আমরা শুরুতেই শিখেছি কিভাবে শর্টেস্ট পাথে এক জায়গা থেকে আরেক জায়গায় যেতে হয়। সেজন্য আমরা শিখেছি ব্রেডথ ফার্স্ট সার্চ নামের একটি সার্চিং অ্যালগোরিদম। অ্যালগোরিদমটি চমৎকার কিন্তু সমস্যা হলো সে ধরে নেয় প্রতিটি রাস্তা দিয়ে যেতে সমান সময় লাগে, মানে সব এজ এর কস্ট সমান। প্র্যাকটিকাল লাইফে বেশিভাগ ক্ষেত্রেই এটা অচল হয়ে পড়ে, তখন আমাদের দরকার পরে ডায়াক্সট্রা। প্রথমে নাম শুনে আমার ধারণা হয়েছিলো ডায়াক্সট্রা খুবই ভয়ংকর কোনো জিনিস কিন্তু আসলে বিএফএস লেখার মতোই সহজ ডায়াক্সট্রা লেখা, আমি তোমাদের দেখানোর চেষ্টা করবো কিভাবে বিএফএসকে কিছুটা পরিবর্তন করে একটা প্রায়োরিটি কিউ যোগ করে সেটাকে ডায়াক্সট্রা বানিয়ে ফেলা যায়।

ডায়াক্সট্রা শুরু করার আগে আমরা পাথ রিল্যাক্সেশন(relax) নামের একটা ছোট্ট জিনিসের সাথে পরিচিত হই। ধরো সোর্স থেকে প্রতিটা নোডের ডিসটেন্স রাখা হয়েছে d[]d[] অ্যারেতে। যেমন d[3]d[3] মানে হলো সোর্স থেকে বিভিন্ন এজ পার হয়ে ৩ এ আসতে মোট d[3]d[3] ডিসটেন্স লেগেছে। যদি ডিসটেন্স জানা না থাকে তাহলে ইনফিনিটি অর্থাৎ অনেক বড় একটা মান রেখে দিবো। আর cost[u][v]cost[u][v] তে রাখা আছে u–vu–v এজ এর cost। ধরো তুমি বিভিন্ন জায়গা ঘুরে ফার্মগেট থেকে টিএসসি তে গেলে ১০ মিনিটে, আবার ফার্মগেট থেকে কার্জন হলে গেলে ২৫ মিনিটে। তাহলে তোমার কাছে ইনফরমেশন আছে:

$$d[\text{টিএসসি}] = ১০, d[\text{কার্জনহল}] = ২৫$$

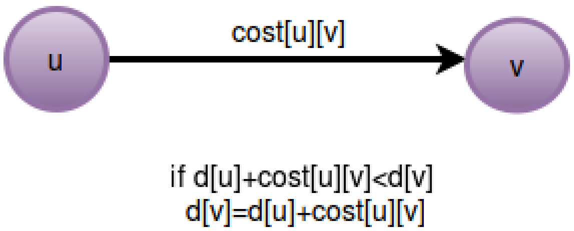
এখন তুমি দেখলে টিএসসি থেকে ৭ মিনিটে কার্জনে চলে যাওয়া যায়,

$$cost[\text{টিএসসি}][\text{কার্জন}] = ৭$$

তাহলে তুমি ২৫ মিনিটের জায়গায় মাত্র ১০ + ৭ = ১৭ মিনিটে কার্জনহলে যেতে পারবে। যেহেতু তুমি দেখেছো:

$$d[\text{টিএসসি}] + cost[\text{টিএসসি}][\text{কার্জন}] < d[\text{কার্জনহল}]$$

তাই তুমি এই নতুন রাস্তা দিয়ে কার্জন হলে গিয়ে d[কার্জনহল] = d[টিএসসি] + cost[টিএসসি][কার্জন] বানিয়ে দিতেই পারো!!

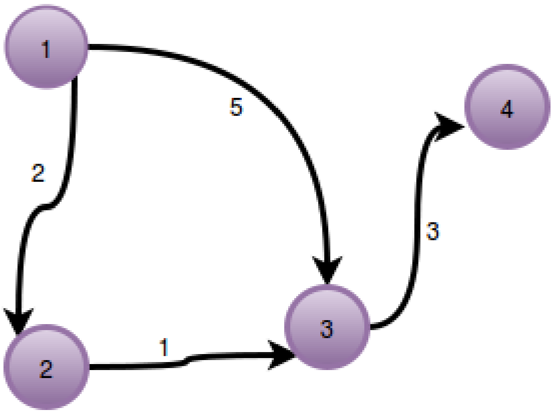


উপরের ছবিটা সেটাই বলছে। আমরা u থেকে v তে যাবো যদি **d[u]** + **cost[u][v]** < **d[v]** হয়। আর d[v]d[v] কে আপডেট করে d[v]=d[u]+cost[u][v]d[v]=d[u]+cost[u][v] বানিয়ে দিবো। ভবিষ্যতে যদি কার্জনহলে অন্য রাস্তা দিয়ে আরো কম সময়ে যেতে পারি তখন সেই রাস্তা এভাবে কম্পেয়ার করে আপডেট করি দিবো। ব্যাপারটা অনেকটা এরকম:

```
if d[u] + cost[u][v] < d[v]:
d[v] = d[u] + cost[u][v]
```

