Algoritma ne?

Bir problemi çözmek amacıyla adım adım uygulanan kurallar dizisi.

- Bir programın işletim süresi programın zaman karmaşıklığını verir.
- " " işletildiği sürece gerekli olan bellek mik. bellek karmaşıklığıdır.

Every-case Running Time

- Temel işlem giriş boyutu n için aynı zamanda yapılır. T(n) n boyut için zaman.
- Bu analiz yönteminde harcanan zaman yalnızca giriş boyutuna bağlı.

Worst-case Running Time

- En kötü ihtimalle ne kadar süreceğini tahmin edilir.
- Giriş boyutuna ve giriş değerlerine bağlı.

Average-case Running Time

- Algoritmanın ortalama olarak ne kadar sürede gerçekleştiğidir.
- Tüm girdilerin ortalamasıdır.

Best-case Running Time

- En iyi ne kadar sürede gerçekleşeceği.

⇒ Asimptotik Analiz

Döngü ⇒ Döngü n defa ise c(sabit) x n = cn dir.

```
i=1;                 → c1    1
sum=0;               → c2    1
while (i<=n) {       → c3    n+1
    i=i+1;           → c4    n
    sum=sum+i;       → c5    n
}
```


İç içe Döngü ⇒ Toplam zaman bütün döngülerin çalışma sayılarının çarpımı.

```
for (i=1; i<=n; i++) {
    for (i=1; i<n; i++) {
        k=k+1;         } n
    }
}                      n
```

$c \times n \times n = cn^2$

If-Then-Else ⇒ Test zamanına then veya else kısmındaki çalışma zamanının hangisi büyükse o kısım eklenir.

```
if (depth() != otherStack.depth()) {
    return false;                           ] Sabit
}
else {
    for (int n=0; n< depth(); n++) {
        if(!list[n].equals(otherStack.list[n]))
            return false;                   ] (Sabit + Sabit) x n
    }
}
```

$c_0 + (c_2 + c_3) \times n$

Logaritmik Karmaşıklık ⇒ Problemin büyüklüğünü belli oranda azaltmak için sabit bir zaman harcanıyorsa bu algoritma O(log N) dir.

N sayfalı bir sözlükten sözcük arama

→ Sözlüğün orta kısmına bakılır.

→ Sağdamı soldamı kalır.

→ Sözcük bulunana kadar devam eder.

Big O Analizi (O - Notasyonu) (3)

Yapılan işi girdi boyutunun bir fonk. olarak ele almış olduk.

Problemin girdi boyutuna bağlı olarak fonk. en hızlı artışı gösteren terim belirler.

O Notasyonu

İki algoritma karşılaştırılırken sonun mertebesinden konuşulur.
Mertebesi büyük olan daha yavaştır.
Sabit ihmal edilir $O(7n^2) < O(n^2)$

$f(n) = 10n + 25n^2$ $O(n^2)$ $\quad f_2(n) = 20n\log n + 5n$ $O(n\log n)$ $\quad f_3(n) = 12n\log n + 0.05n^2$ $O(n^2)$

Selection Sort ⇒ Ever-case Running Time algoritmadır. Toplam maliyet $\frac{n^2-n}{2}$
Her durumda n^2 zamanda çalışır.
En iyi ve en kötü durum eşittir.

Optimum Bubble Sort ⇒ Best case n / Worst case n^2 } Çalışan algoritma. Toplam maliyet $\frac{n^2-n}{2}$
Theta notasyonu ile ifade edilmek.

Insertion Sort ⇒ Best case n / Worst case n^2 } Çalışan algoritma.

