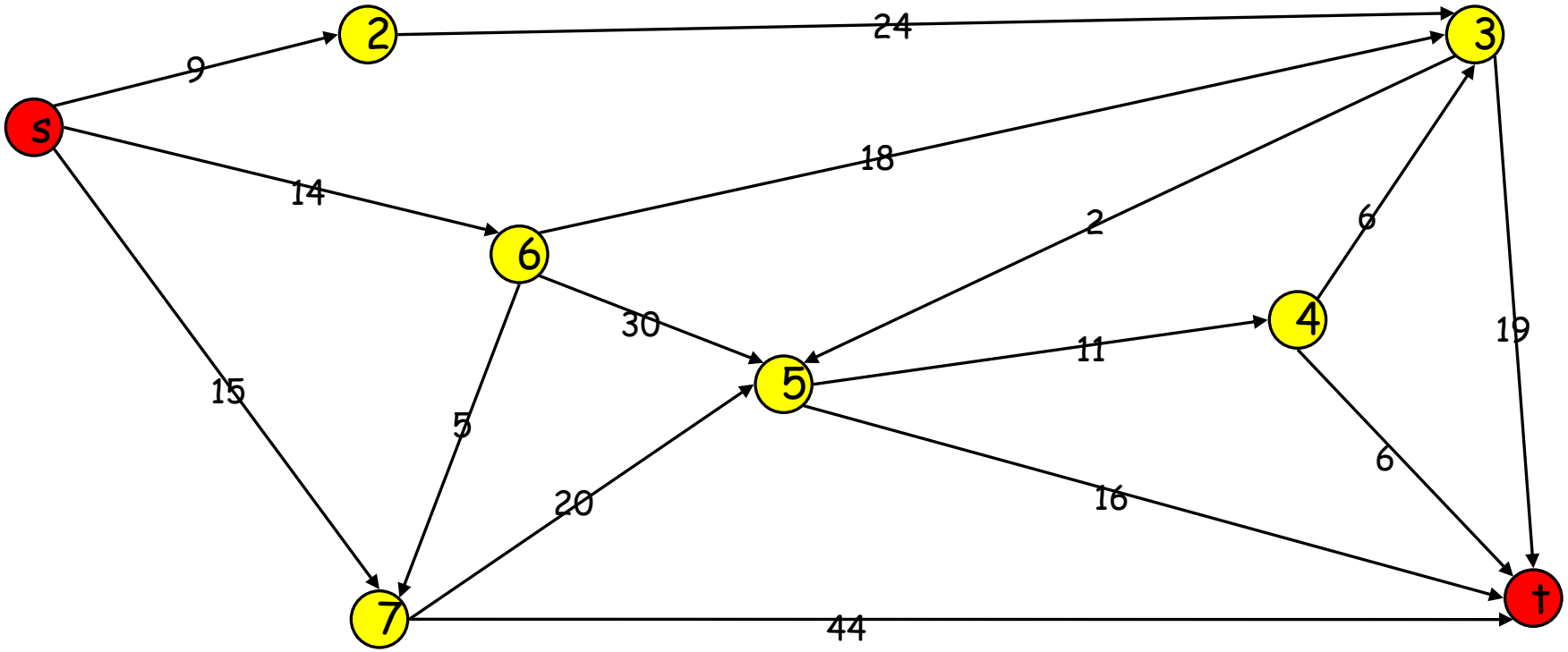
Dynamic Programming

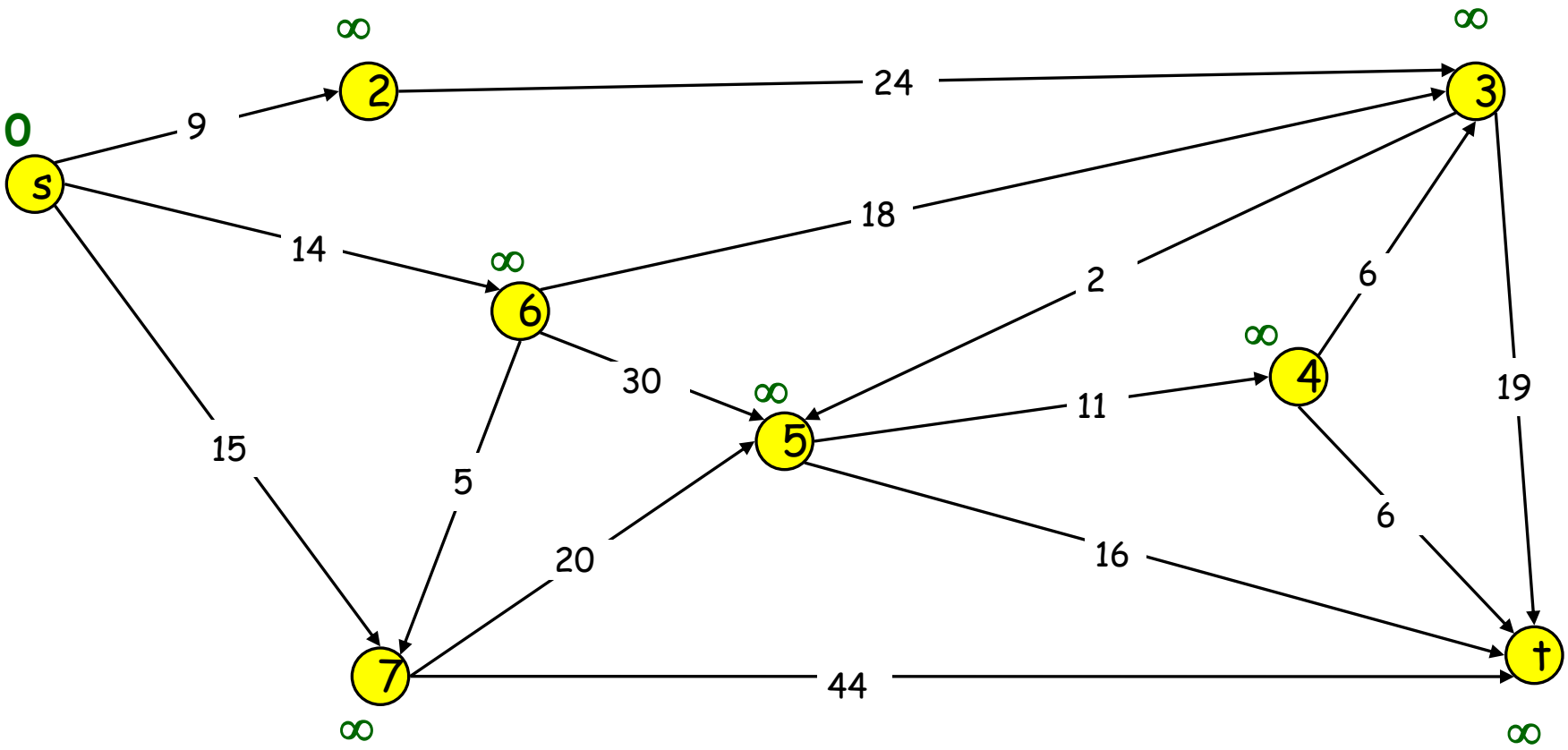
Dijkstra's Algorithm

- Goal: Find the shortest path from s to t



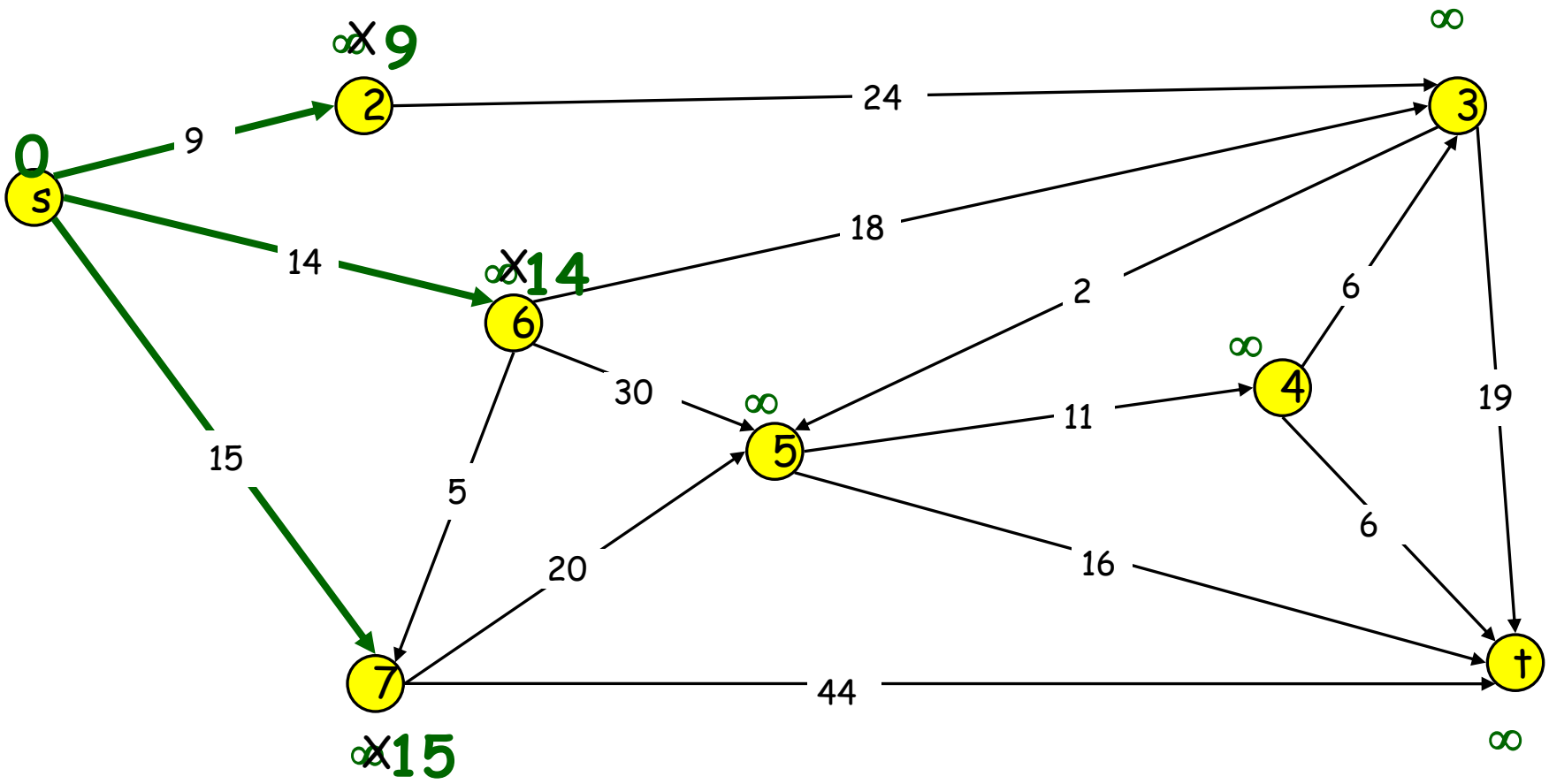
S = { }

PQ = { s, 2, 3, 4, 5, 6, 7, t }

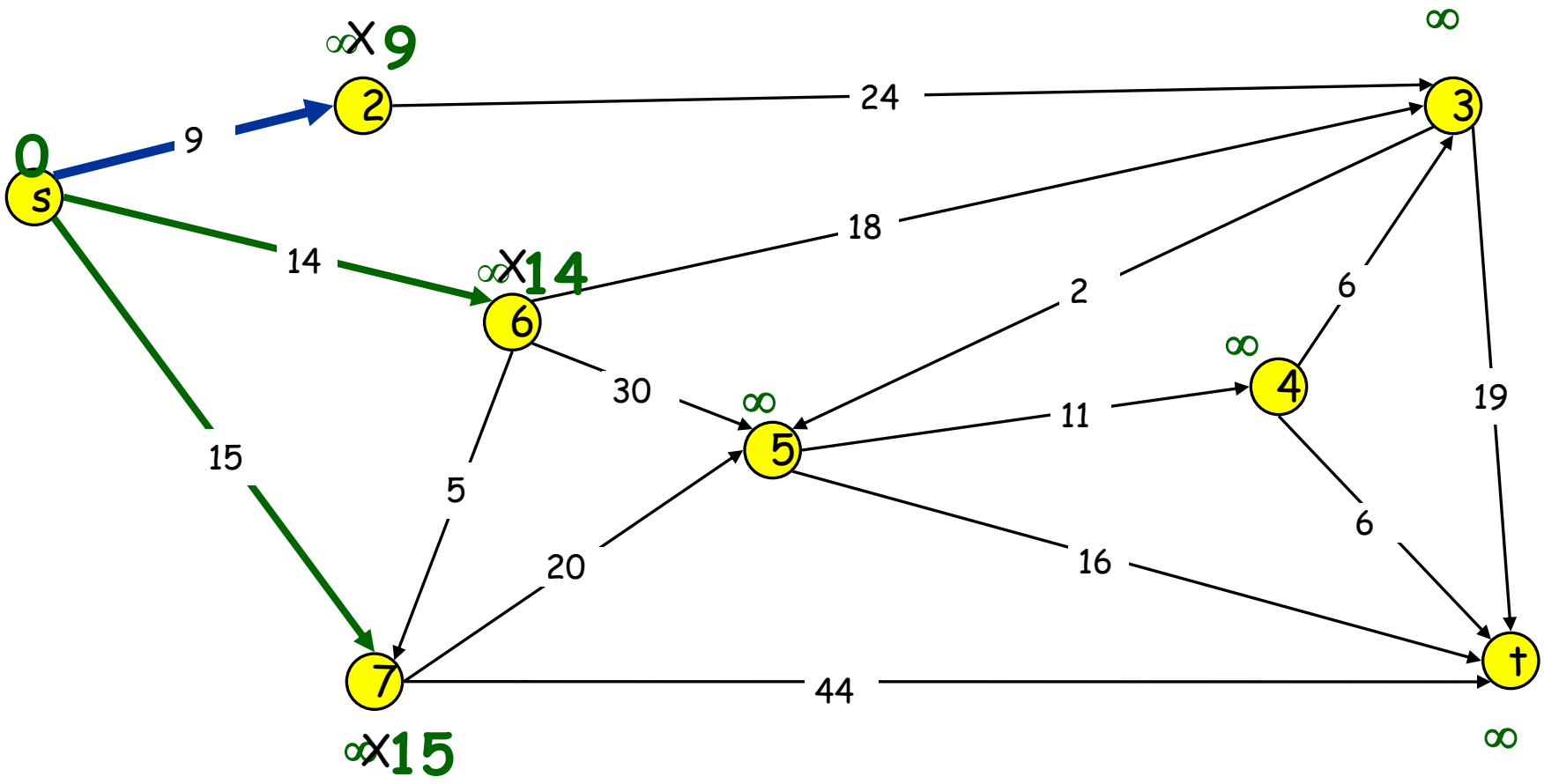


S = { s }

PQ = { 2, 3, 4, 5, 6, 7, t }

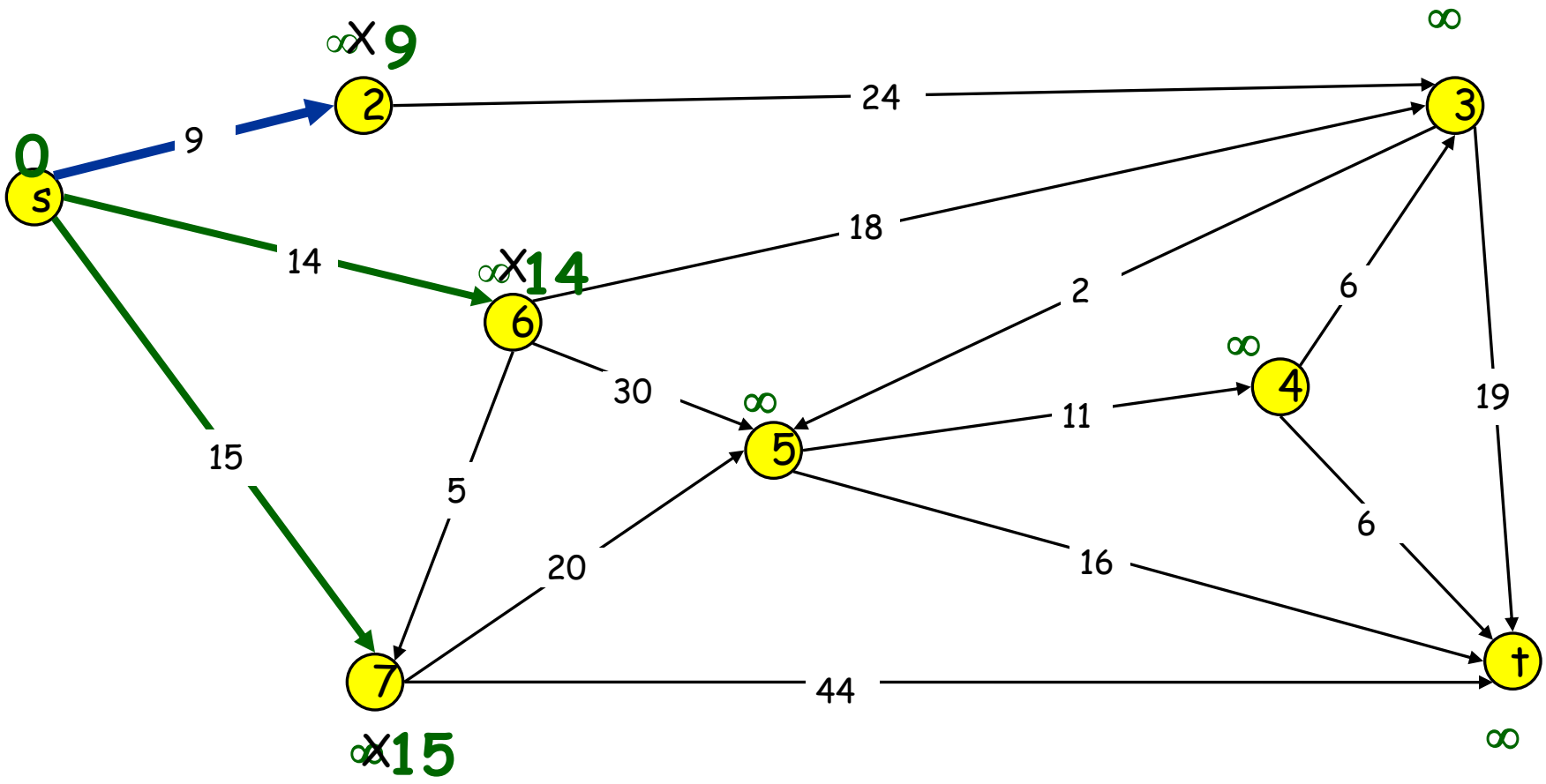


S = { s }
PQ = { 2, 3, 4, 5, 6, 7, t }



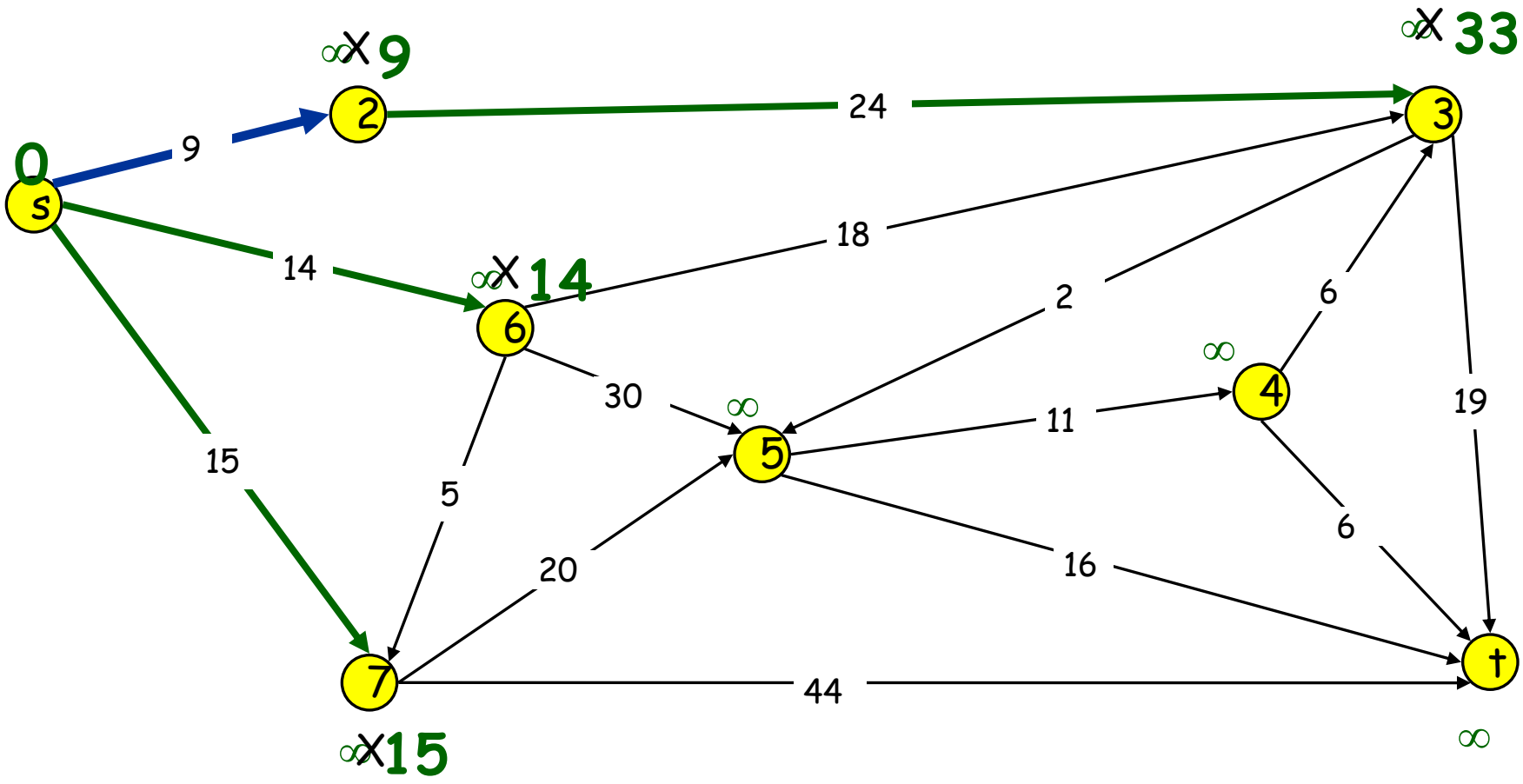
S = { s, 2 }

PQ = { 3, 4, 5, 6, 7, t }

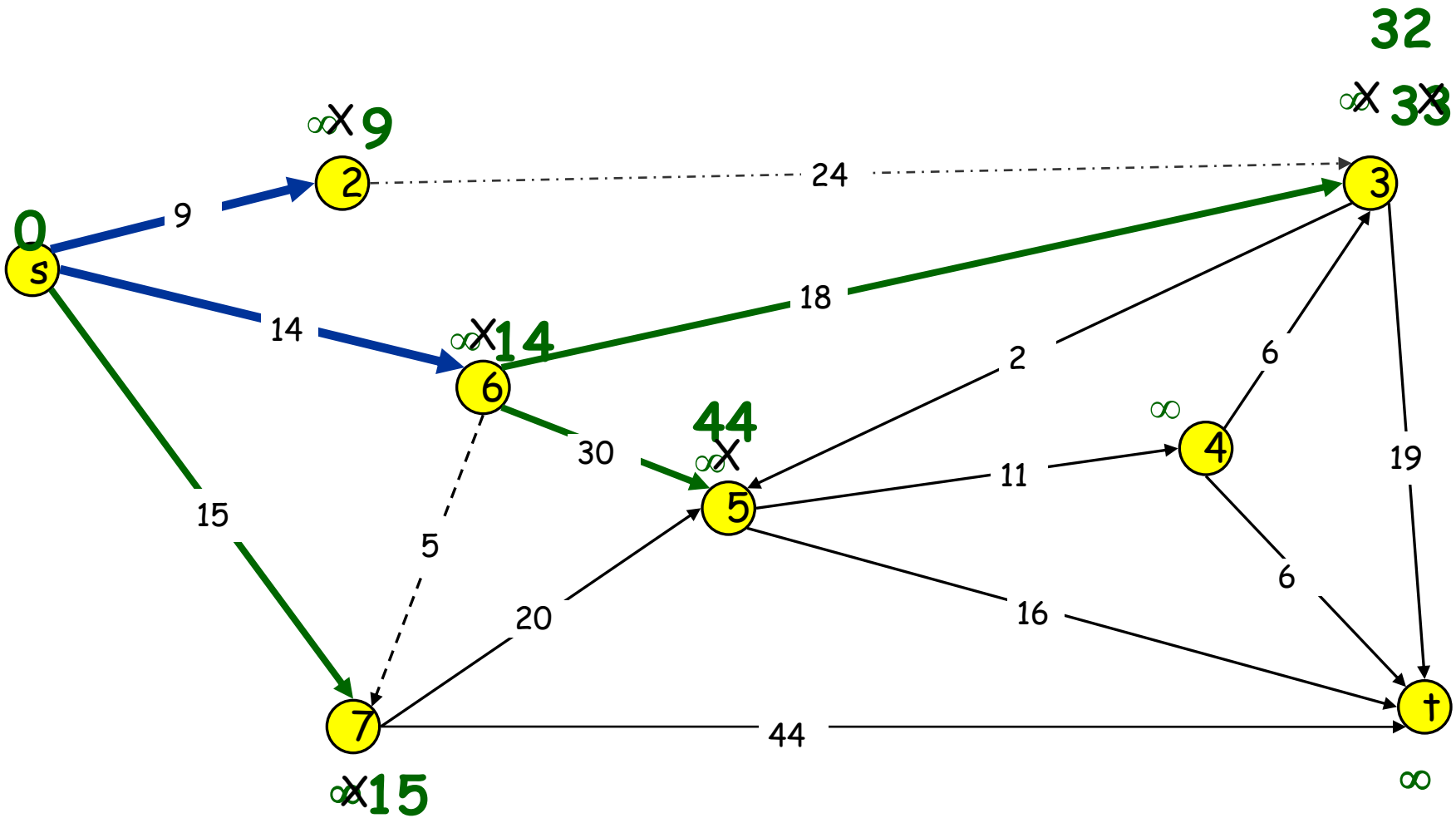


S = { s, 2 }

PQ = { 3, 4, 5, 6, 7, t }

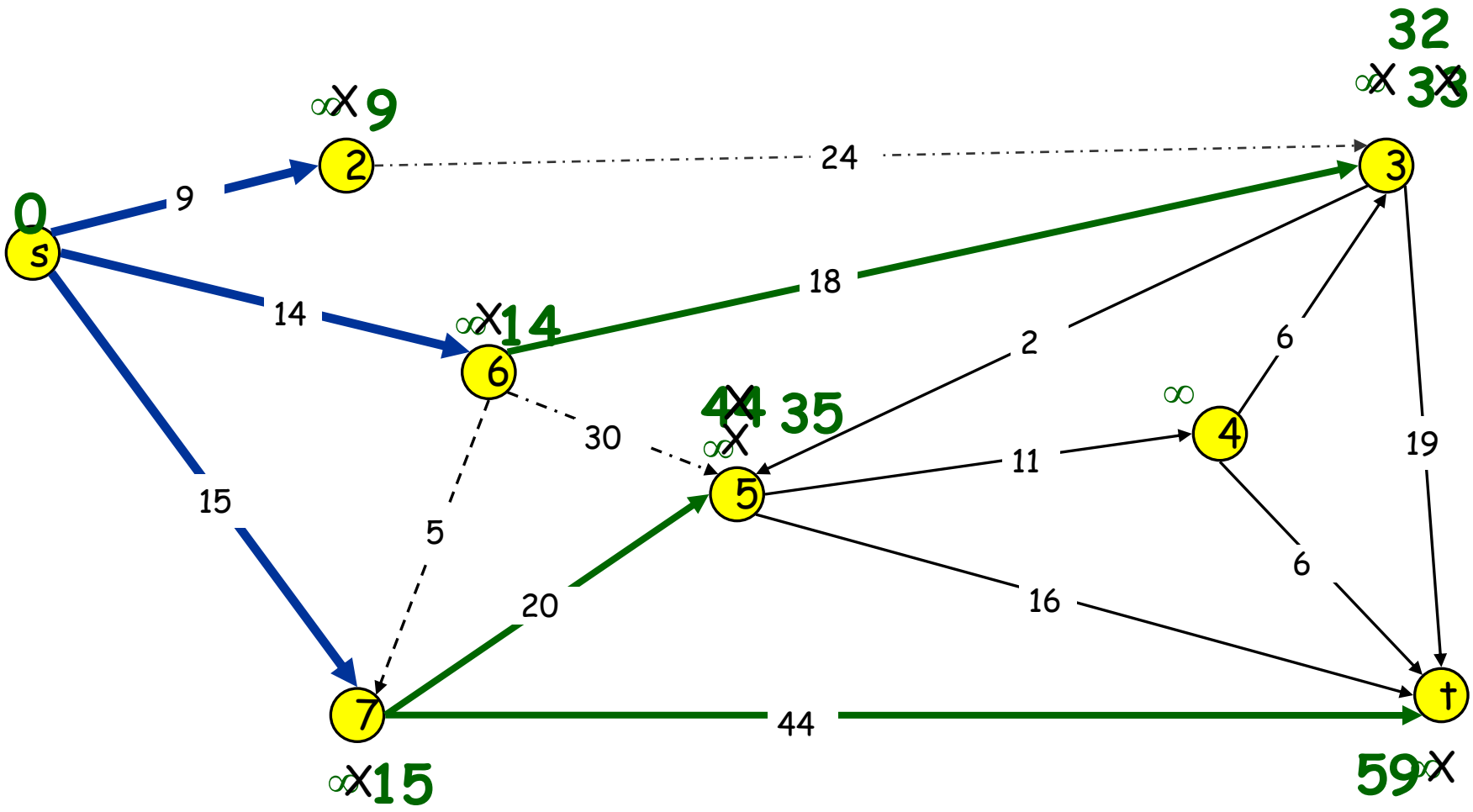


S = { s, 2, 6 }
PQ = { 3, 4, 5, 7, t }



