Chapter 18: Shortest Paths and Activity Networks

Data Structures and Algorithms with C

Brute force · Dijkstra · Predecessor array · Bellman-Ford · Floyd-Warshall · Topological order · Critical path

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/dsa_c/slides/Chapter_18_Shortest_Paths_and_Activity_Networks.html

Learning Outcomes

Model

Set up city/railway and network-delay problems as weighted directed graphs.

Dijkstra

Find shortest paths from one source to all destinations greedily; trace the $O(n^2)$ code and print paths with the predecessor array.

Extend

Apply Bellman-Ford to negative edges and Floyd-Warshall to all pairs.

Topological order

Find a linear order consistent with precedence in AoV networks using counts + a stack; detect cycles.

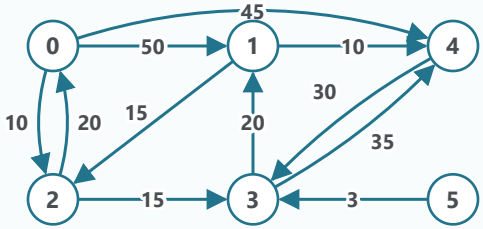
Time analysis

Compute earliest and latest start times in AoE networks by forward and backward passes.

Critical path

Compute slack and identify the critical activities and the critical path.

The Shortest Path Problem

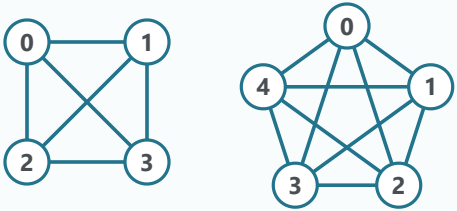


Horowitz–Sahni example

Source	Target	Cost
0	5	No path!
0	2	10
0	3	25
0	1	45
0	4	45

- Vertices are cities, edges are rail distances between them: which is the **shortest path** from one city to another?
- Or the edges are packet delays in a network (congestion): how should a **router** be written for this data?
- Two questions: 1) **Is there a path** from x to y? 2) If there are several, which is **the shortest**?
- Visiting every vertex once and returning to the start (the **traveling salesman**) is a different and much harder problem.

Single Source, Single Target: Brute Force



K4 is planar; K5 is not

K4 (0 → 3), 5 paths for n = 4: 0-1-2-3, 0-1-3, 0-2-1-3, 0-2-3, 0-3

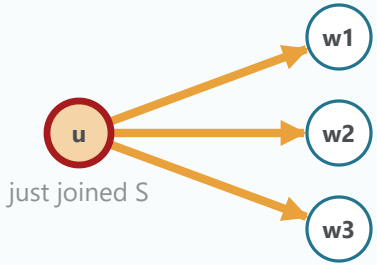
K5 (0 → 4), 1 + 3 + 6 + 6 = 16 paths for n = 5: 0-1-2-3-4, 0-1-2-4, 0-1-3-2-4, 0-1-3-4, 0-1-4, 0-2-1-3-4, 0-2-1-4, 0-2-3-1-4, 0-2-3-4, 0-2-4, 0-3-1-2-4, 0-3-1-4, 0-3-2-1-4, 0-3-2-4, 0-3-4, 0-4

Number of paths in a complete graph: **1 + (n−2) + (n−2)(n−3) + ... + (n−2)!** ≈ 1 + O(n) + O(n^2) + ... ≈ **O(n^n)** (a rough but valid model). And that is for a single target!

Instructor: "As you see, brute force costs O(n^n) even for the shortest path from one source to one target... The logic is right, of course, but this is not how we will solve it."

Toward an Efficient Solution: The Greedy Idea

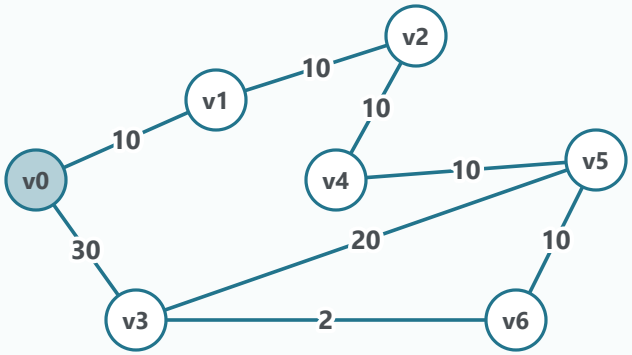
- **S**: vertices whose shortest paths are final (initially just the source 0).
- For w ∉ S, `dist[w]` is the length of the shortest path that starts at 0, passes only through vertices **in S**, and ends at w.
- Pick the u with the smallest `dist[u]` and add it to S.
- Adding u opens **new paths** through u: if `dist[u] + cost(<u,w>) < dist[w]`, update dist[w] (no intermediate vertex between u and w).



new paths to u's neighbors through u

This is Edsger Dijkstra's algorithm (1959). Edge weights **must not be negative** (see Bellman-Ford).

