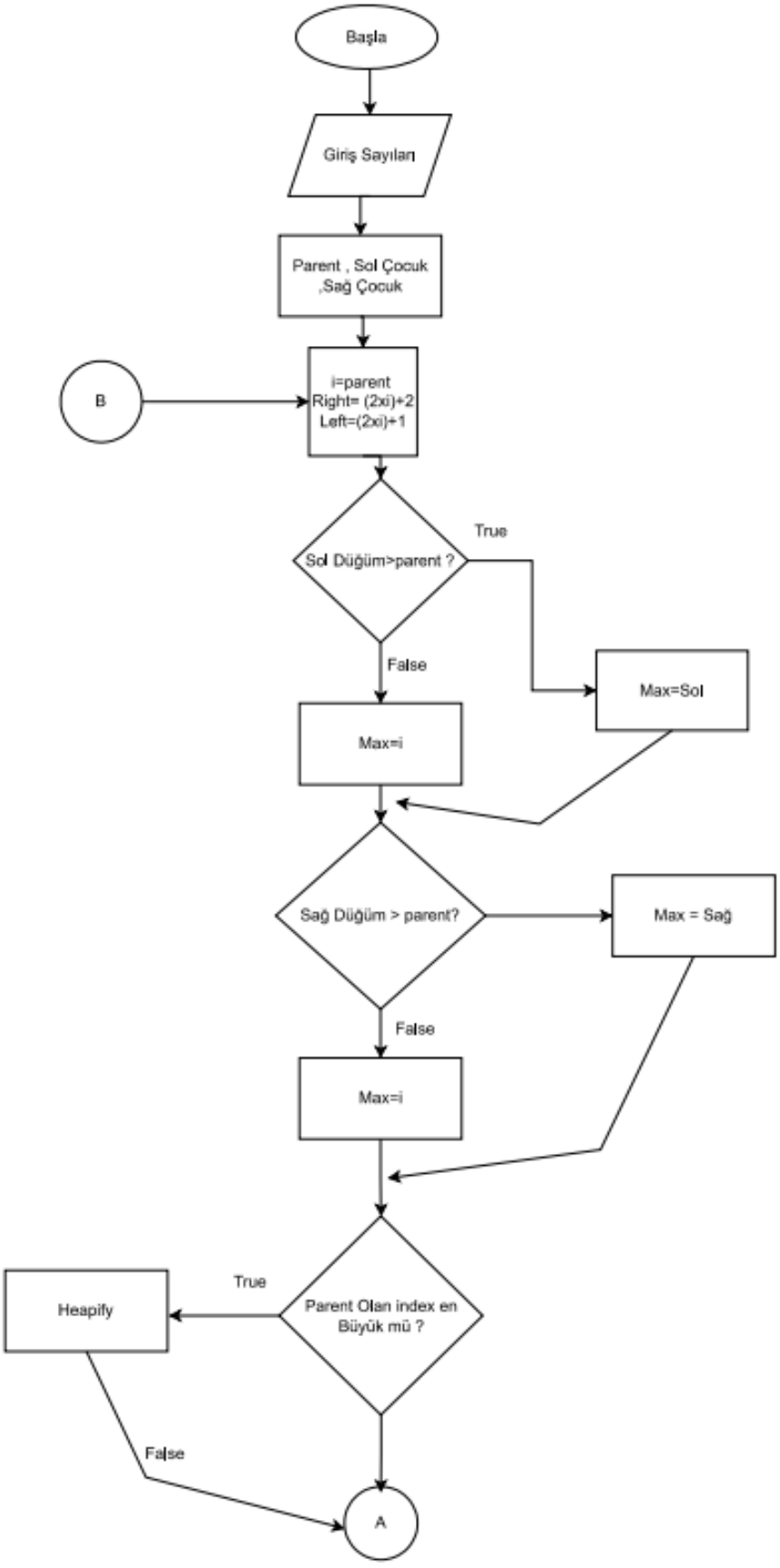


0	1	2	3	4	5
3	2	1	4	5	6

Swap


0
1
1
2
0 1 2 3 4 5
1 2 3 4 5 6
Remove
0
1
1
2
2
3
0 1 2 3 4 5
1 2 3 4 5 6
Heapify
0
2
1
1
0 1 2 3 4 5
2 1 3 4 5 6
Swap
0
1
1
2
0 1 2 3 4 5
1 2 3 4 5 6
remove
0
1
son
0 1 2 3 4 5
1 2 3 4 5 6
0 1 2 3 4 5
1 2 3 4 5 6