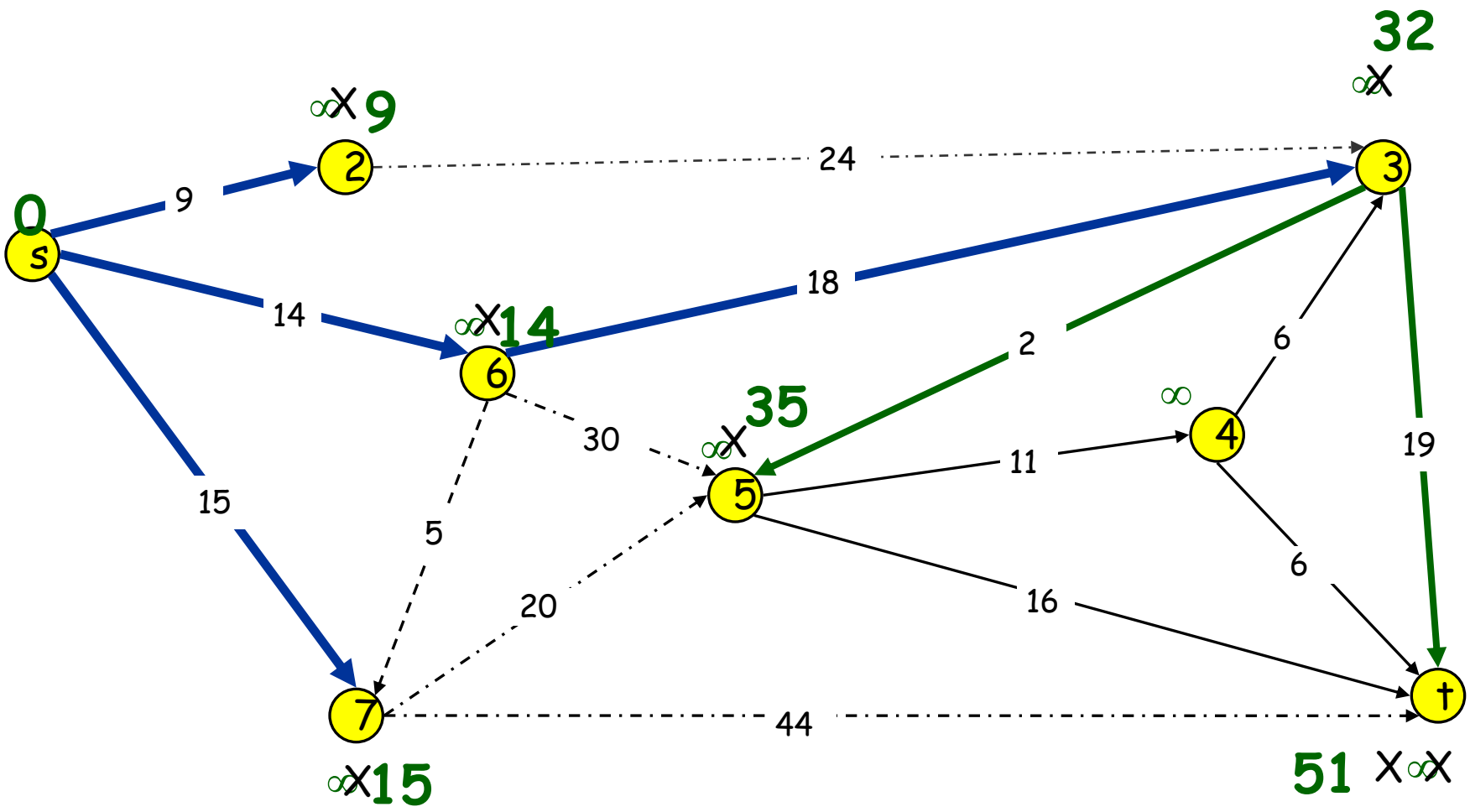
S = { s, 2, 6, 7 }

PQ = { 3, 4, 5, t }

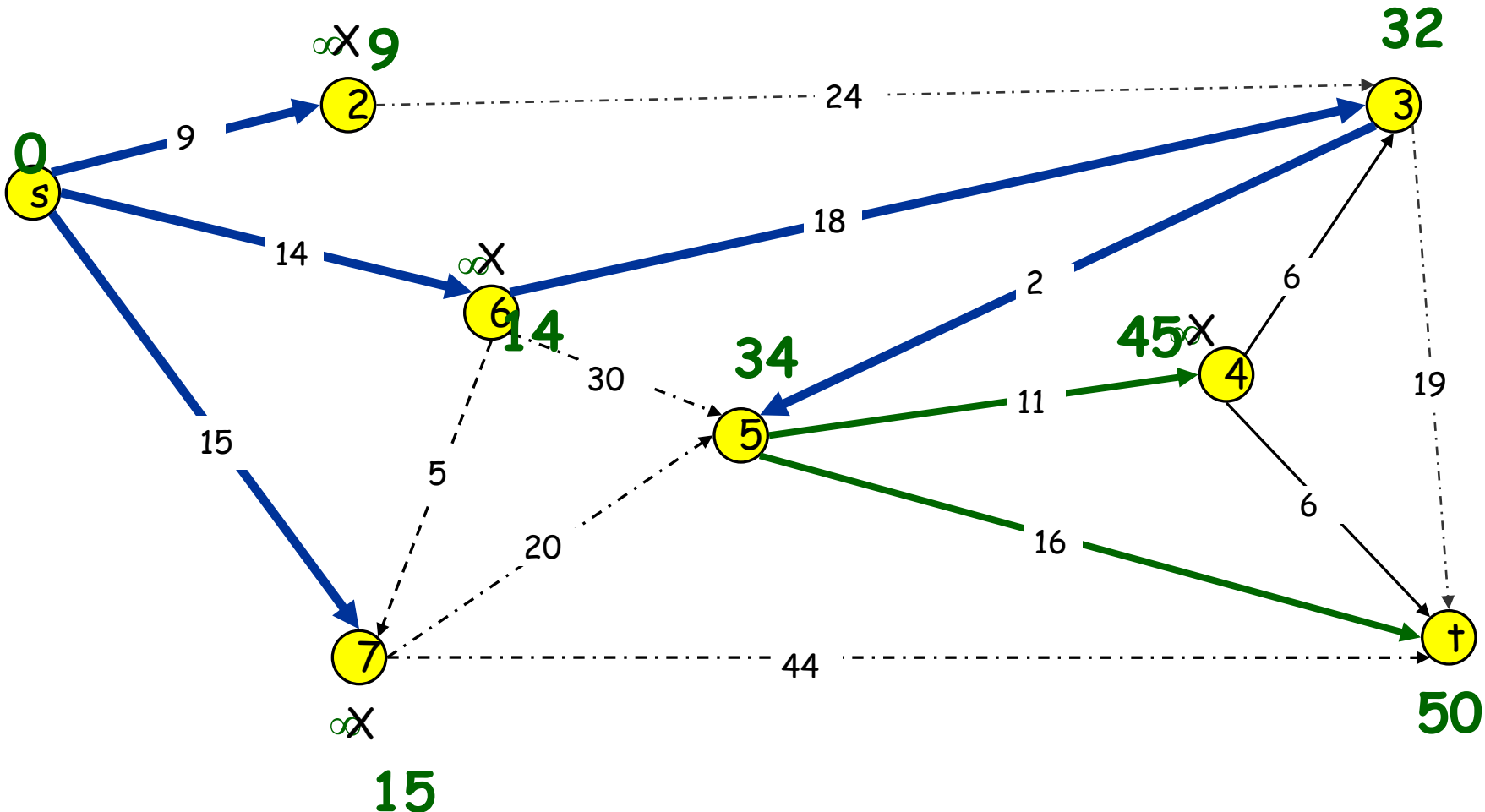


S = { s, 2, 3, 6, 7 }

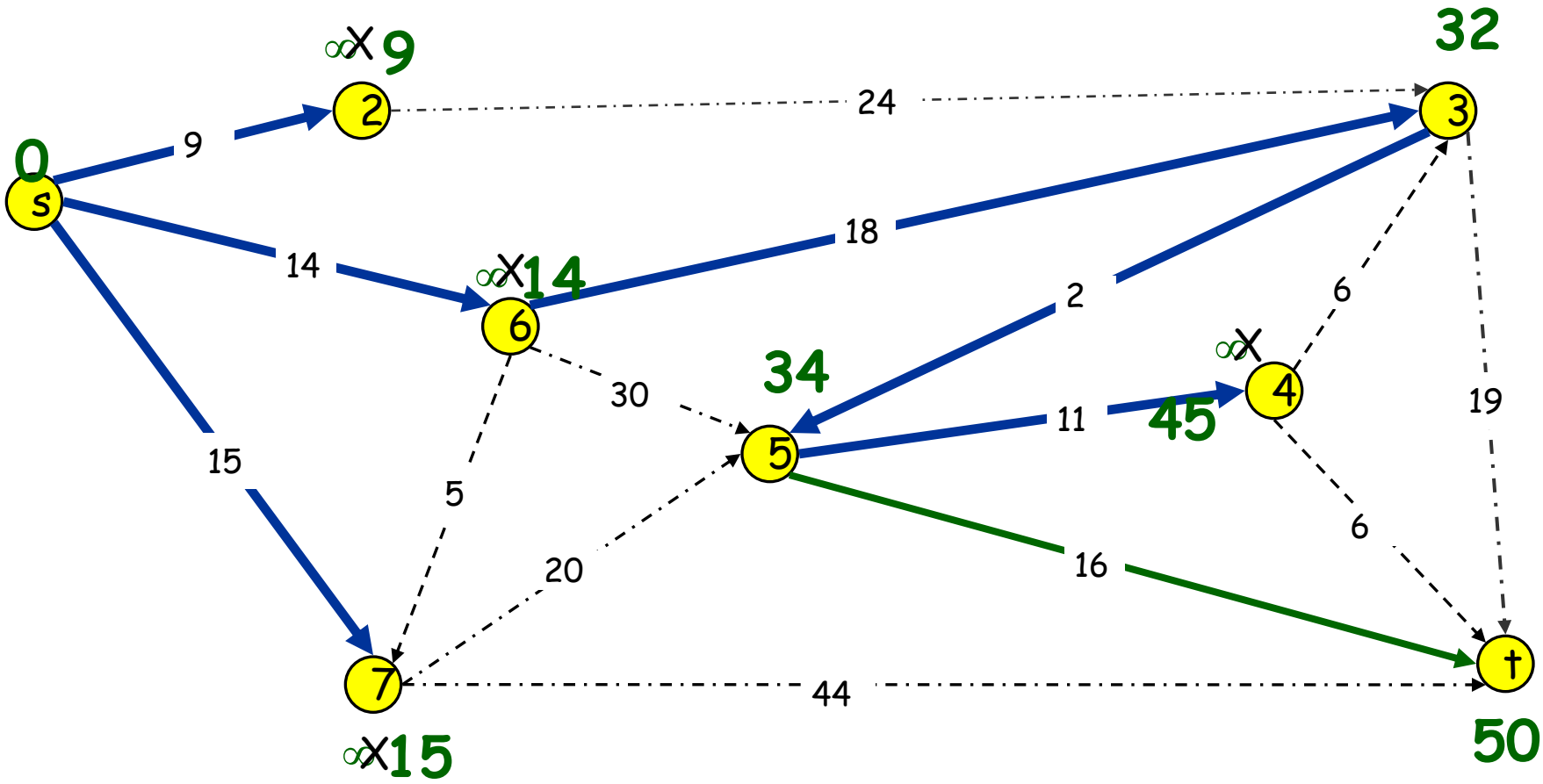
PQ = { 4, 5, t }



S = { s, 2, 3, 5, 6, 7 }
PQ = { 4, t }



S = { s, 2, 3, 4, 5, 6, 7 }
PQ = { t }



```
dijkstra(graph *g, int start)          /* WAS prim(g,start) */
{
        int i,j;                       /* counters */
        bool intree[MAXV];             /* is vertex in the tree yet? */
        int distance[MAXV];            /* vertex distance from start */
        int v;                         /* current vertex to process */
        int w;                         /* candidate next vertex */