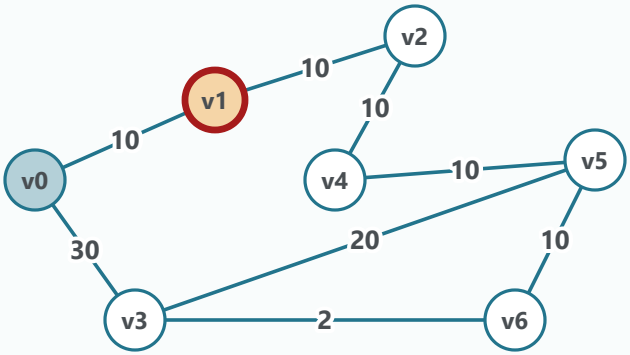
Step by Step: From v0 to All Vertices



	found	distance
v0	T	0
v1	F	10
v2	F	∞
v3	F	30
v4	F	∞
v5	F	∞
v6	F	∞

S = {v0}. The neighbors' distances are the edge costs: v1 = 10, v3 = 30; the rest are ∞.

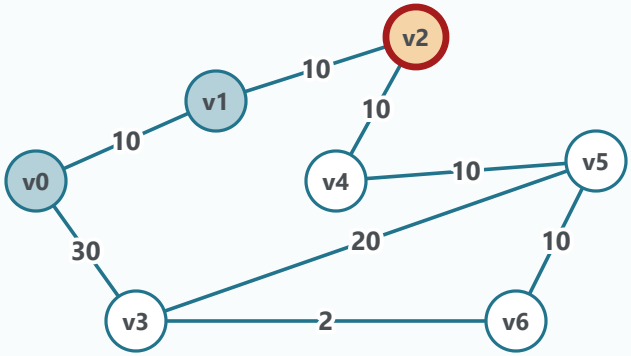
Step by Step: From v0 to All Vertices



	found	distance
v0	T	0
v1	T	10
v2	F	20
v3	F	30
v4	F	∞
v5	F	∞
v6	F	∞

v1 (10) has the smallest distance and joins S. Updated: v2 = 20.

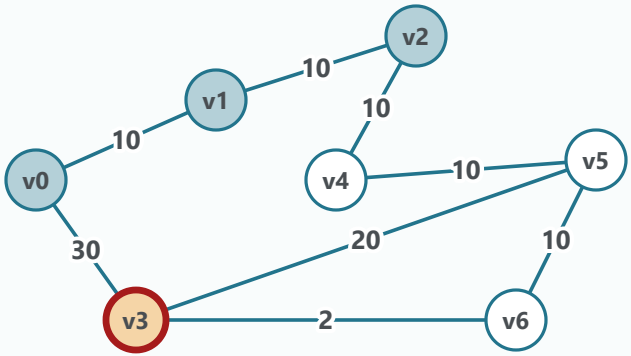
Step by Step: From v0 to All Vertices



	found	distance
v0	T	0
v1	T	10
v2	T	20
v3	F	30
v4	F	30
v5	F	∞
v6	F	∞

v2 (20) has the smallest distance and joins S. Updated: v4 = 30.

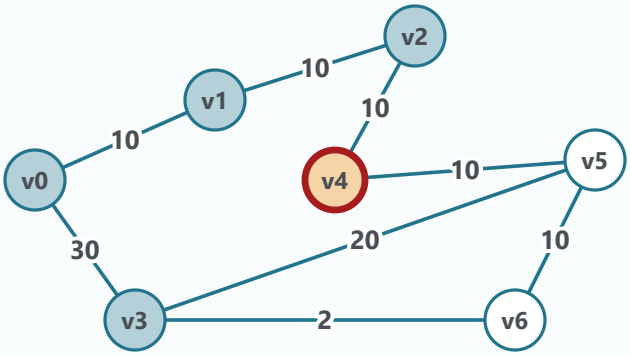
Step by Step: From v0 to All Vertices



	found	distance
v0	T	0
v1	T	10
v2	T	20
v3	T	30
v4	F	30
v5	F	50
v6	F	32

v3 (30) has the smallest distance and joins S. Updated: v5 = 50, v6 = 32.

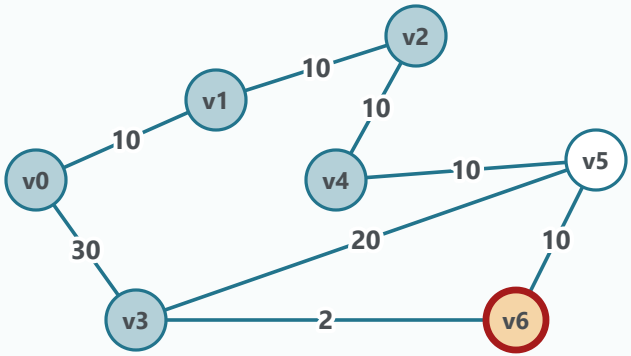
Step by Step: From v0 to All Vertices



	found	distance
v0	T	0
v1	T	10
v2	T	20
v3	T	30
v4	T	30
v5	F	40
v6	F	32

v4 (30) has the smallest distance and joins S. Updated: v5 = 40. A shorter path to v5 through v4 was found (50 → 40)!