Çözümün Akış Diyagramı:



Heap Sort C++ Kod:

```
// Heap Sort
#include <iostream>
using namespace std;

void heapify(int arr[], int n, int i) {
//Kök, sol alt ve sağ alt öğe arasında en büyüğünü bulun
int largest = i;
int left = 2 * i + 1;
int right = 2 * i + 2;

if (left < n && arr[left] > arr[largest])
largest = left;

if (right < n && arr[right] > arr[largest])
largest = right;

//Kök en büyük değilse takas edin ve yığınlamaya devam edin
if (largest != i) {
swap(arr[i], arr[largest]);
heapify(arr, n, largest);
}
}


// yığın sıralama yapmak için fonksiyon
void heapSort(int arr[], int n) {
// Max heap oluştur
for (int i = n / 2 - 1; i >= 0; i--)
heapify(arr, n, i);

// Heap sort
for (int i = n - 1; i >= 0; i--) {
swap(arr[0], arr[i]);

// En yüksek öğeyi kökte tekrar elde etmek için kök öğeyi yığınla
heapify(arr, i, 0);
}
}


//diziyi yazdır
void printArray(int arr[], int n) {
for (int i = 0; i < n; ++i)
cout << arr[i] << " ";
cout << "\n";
}

//Sürücü fonk.
int main() {
int arr[] = {6, 3, 5, 2, 1, 4};
int n = sizeof(arr) / sizeof(arr[0]);
heapSort(arr, n);

cout << "Sorted array is \n";
printArray(arr, n);
}
```

Heap Sort Algoritma Analizi:

İkili bir ağaç kullandığımız için, yığının alt kısmı maksimum sayıda düğüm içerir. Bir seviye yukarı çıktıkça, düğüm sayısı yarı yarıya azalır. 'N' sayıda düğüm olduğu göz önüne alındığında, en alt seviyeden başlayan düğüm sayısı :

n/2
n/4
n/8
.
.
.

Devam eder

1 = N / 2^x
2^x = N
log(2^x) = log2 N
x * log2(2) = log2 N
x * 1 = log2 N

Yeni bir düğüm eklemenin karmaşıklığı
Bu nedenle yığını yaparken yığına yeni bir değer eklediğimizde, atmamız gereken maksimum adım sayısı O(log(n)) olarak çıkıyor.

İkili bir ağaç kullandığımız için, bu yapının maksimum yüksekliğinin her zaman O(log(n)) olur. Yığına yeni bir değer eklediğimizde, max heap özelliğini korumak için onu daha büyük bir değerle değiştiririz. Bu tür takasların sayısı O(log(n)) olacaktır. Bu nedenle, maksimum yığın oluştururken yeni bir değer eklemek O(log(n)) olur.

Yığından maksimum değerli düğümü çıkarmanın karmaşıklığı
Benzer şekilde, listenin sonuna eklemek için maksimum değerli düğümü yığından çıkardığımızda, gereken maksimum adım sayısı da O(log(n)) olacaktır. Maksimum değerli düğümü, en alt düzeye inene kadar değiştirdiğimiz için, atmamız gereken maksimum adım sayısı, yeni bir düğüm eklerken olduğu gibi, O(log(n)) ile aynıdır.