        int weight;                    /* edge weight */
        int dist;                      /* shortest current distance */

        for (i=1; i<=g->nvertices; i++) {
                intree[i] = FALSE;
                distance[i] = MAXINT;
                parent[i] = -1;
        }

        distance[start] = 0;
        v = start;

        while (intree[v] == FALSE) {
            intree[v] = TRUE;
            for (i=0; i<g->degree[v]; i++) {
                w = g->edges[v][i].v;
                weight = g->edges[v][i].weight;
/* CHANGED */       if (distance[w] > (distance[v]+weight)) {
/* CHANGED */               distance[w] = distance[v]+weight;
                        parent[w] = v;
                }
            }

            v = 1;
            dist = MAXINT;
            for (i=2; i<=g->nvertices; i++)
                if ((intree[i]==FALSE) && (dist > distance[i])) {
                        dist = distance[i];
                        v = i;
                }
        }
}
```

Dynamic Programming

In mathematics and computer science, **dynamic programming** is a method of solving complex problems by breaking them down into simpler steps - wikipedia

Algorithms Review

- Backtracking
  - search every possible solution
  - optimal guaranteed
  - inefficient
- Greedy Algorithm
  - find the best at each stage – MST
  - not always optimal
  - needs a proof
- Dynamic Programming
  - efficient recursive algorithm

Recursive Algorithm

- Divide-and-Conquor
  - top-down solution
  - Fibonnaci
  - can be inefficient
  - duplicated computations

```
F(n) = F(n-1) + F(n-2)
     = {F(n-2)+F(n-3)} + {F(n-3)+F(n-4)}
     = [{F(n-3)+F(n-4)}+{F(n-4)+F(n-5)}]
      +[{F(n-4)+F(n-5)}+{F(n-5)+F(n-6)}] = ...
```

- Dynamic Programming
  - bottom-up
  - recursive and solve above limitations
  - Fibonacci
    - calculate F(1), F(2), ... and store the results
    - use them to calculate F(n)

Binomial Coefficient

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

```
int bino(int n, int k) {
     if (k==0 || n==k)
          return 1;
     else