En iyi durum maliyet: $n-1$

En kötü durum maliyet: $\frac{n^2-n}{2}$

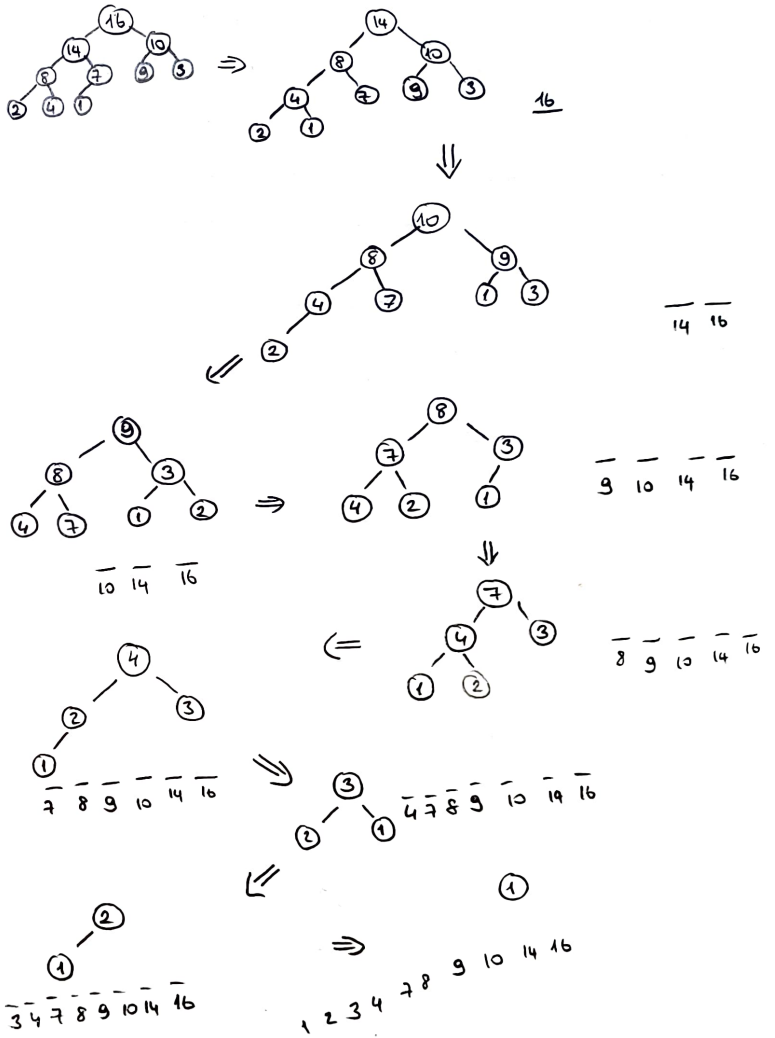
Quick Sort ⇒ Best case $n\log n$ En kötü durum maliyeti: $\frac{n \times (n-1)}{2}$
Worst case n^2

Merge Sort ⇒ Every case running time bir algoritma Toplam maliyet $2n\log n$
Her durumda $n\log n$ zamanda çalışır.

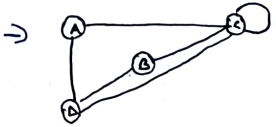
Heap Sort ⇒ Every case running time bir algoritma
Her durumda $n\log n$ ifade edilir

(4)

Max - Heap yapısına uygun dizinin ilk elemanı max / Min - " " " " " " min } elemandır.



Graflar



	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	1	1
D	1	1	1	0

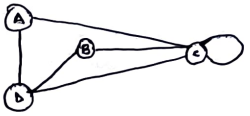
```
int A[4][4] = {{0,0,1,1},{0,0,1,1},{1,1,1,1},{1,1,1,0}};

int baglantiVarmi (int dugum1, int dugum2)
{
    if( A[dugum1][dugum2] != 0) return 0;
    else return -1;
}
```

```
5   7   (1)
0   2   17
0   4   23
1   2   39
1   4   15
2   3   44
2   4   21
3   4   20
(i) (j) (maliyet)
```

```
main()
{
  FILE *di;
  int *A, i, j, maliyet;
  if( (di=fopen("graf.txt","r"))==NULL) {
      Puts("Dosya Acilamadi!");
      exit(0);
    }
  if(feof(di)) exit(0); (hic yeri yok)
  (1) fscanf(di,"%d,%d", &N, &M);
  A = malloc(N*N*sizeof(int)); (A[5][5]
  if(A==NULL) printf("Bellekte boş yer yok");
  for(i=0; i<N; i++)
      for(k=0; k<N; k++)
          A[i][k] = 0          } matrisi 0 la.
  for(k=0; k<m; k++)
      fscanf(di, "%d,%d,%d", &i, &j, &maliyet);
      A[i][j] = maliyet
      A[j][i] = maliyet
```

```
for(i=0;i<N;i++)
  for(k=0;k<N;k++)
    Printf("%d", A[i][k]
```

