-Sorting-

Verilerin sıralanmış biçimde olması birçok kodda işimizi kolaylaştırır, hatta bazılarında olmazsa olmazdır. Tabi ki öncelikle bu sıralama işlemini neye göre yapacağımızı belirlemiş olmamız gerekir. Örneğin elimizdeki 10 çocuğu sınav notlarına göre sıralayabiliriz. Veya boy uzunluklarına göre sıralayıp, eşitlik durumunda kilolarını göz önünde bulundurmamız gerekebilir. Bu gibi durumlar için öncelikle '<' ve '>' operatörlerimizin doğru çalıştığından emin olmalıyız. Zaman zaman kendi struct'ımızı yazıp, kendi operatörlerimizi tanımlamamız gerekli olabilir. Bazen ise tek ihtiyacımız olan sayılarımızı küçükten büyüğe basit bir şekilde dizmektir. Küçüklük-Büyüklük ilişkisini aşağıdaki örneklerdeki gibi doğru bir şekilde oturttuktan sonra sıralama algoritmalarına geçebiliriz.

```
int ar[100];

sort(ar+1,ar+1+N);      // Küçükten büyüğe sıralar.
reverse(ar+1,ar+1+N);   // Diziyi ters çevirerek büyükten küçüğe
                        // sıralanmış olarak elde edebiliriz.
```

```
pair <int,int> ar[100];

sort(ar+1,ar+1+N); // İlk N elemanı küçükten büyüğe sıralar.
                   // First'lerine bakar, eşitse second'larına.
```

```
struct veled{

    int boy,kilo;

    friend bool operator < ( const veled &a , const veled &b){

        if(a.kilo < b.kilo) return true;
        if(b.kilo < a.kilo) return false;
        if(a.boy < b.boy) return true;
        return false;
    }
};

// Bu şekilde tanımlanırsa kilolarına göre sıralar. Kiloları eşit ise boylarına bakar.
```

Bubble Sort — Time: $O(N^2)$ Memory: O(N)

Bubble sort, yan yana bulunan iki elemandan soldaki sağdakinden büyük ise bu ikisini yer değiştirme prensibine dayanır. Anlaması ve kodlaması kolaydır ama yavaş çalışır. Tüm işlemler orijinal dizinin kendisi üzerinde gerçekleştiği için hafızadan az yer alır. Yavaş çalıştığı için ileri düzey problemlerde bir işe yaramaz.

```
for(int i=1;i<=N;i++)
    for(int j=1;j<N;j++)
        if(ar[j]>ar[j+1]){

            int tmp=ar[j];
            ar[j]=ar[j+1];
            ar[j+1]=tmp;
        }
```

İspat için: **cs.indiana.edu/classes/p415/post/sorting_tutorial_2.pdf**

Insertion Sort — Time: $O(N^2)$ Memory: O(N)

Bu algoritmada, her bir adımda elimize gelen yeni elemanı sıralanmış olan dizimizin içindeki ait olduğu yere yerleştireceğiz. Önce 1 numaralı eleman, sonra ikinci sıradaki, sonra üç... Her bir adımdan sonra, sıralanmış olan dizimiz bir birim daha büyümüş olacak. N numaralı elemanı da eklediğimizde dizinin tamamını sıralamış olacağız. Örnek olarak bir adımı tarif edecek olursak, 10. adımı gerçekleştiriyoruz diyelim, o anda ilk 9 sayı kendi arasında sıralanmış durumda olacaktır. 10. sayıyı, ilk dokuzunun arasındaki yerine koyup sıralanmış olan dizimizi 10.indise kadar genişletmiş olacağız. Sonra 11. sayımızdan devam edeceğiz. Kısaca X'inci adımda, ilk X-1 sayı sıralanmış durumda olacaktır. Adım adım görüntülenmiş bir girdi örneği ve algoritmanın C++ kodu aşağıda.

```
{18,  6,  9,  1,  4, 15, 12,  5,  6,  7, 11}
{ 6, 18,  9,  1,  4, 15, 12,  5,  6,  7, 11}
{ 6,  9, 18,  1,  4, 15, 12,  5,  6,  7, 11}
{ 1,  6,  9, 18,  4, 15, 12,  5,  6,  7, 11}
{ 1,  4,  6,  9, 18, 15, 12,  5,  6,  7, 11}
{ 1,  4,  6,  9, 15, 18, 12,  5,  6,  7, 11}
{ 1,  4,  6,  9, 12, 15, 18,  5,  6,  7, 11}
{ 1,  4,  5,  6,  9, 12, 15, 18,  6,  7, 11}
{ 1,  4,  5,  6,  6,  9, 12, 15, 18,  7, 11}
{ 1,  4,  5,  6,  6,  7,  9, 12, 15, 18, 11}
{ 1,  4,  5,  6,  6,  7,  9, 11, 12, 15, 18}
```

```
for (int i = 0; i <= data.Length; i++) {
   int j = i;
   while (j > 0 && data[i] < data[j - 1])
      j--;
   int tmp = data[i];
   for (int k = i; k > j; k--)
      data[k] = data[k - 1];
   data[j] = tmp;
}
```