Bu nedenle, fonksiyonun toplam zaman karmaşıklığı heapify O(log(n)) olur.(void heapify)

Bir yığın oluşturmanın karmaşıklığı:

(n/2 * 0) + (n/4 * 1) + (n/8 * 2) + (n/16 * 3) + ... =n/2
Bu yüzden bunun karmaşıklığı O(n) olur.

Toplam zaman karmaşıklığı:

Son işlevinde, bir yığın oluşturmak için bir kez çalışan ve çalışma süresi O(n) olan işlevini heapsort kullanırız . //max heap oluştur, Sonra bir for-loop kullanarak, heapify yığına bir düğüm eklediğimizde veya çıkardığımızda max-heap özelliğini korumak için for her düğümü çağırırız. 'n' sayıda düğüm olduğundan, algoritmanın toplam çalışma süresi O(n(log(n)) olur ve heapify işlevi her düğüm için kullanırız.

Matematiksel olarak:

- Bir düğümün ilk kaldırılması log(n) zaman alır
- İkinci kaldırma log(n-1) zaman alır
- Üçüncü kaldırma log(n-2) zamanını alır
- ve böylece log(1) zamanını alacak olan son düğüme kadar

log(n) + log(n-1) + log(n-2) + ...log(1) log(x) + log(y)

= log(x * y olarak)

=log($n*(n-1)*(n-2)*\ldots*2*1$)

=log(n!)

Daha fazla sadeleştirme ile,(Stirling Yöntemi ile Faktöryel değerlerin tahmini)

log(n!)

=$n*$log(n) olur.)$-n+O(\log(n))$

En yüksek sıralı terim dikkate alındığında, toplam çalışma süresi O(n(log(n)) olur.

Yığın Sıralamanın En Kötü Durum Zaman Karmaşıklığı

Yığın sıralama için en kötü durum, listedeki tüm öğeler farklı olduğunda ortaya çıkabilir. Bu nedenle, max-heapifybir öğeyi her kaldırdığımızda aramamız gerekir. Böyle bir durumda, 'n' sayıda düğüm olduğu düşünülürse

- Her öğeyi kaldırmak için takas sayısı log(n) olacaktır, çünkü bu, yığının maksimum yüksekliğidir.
- Bunu her düğüm için yaptığımızı düşünürsek, toplam hamle sayısı n * (log(n)) olur.

Bu nedenle, en kötü durumda çalışma zamanı O(n(log(n)) olacaktır.

Yığın Sıralamanın En İyi Vaka Zaman Karmaşıklığı

Yığın sıralama için en iyi durum, sıralanacak listedeki tüm öğeler aynı olduğunda ortaya çıkar. Böyle bir durumda, 'n' düğüm sayısı için:

- Her bir düğümü öbekten kaldırmak, yalnızca sabit bir çalışma zamanı (O(1)) alacaktır. Tüm öğeler aynı olduğu için herhangi bir düğümü indirmeye veya maksimum değerli düğümü yukarı getirmeye gerek kalmayacaktır.
- Bunu her düğümde yaptığımız için toplam hamle sayısı n * O(1) olacaktır.

Bu nedenle, en iyi durumda çalışma zamanı O(n) olacaktır.

Yığın Sıralamanın Uzay Karmaşıklığı

Yığın sıralama yerinde tasarlanmış bir sıralama algoritması olduğundan, alan gereksinimi sabittir ve bu nedenle O(1). Bunun nedeni, herhangi bir giriş durumunda

- Bir yığın yapısı kullanarak tüm liste öğelerini yerinde düzenliyoruz
- Max düğümünü max-heap'ten çıkardıktan sonra kaldırılan öğeyi aynı listenin sonuna koyuyoruz.

Bu nedenle, bu algoritmayı uygularken fazladan boşluk kullanmıyoruz. Bu, algoritmaya O(1) uzay karmaşıklığını verir.