(5)



	A	B	C	D
A	0	0	1	1
B	0	0	1	1
C	1	1	1	1
D	1	1	1	0

```
Struct liste {
    Char dugumAd;
    int maliyet;
    Struct liste *arka;
};
Struct dizi {
    char dugumAd;
    Struct liste *bag;
};
Struct dizi V[N]; (N düğ. graf için)
```

(6)

Dizi

A → C → D
B → C → D
C → A → B → C → D
D → A → B → C

```
Struct Dugum {
    int Veri;
    Struct Dugum *sonraki;
};
```

```
Struct Graf {
    int dugum_sayisi;
    Struct Dugum **dugumler;
};
```

(7)

```
Struct Dugum* dugum_olustur (int veri) {
    Struct Dugum* yeni_dugum = (struct Dugum*) malloc (sizeof(struct Dugum));
    Yeni_dugum -> Veri = Veri;
    Yeni_dugum -> Sonraki = NULL;
    return yeni_dugum; }
```

```
Struct Graf* graf_olustur(int dugum_sayisi) {
    Struct Graf* yeni_graf = (struct Graf* )malloc (sizeof(struct Graf));
    yeni_graf -> dugum_sayisi = dugum_sayisi;
    yeni_graf -> dugumler = (struct Dugum**) malloc (dugum Sayisi * sizeof(struct Dugum*));
    for(int i=0; i< dugum_sayisi; i++) {
        Yeni_graf -> dugumler [i] = NULL; }
    return yeni_graf;
}
```

```
Void kenar_ekle (Struct Graf* graf, int baslangic, int hedef) {
    Struct Dugum* yeni_dugum = dugum_olustur (hedef);
    Yeni_dugum -> Sonraki = graf -> dugumler [baslangic];
    graf -> dugumler [baslangic] = yeni_dugum;
    Yeni_dugum = dugum_olustur (baslangic);
    Yeni_dugum -> sonraki = graf -> dugumler [hedef];
    graf -> dugumler [hedef] = yeni_dugum;
}
```

başlangıç düğümü

yönlendirilmemiş graf

(8)

```
Void kenar_sil ( Struct Graf* graf, int baslangic, int hedef) {
    Struct Dugum* prev = NULL;
    Struct Dugum* current = graf -> dugumler [baslangic];
    while (current != NULL && current -> veri != hedef) {
        prev = current;
        current = current -> sonraki; }
```

Silinecek kenar bulana kadar

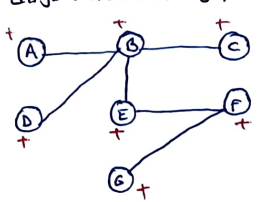
```
    if (current != NULL) {
        if (prev != NULL) prev -> sonraki = current -> sonraki;
        else( graf -> dugumler [baslangic]=current -> sonraki;
        free (current); }
```

baş düğümün baş.

```
    prev = NULL;
    current = graf -> dugumler [hedef];
    while ( current != NULL && current -> veri != baslangic) {
        prev = current;
        current = current -> sonraki; }
    if (current != NULL) {
        if (prev != NULL)
            prev -> sonraki = current -> sonraki;
        else
            graf -> dugumler [hedef] = current -> sonraki;
        free (current);
    } }
int kenar_sayisi (Struct Graf* graf){
    int kenar_sayisi =0;
    for(int i=0; i< graf -> dugum_sayisi; i++) {
        Struct Dugum * temp = graf -> dugumler [i];
        while (temp != NULL)
            kenar_sayisi ++;
            temp = temp -> sonraki;
        return kenar_sayisi / 2; }
```

BFS Yöntemi

Başlangıç düğümünden gidilebilecek tüm komşu düğümlere gidilir ara düğümlerde başlangıç düğümü gibi olur.