İspat için: **hg.schaathun.net/DisMath/Part3Induction/proof.pdf**

Merge Sort Time: O(N x logN) Memory: O(N x logN)

Elimizde iki tane sıralanmış dizi varsa bunları sıralanmış tek bir dizi haline nasıl getirebiliriz? İki tane gösterge alırız, dizilerimizin ilk elemanlarını gösterirler bunlar en başta. Daha sonra her bir adımda hangi göstergenin gösterdiği eleman daha küçükse onu yeni bir diziye yazarız ve o göstergeyi bir sağa kaydırırız. Böylece devam ettiğimizde, sıralanmış iki dizimiz birleşir, sıralanmış tek bir dizi olurlar. Bu işleme Merge(birleştirme) diyeceğiz.

Merge Sort, dizimizi ortadan ikiye böler, bu iki diziyi kendi içinde sıralar, sonra onları Merge yapar, böylece asıl dizimiz sıralanmış olur. Büyük dizinin bölünmesiyle oluşan iki yarım diziyi kendi içinde sıralarken yine aynı işlemi kullanır. Diğer bir deyişle recursion kullanarak sıralarız dizimizi. Her bir birleştirme, N işlemde yapılır. Her seferinde ikiye böleceğimiz için toplamda en fazla logN adet basamak olur. Her basamakta N işlem yaptığımız için masrafımız O(N x logN) olur. Ancak bu birleştirme işlemleri sırasında her seferinde yeni boş bir dizi almamız gerekir. Bu yüzden memory masrafımız da O(N x logN) olur. Aşağıda örnek bir kod parçası ve sıralanmamış bir dizinin parça parça ayrılıp, sıralanıp, merge yapılmasının örneği var.

```
void mergeSort(int bas,int son){

    if(bas==son) return;

    mergeSort(bas,(bas+son)/2); // ikiye ayırıp parçaları kendi içinde
    mergeSort((bas+son)/2+1,son); // sıralaması için fonksiyona gönderdik.

    int tmp[MAXN];
    int uz = son-bas+1;
    int p1=bas,p2=(bas+son)/2+1;

    for(int i=0;i<uz;i++)          // iki diziyi yeni bir dizide
        if(p1 > (bas+son)/2)       // sıralanmış biçimde birleştirdik.
            tmp[i] = ar[p2++];
        else if(p2 > son)
            tmp[i] = ar[p1++];
        else if(ar[p1] > ar[p2])
            tmp[i] = ar[p2++];
        else
            tmp[i] = ar[p1++];

    for(int i=0;i<uz;i++)           // dizinin sıralanmış halini
        ar[bas+i]=tmp[i];           // orijinal dizimize kaydettik.
}

int main(){

    mergeSort(1,N);
```

```
{18, 6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{18, 6, 9, 1, 4} {15, 12, 5, 6, 7, 11}
{18, 6} {9, 1, 4} {15, 12, 5} {6, 7, 11}
{18} {6} {9} {1, 4} {15} {12, 5} {6} {7, 11}
{18} {6} {9} {1} {4} {15} {12} {5} {6} {7} {11}
{18} {6} {9} {1, 4} {15} {5, 12} {6} {7, 11}
{6, 18} {1, 4, 9} {5, 12, 15} {6, 7, 11}
{1, 4, 6, 9, 18} {5, 6, 7, 11, 12, 15}
{1, 4, 5, 6, 6, 7, 9, 11, 12, 15, 18}
```