Özet olarak, heapsort şunları içerir:

- Böl ve fethet yöntemini kullanır
- En iyi durum zaman karmaşıklığı O(n) [tüm öğeler sıralanmışsa]
- O(n(log(n))'nin en kötü durum zaman karmaşıklığı [listedeki öğeler sıralanmamışsa]
- O(1) uzay karmaşıklığı

SHELL SORT ALGORİTMASI ANALİZİ

Verileri bellekte böl ve fethet yöntemi kullanarak sıralı tutmak için geliştirilmiş algoritmalardan biridir.

Önce birbirinden uzaktaki öğeleri sıralar ve sıralanacak öğeler arasındaki aralığı art arda azaltır.

Öğeler arasındaki aralık, kullanılan sıraya göre azaltılır. Ve öğeler karşılaştırılarak sıralanır.

Örnek Problem:

Aşağıda sıralanmamış bir dizi verilmiştir. Shell sort algoritması kullanılarak sıralayınız.

0	1	2	3	4	5	6	7	8	9	10	11
5	6	12	7	4	1	2	8	9	10	3	11

K=12 =temp

K=12/2

K=6

0	1	2	3	4	5	6	7	8	9	10	11
5	6	12	7	4	1	2	8	9	10	3	11

0	1	2	3	4	5	6	7	8	9	10	11
2	6	12	7	4	1	5	8	9	10	3	11

0	1	2	3	4	5	6	7	8	9	10	11
2	6	12	7	4	1	5	8	9	10	3	11

0	1	2	3	4	5	6	7	8	9	10	11
2	6	9	7	4	1	5	8	12	10	3	11

0	1	2	3	4	5	6	7	8	9	10	11
2	6	9	7	4	1	5	8	12	10	3	11

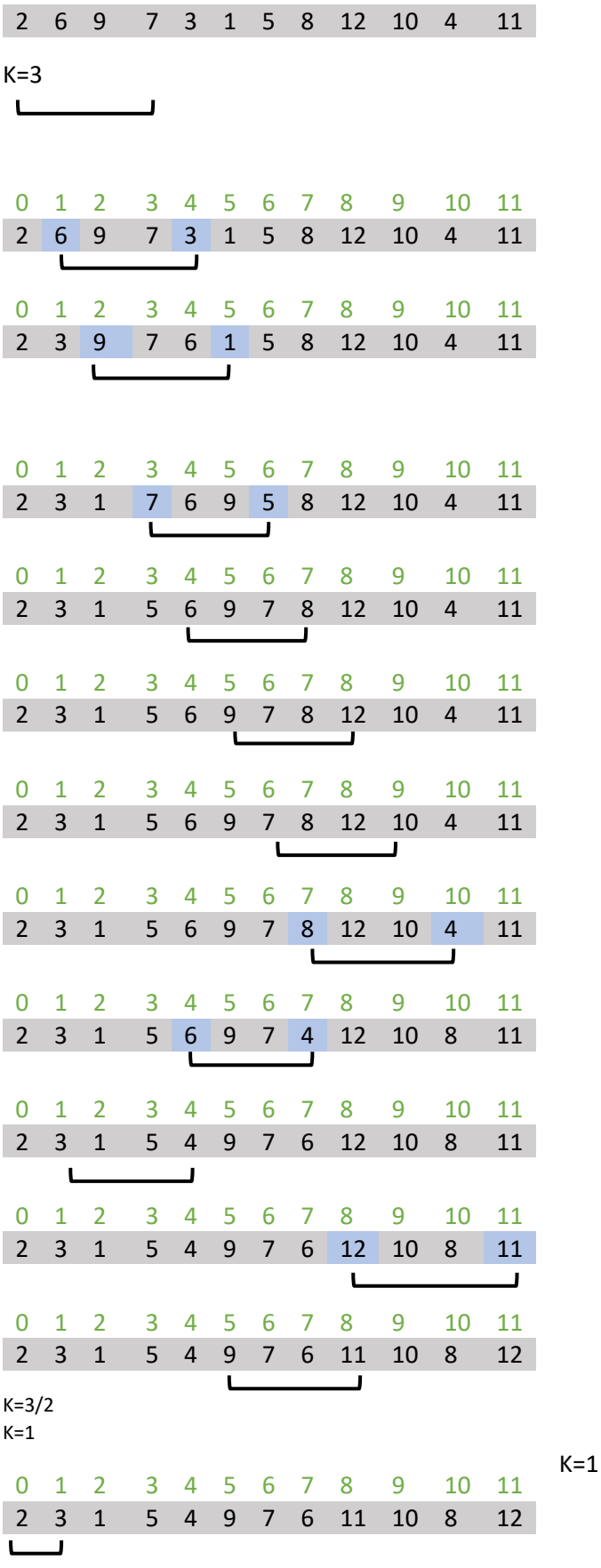
0	1	2	3	4	5	6	7	8	9	10	11
2	6	9	7	4	1	5	8	12	10	3	11