Kuyruk

A ⟶ A
B ⟶ B
C,E,D
E,D ⟶ C
E,D,F ⟶ E
D,F ⟶ D
F,G ⟶ F
G ⟶ G

(9)

```
int dizi[6][6]; bool ziyaret[6]; (1)

Struct node {
    int data;          (2)
    Struct node *next;
};

Struct node* dugumolustur(int x){
  Struct node* dugum = struct node* malloc...
  dugum->data = x          (4)
  dugum->next = NULL;
  return dugum;
}

Void deQueue (){
  if(front==NULL) return;  (6)
  if(front->next)==NULL)
     front=rear=NULL;
  else {
    Struct node*temp = front->next;
    free(front);
    front = temp;
    }
}

Void BFS (int root) {  (8)
  int i;
  for(i=0; i<6; i++) {
     ziyaret[i] = false
  }
  ziyaret[root] = true;
  enQueue(root);
```

(10)

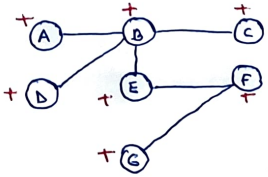
```
Struct node* front = NULL;   (3)
Struct node * rear = NULL;

Void enQueue (int x) {
  Struct node* yeni = dugumolustur(x);
  if (front== NULL):
      front=rear= yeniDugum;
  else rear->next = NULL;      (5)
       rear = yeniDugum;
}

bool bosMu()   (7)
 {
   if(front==NULL) return true;
    else return false;
 }
```

(9)

```
while((bosMu) == false) {
   root = front->data;
   Printf("%d", root);
   deQueue();
   for(i=0; i<6; i++) {
      if(ziyaret[i]= false && dizi[root][i]==1)
        {
          ziyaret[i] = true;
          enQueue(i);
        }
    }
}
```

DFS Yöntemi

Graf üzerinde dolaşmaya başlangıç düğümünün bir ayrıtından başlayıp o ayrıt üzerinden gidilebilecek en uzak düğüme kadar sürdürür.



STACK

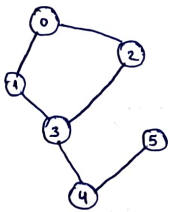
A → B → ~~C~~ → ~~D~~ → E → F → G

C'den sonra gidebileceği bir yer yok pop fonk Stack'ten

D'den " " " " " " " " "

En son Stack'teki tüm düğümleri siler.

```
int Array[6][6];
bool Visited[6];
Void DFS(int root, bool visited[]){
  Visited[root] = true;
  Printf("%d", root);
  int i;
for(i=0; i<6; i++){
   if (Array[root][i]==1 && Visited[i]=false)
      {
        DFS(i, Visited);
      }
}
Void readMatrix() {
int i=0;
FILE *fp = fopen("matris.txt",r);
while(fscanf(fp,"%d %d %d %d %d %d",
       &Array[i][0], &Array[i][1]--
       ) != EOF)
   {i= i+1;
   }
int main() { readMatrix();
             int i;}
        for(i=0; i<6; i++) Visited[i]=false;
        DFS(0, Visited)
          return 0;
```



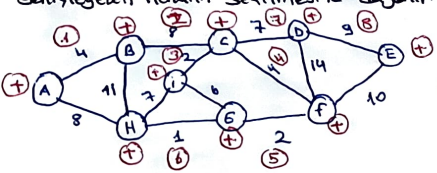

```
0 1 1 0 0 0
1 0 0 1 0 0
1 0 0 1 0 0
0 1 1 0 1 0
0 0 0 1 0 1
0 0 0 0 1 0
```

Matris.txt


```
Void DFS (grafVeriYapisi G)
{
  Short int gidildi[n] ={0};
  Unsigned int duqum;
  for(duqum=0; duqum<n; duqum++)
   {
     if ( gidildi[duqum]==0)
        {
          GrafiDolasDFS(düqum);
        }
   }
}
GrafiDolasDFS(Unsigned int duqum)
{
   Unsigned int i;
   gidildi[duqum]=1;
   --- /* düğüme ulaşılınca yapılacak işlem */
   for(i=0; i<n; i++)
    { if(Gveri[duqum][i] !=0 && gidildi[i]==0)
        {
          GrafiDolasDFS(i);
        }
    }
}
```

Prim Algoritması

En az maliyetli kenardan başlanıp onun uçlarından en az maliyetle genişleyecek kenarın seçilmesine dayanır.