          return bino(n-1, k-1) + bino(n-1, k);
}
```

{bino(n-2, k-2) + bino(n-2, k-1)} + {bino(n-2, k-1) + bino(n-2, k)}

Redundancy !

Binomial Coefficient (2)

$_kC_0 = {}_kC_k = 1$

$_2C_1 = {}_1C_0 + {}_1C_1$

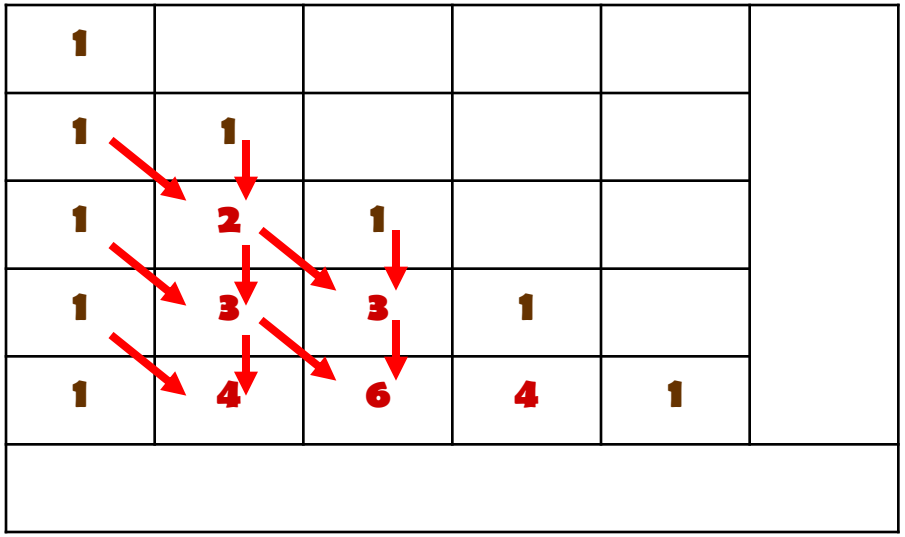
$_3C_1 = {}_2C_0 + {}_2C_1$

$_3C_2 = {}_2C_1 + {}_2C_2$

$_4C_1 = {}_3C_0 + {}_3C_1$

$_4C_2 = {}_3C_1 + {}_3C_2$

… … … …



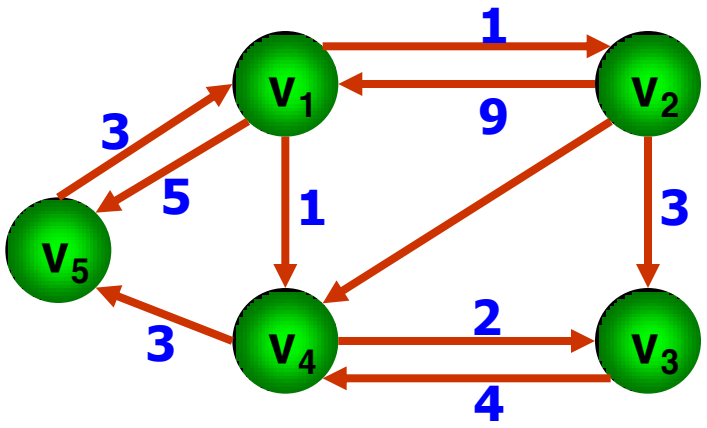

```
bino(int n, int k) {
    long bc[n][k];
    for( int i=0; i<=n; i++)
        for( int j=0; j<=min(i,k); j++ )
            if( j==0 || j==I ) bc[i][j]=1;
            else   bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
    return bc[n][k];
}
```