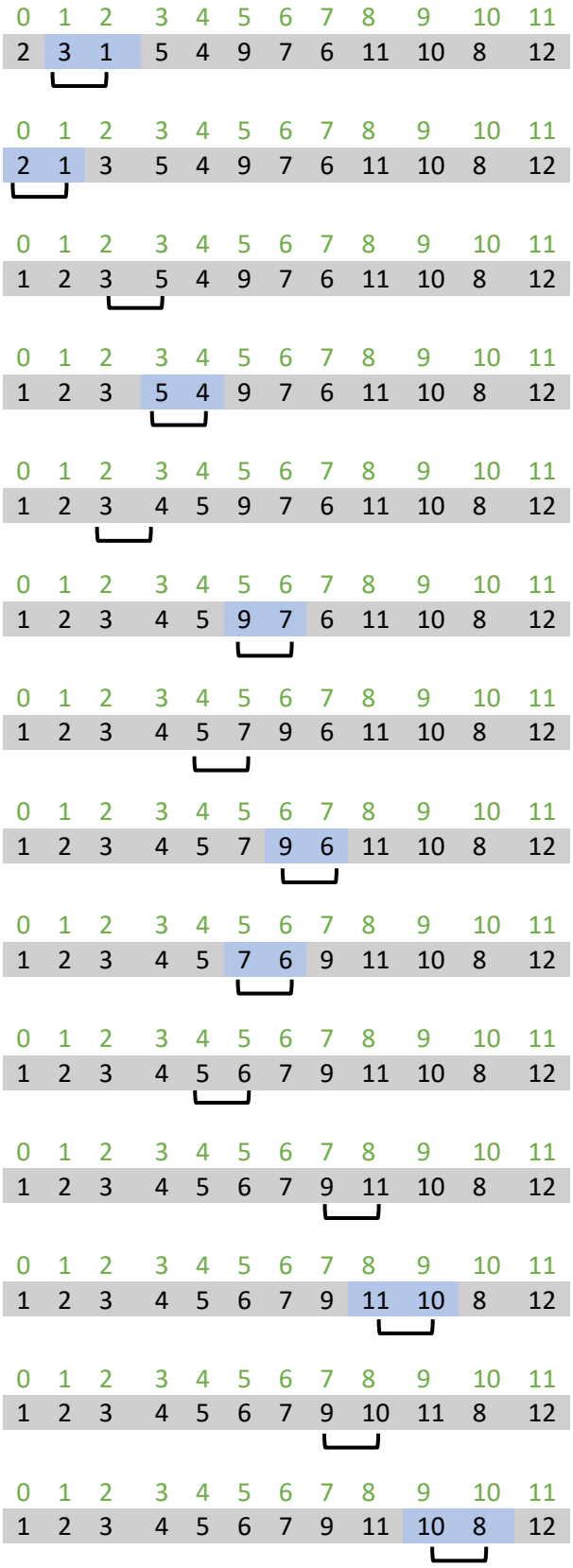
0	1	2	3	4	5	6	7	8	9	10	11
2	6	9	7	3	1	5	8	12	10	4	11