A'dan başlandı

A → B → C → I → F → G → H → D → E

Prim algoritmasında cycle oluşturmamak temel şart.

```
int primsMST(int Key[], bool Visited[]){
    int min = 100;
    int minIndex, i;
    for(i=0; i<5; i++){
        if(Visited[i] == false && Key[i] < min){
            min = Key[i];
            minIndex = i;
        }
    }
    return minIndex;
}

Void PrintMST(int Array[5][5], int Parent[]){
    int i;
    for(i=0; i<5; i++)
    {
        Printf("%d - %d => %d", Parent[i], i, Array[i][Parent[i]]);
    }
}
```



```
Void prims(int graph[][])
{
    int Key[5];
    int Parents[5];
    bool Visited[5];
    int i, j, t;
    for(i=0; i<5; i++){
        Visited[i] = false;
        Key[i] = 100;
    }
    Key[0] = 0;
    Parents[0] = -1;
    for(j=0; j<5; j++)
    {
        int u = primsMST(Key, Visited);
        Visited[u] = true;
        for(t=0; t<5; t++){
            if(graph[u][t] && visited[t] = false && graph[u][t]<Key[t]
                Key[t] = graph[u][t]
                Parents[t] = u;
            }
        }
    }
    PrintMST(int Array[5][5], int Parents[]);
```

Kruskal Algoritması

Kruskal algoritması daha az maliyetli kenarları tek tek değerlendirerek yol ağacını bulmaya çalışır.

```
typedef struct_kenar {
    int ilk;
    int son;
    int maliyet;
} KENAR;

KENAR kenar[200], EKYA[200], gecici;
int kume[100];
int N,M,K;
int Graf[100][100];

int main(){
int i,j,t, x,y;

okugraf();
t=0;
for(i=0;i<N;i++){
    for(j=i+1;j<N;j++){
        if (Graf[i][j] != -1){
            kenar[t].ilk = i;
            kenar[t].son = j;
            kenar[t].maliyet = Graf[i][j];
            t++;
        }
    }
}
```

```
Void birlestir();
Void okugraf();
```
} fonk. bildirimi

```
for(i=0; i<(M-1); i++){
  for(j=0; j<(M-1); j++){
    if(kenar[j].maliyet > kenar[j+1].maliyet)
    {
      gecici = kenar[j];
      kenar[j] = kenar[j+1];
      kenar[j+1] = gecici;
    }
  }
}
for (i=0; i<N; i++) kume[i]=i;

i=0;
while(K<N-1 && i<M){
  x=kume[kenar[i].ilk];
  y=kume[kenar[i].son];
  if( x!=y)
  {birlestir(x,y);
   EKYA[K] = kenar[i];
   K++;
  }
  i++;
}
Void birlestir(int p, int q){
  int i;
/* for(i=0; i<N; i++){
    if(kume[i]==q) {
      kume[i]=p
    }
  }
}
```



```
int main(){
  FILE *fp = fopen("input.txt",r);
  int dugum sayisi;
  int kenarsayisi
  fscanf(fp,"%d", &kenarsayisi);
  fscanf(fp,"%d", &dugumsayisi);
  int *dolasilan = int* malloc(sizeof(int)*dugumsayisi);
  for(int i=0 i<dugumsayisi i++)
        dolasilan[i] = 0;
  int *kimden = int * malloc sizeof int kenar
  int *kime =          "
  int *agirlik =       "
  int i=0;
  while (!feof(fp))
    fscanf(fp %d %d %d kimden, kime, agirlik);
  fclose(fp);
  for(int i=0; i<kenarsayisi; i++)
    for(int j=0; j<kenarsayisi; j++)
      if(agirlik[j] > agirlik[j+1])
        int x = agirlik[j];
        agirlik[j] = agirlik[j+1];
        agirlik[j+1] = x
        x = kime[j]
        kime[j] = kime[j+1]
        kime[j+1] = x
        x = kimden[j]
        kimden[j] = kimden[j+1]
        kimden[j+1] = x
  }}
i=0;
int toplammaliyet =0;
for(int j=0 j<kenarsayisi j++)
  if(dolasilan[kime[j]]=0 || dolasilan kimden[j] =0
        toplam maliyet += agirlik[j]
        dolasilan[kime[j]] = 1
        dolasilan[kimden[j]] = 1
```