Heap Sort Time: O(N x logN) Memory: O(N)

Heap, veya Priority Queue, değişkenleri binary bir ağaçta tutan bir veri yapısıdır. Her eleman, 2 çocuğundan da küçük-eşit olmak zorundadır. Heap'e yeni eklenen sayı yaprak olarak eklenir, yeni eklenen eleman atasından küçük olduğu sürece atasıyla yer değiştire değiştire çıkabildiği kadar yukarıya çıkar. Heap'ten eleman alırken her zaman root, kök, yani en üstteki eleman alınır. En küçük elemanı aldıktan sonra çocuklarından küçük olan onun yerine geçer. Sonra o eleman için de aynısı tekrarlanır, onun çocuklarından küçük olan da onun yerine geçer. Bu şekilde heap'in özelliği her zaman korunmuş olur. Ağaç binary olduğundan dolayı, N eleman için logN kat olur. Bu yüzden her bir ekleme veya çıkarma işlemi logN işlem olur.

Heap sort, N tane elemanın bir heap'e eklenmesi, sonra teker teker alınmasıyla yapılan bir sort yöntemidir. Yukarıda bahsettiğimiz gibi, her bir ekleme ve çıkarma işlemi logN olacağı için, N elemanı heap kullanarak sortlamak N x logN maliyetinde olacaktır. Heap için priority_queue kullanabilirsiniz.

```cpp
#include <cstdio>
#include <iostream>
#include <queue>

using namespace std;

int N;

priority_queue <int> heap;

int main(){

    cin >> N ;

    for(int i=1;i<=N;i++){

        int a;
        scanf(" %d",&a);

        heap.push(a);
    }

    while( !heap.empty() ){

        printf("%d ",heap.top());
        heap.pop();
    }

    return 0;
}
```

*Priority queue en küçük değil en büyük elemanı en üstte tutar.

Quick Sort Time: O(N x logN) Memory: O(N)

Quick sort, insan gibi davranması düşünülerek yazılmış bir algoritmadır. Eldeki verileri büyükler ve küçükler olmak üzere iki sınıfa ayırmak, daha sonra o grupları da kendi içinde sıralamak için aynı metodu uygulamaktan ibarettir. Merge sort'un tersten yapılışı gibi düşünebiliriz, yalnız merge sort yaparken ayırdığımız parçalar her zaman asıl parçanın yarısı uzunluğundaydı; burada ise öyle bir garanti yok elimizde, çünkü dizimizin ortanca elemanını bilmiyoruz. Kısaca, bir değer belirlenir, N işlemde bütün elemanlara bakılır, o değerden küçük olanlar soluna, büyük olanlar sağına geçer. Daha sağ ve sol parça da kendi içinde aynı işlem gerçekleştirilerek sıralanır. Burada önemli olan belirlenen pivot değerdir. Dizinin ortanca elemanını tutturursak maliyetimiz O(N x logN) e kadar düşebilir. Ama ortanca değer değil de en büyük değeri seçersek maliyetimiz $O(N^2)$'ye kadar tırmanabilir. Yani ortanca değeri bulmanızı sağlayacak hızlı bir algoritmanız yoksa Quick sort kullanmak şans ister. Yine de ortalama maliyet, expected value, O(N x logN)' e yakındır, eğer pivot eleman random seçilirse. Quick sort'un farklı varyasyonları vardır. Hem zaman hem hafıza olarak maliyeti kesin değildir, her varyasyon farklı özelliklere sahiptir. Kimisinin optimizasyonları hızlı çalışıp çok memory ister, kimisi tam tersini yapar.

```
Array quickSort(Array data) {
   if (Array.Length <= 1)
      return;
   middle = Array[Array.Length / 2];
   Array left = new Array();
   Array right = new Array();
   for (int i = 0; i < Array.Length; i++)
      if (i != Array.Length / 2) {
         if (Array[i] <= middle)
            left.Add(Array[i]);
         else
            right.Add(Array[i]);
      }
   return concatenate(quickSort(left), middle, quickSort(right));
}
```