K=K/2

K=6/2

0	1	2	3	4	5	6	7	8	9	10	11


2 6 9 7 3 1 5 8 12 10 4 11
K=3
0 1 2 3 4 5 6 7 8 9 10 11
2 6 9 7 3 1 5 8 12 10 4 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 9 7 6 1 5 8 12 10 4 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 7 6 9 5 8 12 10 4 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 6 9 7 8 12 10 4 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 6 9 7 8 12 10 4 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 6 9 7 8 12 10 4 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 6 9 7 8 12 10 4 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 6 9 7 4 12 10 8 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 4 9 7 6 12 10 8 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 4 9 7 6 12 10 8 11
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 4 9 7 6 11 10 8 12
K=3/2
K=1
K=1
0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 4 9 7 6 11 10 8 12


0 1 2 3 4 5 6 7 8 9 10 11
2 3 1 5 4 9 7 6 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
2 1 3 5 4 9 7 6 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 5 4 9 7 6 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 5 4 9 7 6 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 9 7 6 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 9 7 6 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 7 9 6 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 7 9 6 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 7 6 9 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 6 7 9 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 6 7 9 11 10 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 6 7 9 10 11 8 12
0 1 2 3 4 5 6 7 8 9 10 11
1 2 3 4 5 6 7 9 11 10 8 12

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	9	11	8	10	12

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	9	8	11	10	12

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	11	10	12

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	11	10	12

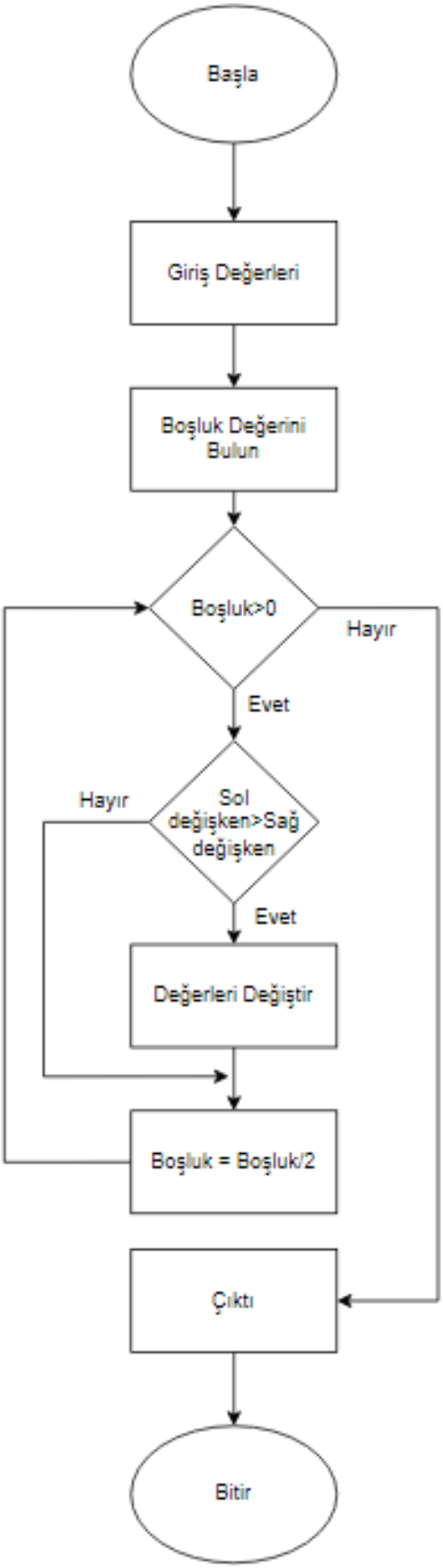
0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

Sıralanmış dizi:

0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12

Çözümün Akış Diyagramı:



Shell Sort C++ Kod:

```
45. //bedirhanbuyukozShellSort
46. #include <iostream>
47. using namespace std;
48. 
49. // Shell sort
50. void shellSort(int array[], int n) {
51.   //Öğeleri her n / 2, n / 4, n / 8, ... aralıklarla yeniden düzenleyin
52.   for (int gap = n / 2; gap > 0; gap /= 2) {
53.     for (int i = gap; i < n; i += 1) {
54.       int temp = array[i];
55.       int j;
56.       for (j = i; j >= gap && array[j - gap] > temp; j -= gap) {
57.         array[j] = array[j - gap];
58.       }
59.       array[j] = temp;
60.     }
61.   }
62. }
63. 
64. // Diziyi Yazd.fonk.
65. void printArray(int array[], int size) {
66.   int i;
67.   for (i = 0; i < size; i++)
68.     cout << array[i] << " ";
69.   cout << endl;
70. }
71. 
72. // Sürücü Kod
73. int main() {
74.   int data[] = {5, 6, 12, 7, 4, 1, 2, 8, 9, 10, 3, 11};
75.   int size = sizeof(data) / sizeof(data[0]);
76.   shellSort(data, size);
77.   cout << "Sıralanmış Dizi: \n";
78.   printArray(data, size);
79. }
```

Shell Sort Algoritma Analizi:

En iyi durum ∈ O(nlogn):

En iyi durum, dizinin zaten sıralanmış olmasıdır. Ardından, artışa dayalı sıralamalarının her biri için karşılaştırma sayısı, dizinin uzunluğudur. Bu nedenle şunları belirleyebiliriz:

  n , bir sıralama için
+ 3 * n/3 , öğeleriyle 3 elemanlı sıralama için
+ 5 * n/5 , ile 5 elemanlı sıralama için
+ 15 * n/15 , 15 elemanlı sıralama için
...
Her terim n'dir . Terim sayısı k değeridir .
2^k - 1 < n

Yani k < log(n+1) , en iyi durumda sıralama süresinin n * log(n+1) = O(n*log(n)) olucaktır.

En kötü durum ∈ O(n^2):

Çok kötü bir senaryoda ,her sıralama ikinci dereceden zaman olacaktır.

  n^2 , 1 sıralama için
+ 3 * (n/3)^2 , 3 elemanlı sıralama için
+ 5 * (n/5)^2 , 5 elemanlı sıralama için
+ 15*(n/15)^2, 15 elemanlı sıralama için

Ve böylece, karşılaştırma sayısının şu şekilde sınırlandırıldığını görebiliriz:
 n^2 * (1 + 1/3 + 1/5 + 1/15 + 1/31 + ...) < n^2 * (1 + 1/2 + 1/4 + 1/6 + 1/16 + ...)
= n^2 en kötü durum
Son adım, geometrik serinin toplamını kullanır

En kötü durum ∈ Ω(n^2):

Tüm çift konumlu öğelerin medyandan daha büyük olduğu dizidir. Son 1'lik artışa ulaşana kadar tek ve çift öğeler karşılaştırılmaz. Son yineleme için gereken karşılaştırma/değişim sayısı Ω(n^2)'dir

Özet olarak, Shell Sort şunları içerir:

- Böl ve fethet yöntemini kullanır
- En iyi durum zaman karmaşıklığı O(nlog n) [tüm öğeler sıralanmışsa]
- O(n^2)'nin en kötü durum zaman karmaşıklığı [listedeki öğeler sıralanmamışsa]
- O(1) uzay karmaşıklığı

HEAP SORT vs SHELL SORT ANALİZİ KARŞILAŞTIRMA

Heapsort şunları içerir:

- BÖL VE FETHET YÖNTEMINI KULLANIR **- [SHELL SORTLA AYNI]**
- EN IYI DURUM ZAMAN KARMAŞIKLIĞI O(N) (TÜM ÖĞELER SIRALANMIŞSA **-[SHELL SORT'TAN(NLOG(N)) DAHA İYİ]**

Bir yığın oluşturmanın karmaşıklığı:

(n/2 * 0) + (n/4 * 1) + (n/8 * 2) + (n/16 * 3) + ... =n/2
Bu yüzden bunun karmaşıklığı O(n) olur.