Dijkstra Algoritması

Başlangıç düğümünden diğer tüm graflara en kısa yolu belirler.

Ağırlıklı ve yönlü ~~graf~~ graflar için geliştirilmiştir.

Negatif ağırlıklı bir grafta Dijkstra algoritması işlemez.

```
#define N 6    # define EBAS 0X7FFFFFFF (32 bitlik en büyük artı sayı)
int GRAF[N][N];
int EKM[n];      Char ROTA[n][n] = {NULL};
int main {
  int i;
  Dijkstra();
  for(i=0; i<N; i++)
     Printf("EKM[%d] = %d \n", i, EKM[i]);
  for(i=0; i<N; i++)
     Printf("ROTA[%d] = %s \n", i, ROTA[i]);
  getch();
}
```



```
Void Dijkstra() {
 char* ptr, ELEALINDI[N] = {0};
 int i,j, ead, ek;
 EKM[0] = 0;
 for(i=1; i<N; i++)
   { EKM[i] = EBAS; }  Tüm uzaklıklar sonsuz

 ead=0;
 for(i=0; i<N; i++) }  Tüm grafı gezer
   for(j=0; j<N; j++)
     if(!ELEALINDI[j]) } Düğüm daha önce gidilmediyse
       if(GRAF[ead][j] != -1)
         if(EKM[j] > GRAF[ead][j] + EKM[ead]) {
             EKM[j] = GRAF[ead][j] + EKM[ead];
             Strcopy(ROTA[j], ROTA[ead]);
             ptr = ROTA[j];
             while(*ptr != NULL)
                 ptr++;
              *ptr = 'A' + ead;
             }

 ek = EBAS;
 for(j=1; j<N; j++)
   if(!ELEALINDI[j])
     if(EKM[j] < ek)
       ek = EKM[j]; ead = j;
 ELEALINDI[ead] = 1;
}
)
```

Bellman Ve Ford Algoritması


```
#define max_vertices 100   #define max_edges 100 #define INF INT_MAX
Struct Edge {
    int source, destination, weight;
}
Struct Graph {
    int V, E;
    Struct Edge edges[MAX_EDGES];
};

int main() {
Struct Graph graph;
int V, E, source;
Printf ("Baslangic düğümü gir");
Scanf ("%d", &source);

Bellman Ford (&graph, source);
return 0;
}
```



```
Void Bellman Ford (Struct Graph* graph, int source) {
int V = graph->V;  int E = graph->E;  int dist[max_vertices];
for (int i=0; i<V; i++)
        dist[i] = INF;
dist[source] = 0;   Baslangic düğümü mesafesi 0

for(int i=1; i<=V-1; i++) {
    for(int j=0; j<E; j++) {
      int u = graph->edges[j].source;
      int v = graph->edges[j].destination;
      int weight = graph->edges.[j].weight;
      if( dist[u] != INF && dist[u] + weight < dist[v])
          dist[v] = dist[u] + weight;

for(int i=0; i<E; i++) {
    int u = graph->edges[i].source;
    int v = graph->edges[i].destination;
    int weight = graph->edges[i].weight;
    if( dist[u] != INF && dist[u] + weight < dist[v];
        Printf ("Grafta negatif döngüm var");

Printf ("u Mesafe");
for(int i=0; i<V; i++){
      Printf ("%d %d \n") source, i, dist[i]);

}
```