Floyd's Algorithm

- find all pairs shortest paths
  - apply Dijkstra alg. for each vertex
  - inefficient
- Floyd Alg.
  - number each vertex from 1 to n
  - start from $W^{(0)}=W$, ends with $W^{(n)}$ which is a set of shortest paths – bottom up
  - compute $W^{(k)}$ from $W^{(k-1)}$ - recursion
- typical dynamic programming problem

Floyd's Algorithm

- Adjacent Matrix: W



$$W[i][j] = \begin{pmatrix} - & 1 & \infty & 1 & 5 \\ 9 & - & 3 & 2 & \infty \\ \infty & \infty & - & 4 & \infty \\ \infty & \infty & 2 & - & 3 \\ 3 & \infty & \infty & \infty & - \end{pmatrix}$$

- $W^{(k)}[i][j]$
  - shortest path from v_i to v_j
  - the path traverses vertices among $\{v_1, v_2, \ldots, v_k\}$

Floyd's Algorithm

- $W^{(k)}[i][j] = \min(W^{(k-1)}[i][j], W^{(k-1)}[i][k]+W^{(k-1)}[k][j])$

  Case 1 **Case 2**

  - shortest path from v_i to v_j using $\{v_1, v_2, \dots, v_k\}$
  - Case 1: the path does not visit v_k
    - Ex) $W^{(5)}[1][3] = W^{(4)}[1][3] = 3$
  - Case 2: the path visits v_k
    - Ex) $W^{(2)}[5][3] = W^{(1)}[5][2] + W^{(1)}[2][3] = 4+3 = 7$

플로이드 알고리즘 <4>

- 데이터 구조

```
typedef struct {
        int weight[MAXV+1][MAXV+1];     /* adjacency/weight info */
        int nvertices;                  /* number of vertices in graph */
} adjacency_matrix;
```

- 초기화

```
initialize_adjacency_matrix(adjacency_matrix *g)
{
        int i,j;                                /* counters */
        g -> nvertices = 0;

        for (i=1; i<=MAXV; i++)
                for (j=1; j<=MAXV; j++)
                        g->weight[i][j] = MAXINT;
}
```

- 인접배열

```
read_adjacency_matrix(adjacency_matrix *g, bool directed)
{
        int i;                          /* counter */
        int m;                          /* number of edges */
        int x,y,w;                      /* placeholder for edge/weight */

        initialize_adjacency_matrix(g);

        scanf("%d %d\n",&(g->nvertices),&m);

        for (i=1; i<=m; i++) {
                scanf("%d %d %d\n",&x,&y,&w);
                g->weight[x][y] = w;
                if (directed==FALSE) g->weight[y][x] = w;
        }
}
```

플로이드 알고리즘 <5>

- 재귀 관계식 계산
  - $W^{(k)}[i][j] = \min(W^{(k-1)}[i][j], W^{(k-1)}[i][k]+W^{(k-1)}[k][j])$

```
floyd(adjacency_matrix *g)
{
        int i,j;                        /* dimension counters */
        int k;                          /* intermediate vertex counter */
        int through_k;                  /* distance through vertex k */
        for (k=1; k<=g->nvertices; k++)
                for (i=1; i<=g->nvertices; i++)
                        for (j=1; j<=g->nvertices; j++) {
                                through_k = g->weight[i][k]+g->weight[k][j];
                                if (through_k < g->weight[i][j])
                                        g->weight[i][j] = through_k;
                        }
}
```

String Matching

- Edit Distance
  - number of any following costs to make two strings match
    - subs – match
    - insertion
    - deletion
  - after any above operation, you are given shorter strings to match

```
#define MATCH           0       /* enumerated type symbol for match */
#define INSERT          1       /* enumerated type symbol for insert */
#define DELETE          2       /* enumerated type symbol for delete */

int string_compare(char *s, char *t, int i, int j)
{
        int k;                  /* counter */
        int opt[3];             /* cost of the three options */
        int lowest_cost;        /* lowest cost */

        if (i == 0) return(j * indel(' '));
        if (j == 0) return(i * indel(' '));

        opt[MATCH] = string_compare(s,t,i-1,j-1) + match(s[i],t[j]);
        opt[INSERT] = string_compare(s,t,i,j-1) + indel(t[j]);
        opt[DELETE] = string_compare(s,t,i-1,j) + indel(s[i]);

        lowest_cost = opt[MATCH];
        for (k=INSERT; k<=DELETE; k++)
                if (opt[k] < lowest_cost) lowest_cost = opt[k];

        return( lowest_cost );
}
```

delete remaining tail of t

Algorithm Review

- The number of function call grows exponentially
- The maximum number of distinct function call is

```
int string_compare(char *s, char *t, int i, int j)
```

- Let's make a table like ....

		y	o	u	-	s	h	o	u	l	d	-	n	o	t
:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	1	2	3	4	5	6	7	8	9	10	11	12	13	13
h:	2	2	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	2	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	3	2	3	4	5	6	5	6	7	8	9	10	11
-:	5	5	4	3	2	3	4	5	6	6	7	7	8	9	10
s:	6	6	5	4	3	2	3	4	5	6	7	8	8	9	10
h:	7	7	6	5	4	3	2	3	4	5	6	7	8	9	10
a:	8	8	7	6	5	4	3	3	4	5	6	7	8	9	10
l:	9	9	8	7	6	5	4	4	4	4	5	6	7	8	9
t:	10	10	9	8	7	6	5	5	5	5	5	6	7	8	8
-:	11	11	10	9	8	7	6	6	6	6	6	5	6	7	8
n:	12	12	11	10	9	8	7	7	7	7	7	6	5	6	7
o:	13	13	12	11	10	9	8	7	8	8	8	7	6	5	6
t:	14	14	13	12	11	10	9	8	8	9	9	8	7	6	5

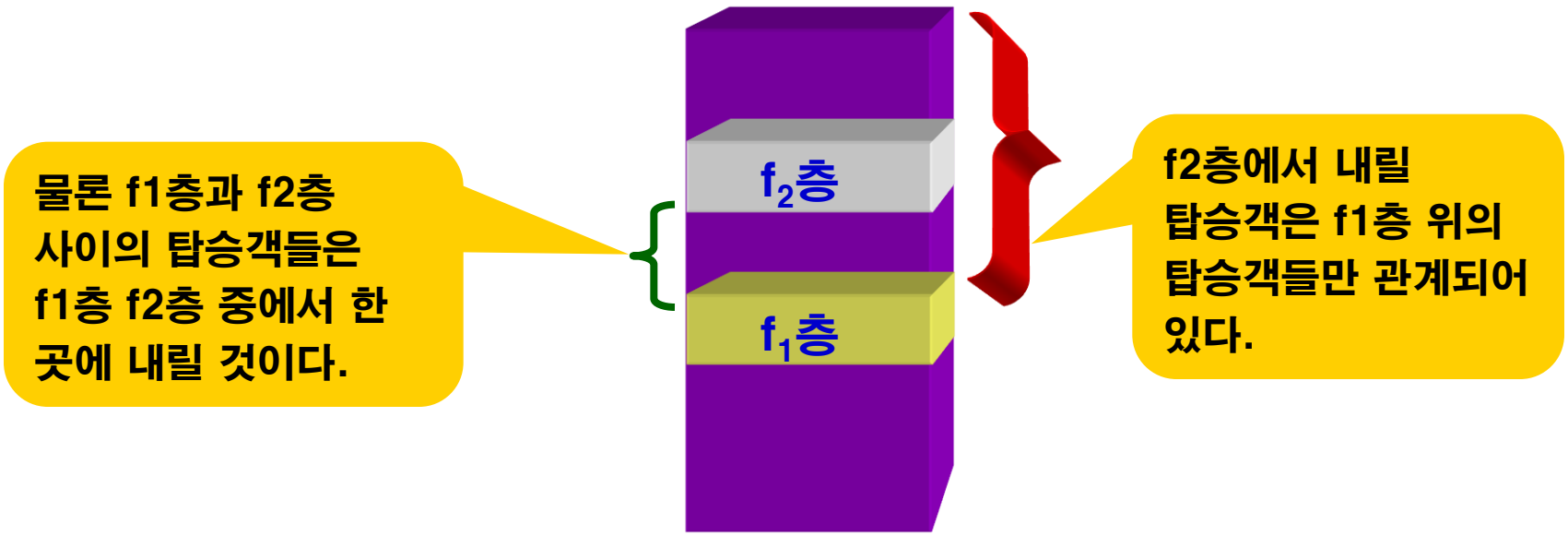
- cell[i, j] means cost to make two strings match s[0..i] and t[0..j]
- worst case: subs all chars – diagonal values are i
- computing a cell requires three cells