Quick sort hakkında her şey > **wikipedia.org/wiki/Quicksort**

Radix Sort

Radix sort uygulaması bütün verileri ikili olarak kıyaslamak üzerine tasarlanmamıştır. Radix sort önce en küçük basamakları ele alır ve bu değerlerine göre elemanları kutularda saklar. Eğer 4 bit için oluşturursak toplamda 16 adet kutu oluşması gerekmektedir. Ardından elemanları en küçük 4 bitlerine göre bu kutulara yerleştiririz. Ardından aynı işlemi ikinci en küçük 4 bite göre gerçekleştiririz. Bu işlemi en büyük basamaklarına kadar devam ettireceğiz. Bu işlemler sırasında kutulara yerleşecek elemanları önceki sıralamaları da göz önüne almalıyız.

Örneğin 1 bit için radix sort uygulamasının gerçekleştirilişini görelim:

```
{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{6, 4, 12, 6} {9, 1, 15, 5, 7, 11}
{4, 12, 9, 1, 5} {6, 6, 15, 7, 11}
{9, 1, 11} {4, 12, 5, 6, 6, 15, 7}
{1, 4, 5, 6, 6, 7} {9, 11, 12, 15}
```

Şimdi işlemin nasıl gerçekleştiğinden bahsedelim. Elimizdeki diziyi önce ilk yani en küçük bitlerine göre ait oldukları kutulara atıyoruz. Ardından ikinci bitlerine göre yine aynı işlemi gerçekleştireceğiz. Ancak bir kutuya öncelikle üst basamakta dizilen elemanlar arasından en küçüğünü atmalı ve daha sonra adım adım üst sıradaki gibi atmalıdır. Bu sayede 2. basamağı aynı olan elemanlar ilk basamaklarına göre de sıralanmış olacaktır. Ardından bu işlemi basamak sayısı sona erene dek devam ettirecektir.

Şimdi de aynı işlemi 2 bitlik gruplar saklayarak nasıl yapacağımıza bakalım.

```
{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{4, 12} {9, 1, 5} {6, 6} {15, 7, 11}
{1} {4, 5, 6, 6, 7} {9, 11} {12, 15}
```

Görüldüğü gibi sıralamak daha az adım gerektirmiştir.

Radix sort diğer veri yapılarından farklı bir mantığa sahiptir ve bir çok alanda diğer veri yapılarından daha hızlı çalışma imkanına sahiptir. Radix sort'un çalışma zamanı O(nk)'dır. k kutu sayısını temsil etmektedir. Ancak radix sort'un dezavantajı her adımda sakladığı verilerin büyük olması ve tekrar tekrar kopyalanıp

taşınmasıdır. Bu durumda her ne kadar zaman karmaşıklığına etki etmese de çalışma zamanını düşürmektedir.

Son açıklama

Bilindiği üzere birçok dilin kütüphanelerinde yer alan hazır sort fonksiyonları çoğu zaman yazılan kodlar için yeterli olmaktadır. Ancak sort algoritmalarının bilinmesi ve mantıklarının anlaşılmış olması yeni algoritmaların öğrenilmesine de kolaylık sağlayacaktır. Bunun yanında verilen algoritmaların kullanılmasının kodunuzun çalışma zamanını hızlandırdığı durumlar da mevcuttur. Buna bir örnek olarak suffix array algoritmasının kodlanmasında radix sortun kullanılması olarak verilebilir.