উপরের অংশটা যদি বুঝে থাকো তাহলে ডায়াক্সট্রা বোঝার ৬০% কাজ হয়ে গেছে। না বুঝে থাকলে আবার পড়ো।

বিএফএস নিশ্চয়ই তুমি ভালো করে বুঝো। বিএফএস এ আমাদের একটা নোডে কখনো দুইবার যাওয়া দরকার হয়নি, আমরা প্রতিবার দেখেছি একটা নোড ভিজিটেড কিনা, যদি ভিজিটেড না হয় তাহলে সেই নোডকে কিউতে পুশ করে দিয়েছি এবং ডিসটেন্স ১ বাড়িয়ে দিয়েছি। ডায়াক্সট্রাতে আমরা একই ভাবে কিউ তে নোড রাখবো তবে ভিজিটেড দিয়ে আপডেট না করে নতুন এজকে "রিল্যাক্স" বা আপডেট করবো উপরের পদ্ধতিতে। নিচের গ্রাফটা দেখো:



ধরে নেই সোর্স হলো ১ নম্বর নোড। তাহলে

$$d[1] = 0, d[2] = d[3] = d[4] = infinity(a\ large\ value)$$

ইনফিনিটি কারণ ২,৩,৪ এর দূরত্ব আমরা এখনো জানিনা, আর সোর্সের দূরত্ব অবশ্য শূন্য। এখন তুমি আগের বিএফএস এর মতোই সোর্স থেকে যতগুলো নোডে যাওয়া যায় সেগুলা আপডেট করার চেষ্টা করো, আপডেট করতে পারলে কিউতে পুশ করো। যেমন ১ – ২ এজটা ধরে আমরা আগাবো কারণ $d[1]+2<d[2]d[1]+2<d[2]$ এই শর্তটা পূরণ হচ্ছে। তখন $d[2]d[2]$ হয়ে যাবে ২, একই ভাবে ১ থেকে ৩ এ গেলে $d[3]d[3]$ হয়ে যাবে ৫।
কিন্তু ৫ তো ৩ নম্বরনোডে যাওয়ার শর্টেস্ট ডিসটেন্স না! আমরা বিএফএস এ দেখেছি একটা নোড একবারের বেশি আপডেট হয়না, সেই প্রোপার্টি এখানে কাজ করছেনা। ২ নম্বর নোড থেকে ২-৩ এজ ধরে এগিয়ে আবার আপডেট করলে তখন $d[3]d[3]$ তে $d[2]+1=3d[2]+1=3$ পাবো। তাহলে আমরা দেখলাম এক্ষেত্রে একটা নোড অনেকবার আপডেট হতে পারে। (প্রশ্ন: সর্বোচ্চ কত বার?)
আমরা তাহলে আগের বিএফএস এর সুডোকোডের আপডেট অংশ একটু পরিবর্তন করি যাতে একটা নোড বার বার আপডেট হতে পারে:

```
procedure BFSmodified(G,source):
    Q = queue(), distance[] = infinity
    Q.enqueue(source)
    distance[source] = 0
    while Q is not empty
       u ← Q.pop()
       for all edges from u to v in G.adjacentEdges(v) do
           if distance[u] + cost[u][v] < distance[v]
                    distance[v] = distance[u] + cost[u][v]
                     Q.enqueue(v)
           end if
       end for
     end while
     Return distance
```

আমরা ঠিক আগের বিএফএস এর কোডেই জাস্ট কস্ট বসায় বারবার আপডেট করছি! এই কোড তোমাকে সোর্স থেকে প্রতিটা নোডের শর্টেস্ট পাথ বের করে দিবে কিন্তু কমপ্লেক্সিটির দিক থেকে এটা খুবই বাজে! এজন্য আমাদের লাগবে একটা প্রায়োরিটি কিউ।

বিএফএস এ আমরা যখন ১ নোড থেকে অনেকগুলো নোডে যাচ্ছি তখন সেই নোডগুলো থেকে আবার নতুন করে কাজ করার সময় “আগে আসলে আগে পাবেন” ভিত্তিতে কাজ করছি। যেমন উপরের গ্রাফে ১ থেকে আগে ৩ নম্বর নোডে এবং তারপর ২ নম্বর নোডে এ গেলে আগে ৩ নিয়ে কাজ করছি, এরপর ২ নিয়ে কাজ করছি।

ভালো করে দেখো এখানে কি সমস্যাটা হচ্ছে। ৩ নিয়ে আগে কাজ করলে আমরা ৪ এর ডিসটেন্সকে আপডেট করে দিচ্ছি ডিসটেন্স ৫ + ৩ = ৮ হিসাবে। পরবর্তীতে যখন ২ দিয়ে ৩ কে আবার আপডেট করা হচ্ছে তখন ৩ এর ডিসটেন্স হয়ে গিয়েছে ৩, এবার ৪ এর ডিসটেন্সকে আবার আপডেট করছি ৩ + ৩ = ৬ হিসাবে। ৪ কে মোট দুইবার আপডেট করা লাগলো।