엘리베이터 최적화 <1>

- 엘리베이터 최적화 문제 소개
  - 모든 사람들이 입력한 층에서 엘리베이터가 멈춘다면 최악의 경우, 모든 층에서 멈춰야 하는 경우가 발생하므로, 최적화가 필요하다.
  - 탑승객들은 엘리베이터가 움직이기 전에 원하는 층을 입력한다.
  - 엘리베이터가 멈추는 횟수는 k번으로 제한되어 있다고 가정.
  - 계단을 올라가는 것과 내려가는 것 사이에 차이가 없다고 가정.
  - 걸어 올라가거나 내려가는 사람 수가 같은 층이 여러 명 존재할 경우 가장 낮은 층에 세운다.
  - 반드시 탑승객이 내리겠다고 한 위치에서만 서는 것은 아니다.
    - 예) 27층과 29층에서 내리는 탑승객이 있을 경우 28층에서 멈출 수 있음.
  - 문제의 목적은 걸어서 올라가거나 내려가야 하는 사람 수를 최소화할 수 있는 층에 멈추는 것이다.
    - 이때, 사람 수를 최소화한다는 것은 걷는 비용의 최소화를 의미한다.

엘리베이터 최적화 <2>

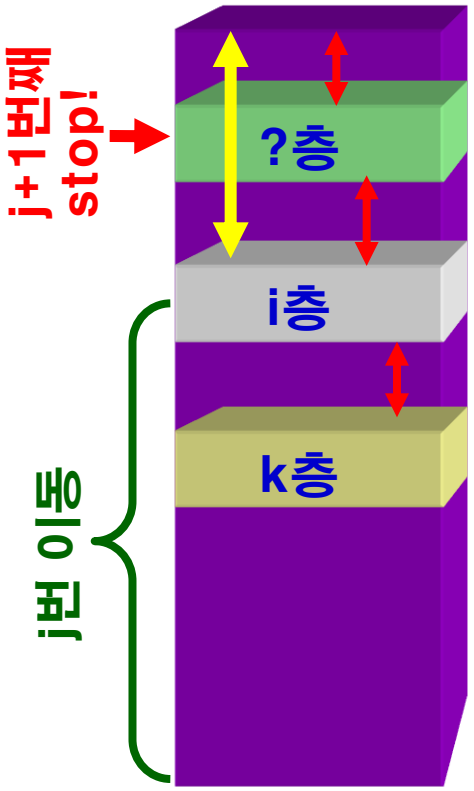
- 동적 프로그래밍을 이용한 해결 방법
  - 동적 프로그래밍은 재귀 알고리즘을 바탕으로 함을 생각하자.
  - k번째 멈출 층을 결정하는 것은 k-1번째 멈추는 모든 가능한 풀이의 비용에 따라 결정할 수 있다.
  - 우선, f1층에서 선 후에 f2층에서 서는 경우를 생각해 보자.
  - 이때, f2층은 목적지가 f1층 이하인 탑승객과는 상관이 없다.
  - 즉, 문제를 두 조각으로 나눌 수 있다.
  - 이 부분에서 동적 프로그래밍을 사용할 수 있음을 알 수 있다.



엘리베이터 최적화 <3>

$m_{i,j}$

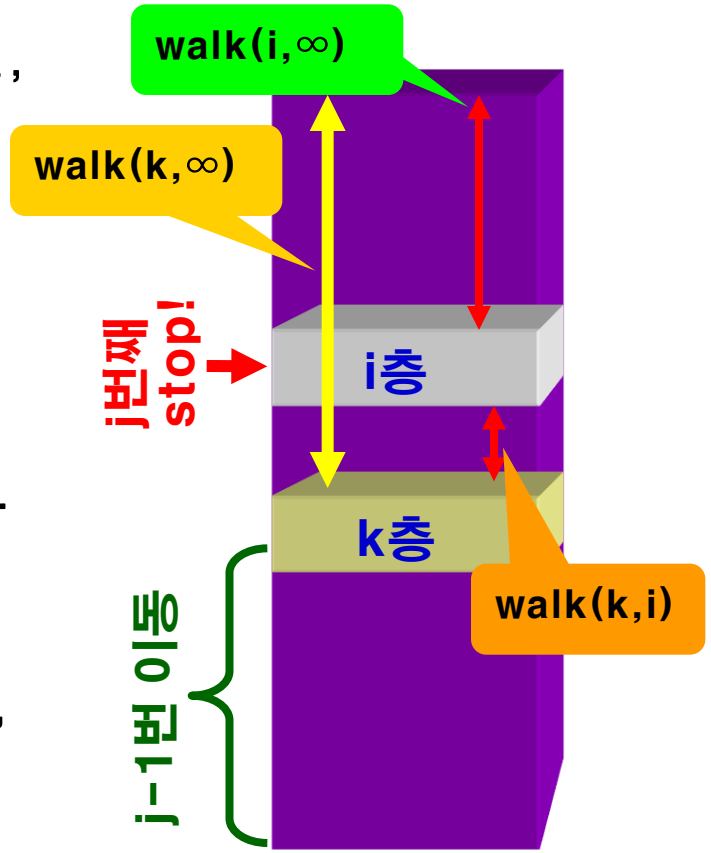
- i층 까지, j번 멈추면서 모든 승객을 운반하기 위한 최소 비용을 $m_{i,j}$으로 정의한다.
- 그렇다면, (j+1)번째 멈출 위치는 이전에 i층에서 j번째로 멈추는 층보다 높으며, 이것은 i층보다 위로 올라갈 사람에게만 유효하다.
- 즉, 어느 층이 더 가까운지를 바탕으로 새로 멈추게 될 층과 i층에서 내릴 승객으로 나눠야 한다.



엘리베이터 최적화 <3>

$$m_{i,j}=\min_{\{k=0\ldots i-1\}}\{m_{k,j-1} - walk(k,\infty)+walk(k,i)+walk(i,\infty)\}$$

- 마지막으로 (j번째) 멈춘 층이 i층이었다면, 그 전에 (j−1번째) 멈췄던 층은 i층보다 작은 어떤 k층이었다.
- 그렇다면, $m_{i,j}$은 $m_{k,j-1}$로부터 k층 위로 올라가는 모든 승객에 대한 비용(walk(k,∞)) 을 빼고, i층에 멈추는 비용(walk(k,i)+walk(i,∞))을 더하면 된다.
- 이 문제를 해결하는데 있어 가장 중요한 것은 a층에서 멈춘 다음 b층에서 멈출 때, 승객들이 걸어서 이동한 층수의 총합을 계산하는 walk(a,b) 함수이다.



Similar Problems

1. A numerical sequence is monotonically increasing if the ith element is at least as big as the (i − 1)st element. The maximum monotone subsequence problem seeks to delete the fewest number of elements from an input string S to leave a monotonically increasing subsequence. Thus a longest increasing subsequence of "243517698" is "23568."

2. Find the longest sequence of elephants whose weights are increasing but whose IQ's are decreasing.

- Can this be done as a special case of edit distance?