Yığın sıralama için en iyi durum, sıralanacak listedeki tüm öğeler aynı olduğunda ortaya çıkar. Böyle bir durumda, 'n' düğüm sayısı için:

- Her bir düğümü öbekten kaldırmak, yalnızca sabit bir çalışma zamanı (O(1)) alacaktır. Tüm öğeler aynı olduğu için herhangi bir düğümü indirmeye veya maksimum değerli düğümü yukarı getirmeye gerek kalmayacaktır.
- Bunu her düğümde yaptığımız için toplam hamle sayısı n * O(1) olacaktır.

Bu nedenle, en iyi durumda çalışma zamanı O(n) olacaktır.

- O(N(LOG(N))'NIN EN KÖTÜ DURUM ZAMAN KARMAŞIKLIĞI (LISTEDEKI ÖĞELER SIRALANMAMIŞSA)- **[SHELL SORT'TAN (N^2) DAHA İYİ]**

Yığın sıralama için en kötü durum, listedeki tüm öğeler farklı olduğunda ortaya çıkabilir. Bu nedenle, max-heapifybir öğeyi her kaldırdığımızda aramamız gerekir. Böyle bir durumda, 'n' sayıda düğüm olduğu düşünülürse

- Her öğeyi kaldırmak için takas sayısı log(n) olacaktır, çünkü bu, yığının maksimum yüksekliğidir.
- Bunu her düğüm için yaptığımızı düşünürsek, toplam hamle sayısı n * (log(n)) olur.

Bu nedenle, en kötü durumda çalışma zamanı O(n(log(n)) olacaktır.

- O(1) UZAY KARMAŞIKLIĞI-**[HEAP SORT İLE AYNI]**

Shell Sort şunları içerir:

- BÖL VE FETHET YÖNTEMINI KULLANIR**-[HEAP SORT İLE AYNI YÖNTEM]**
- EN IYI DURUM ZAMAN KARMAŞIKLIĞI O(NLOG N)- (TÜM ÖĞELER SIRALANMIŞSA)-**[HEAP SORT(O(N)) DAHA İYİ]**

En iyi durum, dizinin zaten sıralanmış olmasıdır. Ardından, artışa dayalı sıralamalarının her biri için karşılaştırma sayısı, dizinin uzunluğudur. Bu nedenle şunları belirleyebiliriz:

n , bir sıralama için
+ 3 * n/3 , öğeleriyle 3 elemanlı sıralama için
+ 5 * n/5 , ile 5 elemanlı sıralama için
+ 15 * n/15 , 15 elemanlı sıralama için
...
Her terim n'dir . Terim sayısı k değeridir .
2^k - 1 < n

Yani k < log(n+1) , en iyi durumda sıralama süresinin n * log(n+1) = O(n*log(n)) olucaktır.

- O(N^2)'NIN EN KÖTÜ DURUM ZAMAN KARMAŞIKLIĞI (LISTEDEKI ÖĞELER SIRALANMAMIŞSA) - [**HEAP SORT O(NLOG(N)) DAHA İYİ]**

Çok kötü bir senaryoda ,her sıralama ikinci dereceden zaman olacaktır.

n^2 , 1 sıralama için
+ 3 * (n/3)^2 , 3 elemanlı sıralama için
+ 5 * (n/5)^2 , 5 elemanlı sıralama için
+ 15*(n/15)^2, 15 elemanlı sıralama için

Ve böylece, karşılaştırma sayısının şu şekilde sınırlandırıldığını görebiliriz:
n^2 * (1 + 1/3 + 1/5 + 1/15 + 1/31 + ...) < n^2 * (1 + 1/2 + 1/4 + 1/6 + 1/16 + ...)
= n^2 en kötü durum
Son adım, geometrik serinin toplamını kullanır

- O(1) UZAY KARMAŞIKLIĞI **- [HEAP SORT İLE AYNI]**