বিজ্ঞানী ডায়াক্সট্রা চিন্তা করলেন যদি এই “আগে আসলে আগে পাবেন” ভিত্তিতে কাজ না করে সবথেকে কাছের নোডগুলোকে আগে প্রসেস করি তাহলে অনেক কমবার আপডেট করা লাগে। আমরা যদি ২ কে নিয়ে আগে কাজ করতাম তাহলে ৩ আগেই আপডেট হয়ে যেত এবং ৪ কে একবার আপডেট করেই শর্টেস্ট ডিসটেন্স পেয়ে যেতাম! একটু হাতে কলমে সিমুলেট করে দেখো। আইডিয়াটা হলো যেকোনো সময় কিউ তে যতগুলো নোড থাকবে তাদের মধ্যে যেটা সোর্স থেকে সবথেকে কাছে সেটা নিয়ে আগে কাজ করবো। এজন্যই আমরা কিউ এর জায়গায় বসিয়ে দিবো একটি প্রায়োরিটি কিউ যে কিউতে নোড পুশ করার সাথে সাথে কাছের নোডটাকে সামনে এনে দিবে। পার্থক্য হলো আগে খালি নোড নাম্বার পুশ করেছি, এখন বর্তমান ডিসটেন্স অর্থাত d[u]d[u] এর মানটাও পুশ করতে হবে।
নিচে একটা সম্পূর্ণ ডায়াক্সট্রার সুডোকোড দিলাম যেটা ১ থেকে nn তম নোডে যাবার শর্টেস্ট পাথ বের করে এবং পাথটাও প্রিন্ট করে:

```
procedure dijkstra(G, source):
    Q = priority_queue(), distance[] = infinity
    Q.enqueue({distance[source], source})
    distance[source]=0
    while Q is not empty
       u = nodes in Q with minimum distance[]
       remove u from the Q
       for all edges from u to v in G.adjacentEdges(v) do
           if distance[u] + cost[u][v] < distance[v]
                       distance[v] = distance[u] + cost[u][v]
                       Q.enqueue({distance[v], v})
          end if
        end for
   end while
   Return distance
```

সি++ কোড দেখতে ক্লিক করো এখানে।
এখানে আগের সুডোকোডের থেকে কয়েক জায়গায় একটু ভিন্নতা আছে। এখানে সাধারণ কিউ এর জায়গায় প্রায়োরিটি কিউ ব্যবহার করা হয়েছে। কিউ থেকে পপ হবার সময় তাই সোর্স থেকে এখন পর্যন্ত পাওয়া সবথেকে কাছের নোডটা পপ হচ্ছে এবং সেটা নিয়ে আগে কাজ করছি।

উপরের সুডোকোডে সোর্স থেকে বাকি সব নোডের দূরত্ব বের করা হয়েছে। তুমি যদি শুধু একটা নোডের দুরত্ব বের করতে চাও তাহলে সেটা যখন কিউ থেকে পপ হবে তখনই রিটার্ন করে দিতে পারো।

নেগেটিভ এজ থাকলে কি ডায়াক্সট্রা অ্যালগোরিদম কাজ করবে? যদি নেগেটিভ সাইকেল থাকে তাহলে ইনফিনিট লুপে পরে যাবে, বারবার আপডেট করে কস্ট কমাতে থাকবে। যদি নেগেটিভ এজ থাকে কিন্তু সাইকেল না থাকে তাহলেও কাজ করবেনা। তবে তুমি যদি টার্গেট পপ হবার সাথে সাথে রিটার্ণ করে না দাও তাহলে কাজ করবে কিন্তু সেটা তখন আর মূল ডায়াক্সট্রা অ্যালগোরিদম থাকবেনা।

কমপ্লেক্সিটি:

বিএফএস এর কমপ্লেক্সিটি ছিলো $O(V+E)O(V+E)$ যেখানে VV হলো নোড সংখ্যা আর EE হলো এজ সংখ্যা। এখানেও আগের মতোই কাজ হবে তবে প্রায়োরিটি কিউ তে প্রতিবার সর্ট করতে $O(logV)O(logV)$ কমপ্লেক্সিটি লাগবে। মোট: $O(VlogV+E)O(VlogV+E)$ ।
নেগেটিভ সাইকেল নিয়ে কাজ করতে হলে আমাদের জানতে হবে বেলম্যান ফোর্ড অ্যালগোরিদম। সেখানেও এজ রিল্যাক্স করে আপডেট করা হয়, একটা নোডকে সর্বোচ্চ $n-1n-1$ বার আপডেট করা লাগতে পারে সেই প্রোপার্টি কাজে লাগানো হয়।
ডায়াক্সট্রা ভালো করে শিখতে নিচের প্রবলেমগুলো ঝটপট করে ফেলো:

Dijkstra?
Not the Best
New Traffic System

হ্যাপি কোডিং!