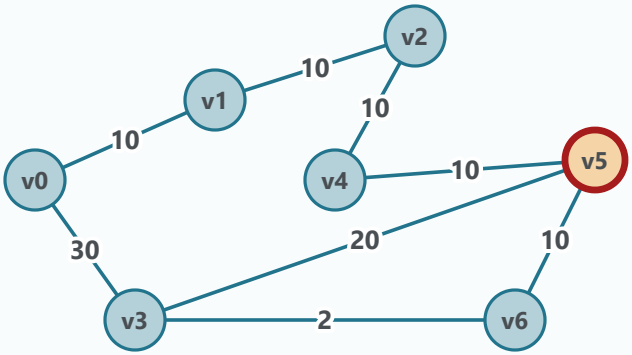
Step by Step: From v0 to All Vertices



	found	distance
v0	T	0
v1	T	10
v2	T	20
v3	T	30
v4	T	30
v5	F	40
v6	T	32

v6 (32) has the smallest distance and joins S. No updates. The order depends on distance, not on vertex number; 32 < 40, so **v6** comes first.

Step by Step: From v0 to All Vertices



	found	distance
v0	T	0
v1	T	10
v2	T	20
v3	T	30
v4	T	30
v5	T	40
v6	T	32

v5 (40) has the smallest distance and joins S. No updates.

C Code: The Cost Matrix and choose

```c
#include <limits.h>
#define MAX_VERTICES 6
#define INF 1000          /* no edge */
int cost[][MAX_VERTICES] = {
    {   0,   50,  10, 1000,   45, 1000},
    {1000,    0,  15, 1000,   10, 1000},
    {  20, 1000,   0,   15, 1000, 1000},
    {1000,   20, 1000,   0,   35, 1000},
    {1000, 1000, 1000,  30,    0, 1000},   /* missing row */
    {1000, 1000, 1000,   3, 1000,    0}};  /* 100 -> 1000 */
int choose(int dist[], int n, short found[])
{
    int i, min = INT_MAX, pos = -1;
    for (i = 0; i < n; i++)
        if (dist[i] < min && !found[i]) {
            min = dist[i];  pos = i;
        }
    return pos;
}
```

Details:

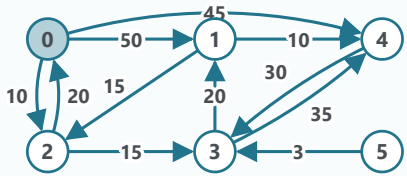
- Write `!found[w]`; `!found(w)` calls the array as if it were a function.
- `<limits.h>` for `INT_MAX`; FALSE/TRUE must be defined.
- `choose` may return −1 (no unmarked vertex left); the caller must check. An unreachable vertex can also be chosen: `INF` (1000) < `INT_MAX`.

shortest_path(): The Main Function

```
void shortest_path(int v, int cost[][MAX_VERTICES],
                   int dist[], int pred[],
                   int n, short found[])
{
    int i, u, w;
    for (i = 0; i < n; i++) {                  /* A1 */
        found[i] = FALSE;
        dist[i] = cost[v][i];
        pred[i] = v;                /* kept for the path */
    }
    found[v] = TRUE;  dist[v] = 0;
    for (i = 2; i < n; i++) {         /* n-2 rounds */
        u = choose(dist, n, found);
        if (u < 0) break;             /* none left */
        found[u] = TRUE;
        for (w = 0; w < n; w++)
            if (!found[w] &&
                dist[u] + cost[u][w] < dist[w]) {
                dist[w] = dist[u] + cost[u][w];
                pred[w] = u;
            }
    }
}
```

- **A1:** initial values; the distances are the direct edge costs.
- **A2:** the outer loop runs n − 2 rounds: the last remaining vertex's distance is already final.
- **A3:** `choose` is O(n).
- **A4:** *relaxation* for every w, O(n).
- `pred[]` is kept to trace the path back.

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	50	10	∞	45	∞
found	T	F	F	F	F	F
pred	0	0	0	0	0	0

After A1: dist = row 0 of the matrix (1000 = ∞); pred[] = 0. The source is marked.

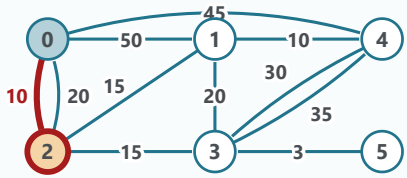
Step 1 / 10

```
▶ found[v] = TRUE;  dist[v] = 0;
2 for (i = 2; i < n; i++) {            /* n-2 rounds */
3     u = choose(dist, n, found);
4     if (u < 0) break;                /* none left */
5     found[u] = TRUE;
6     for (w = 0; w < n; w++)
7         if (!found[w] &&
8             dist[u] + cost[u][w] < dist[w]) {
9             dist[w] = dist[u] + cost[u][w];
10            pred[w] = u;
11        }
12 }
```

VARIABLES

v = 0 | u = –

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	50	10	∞	45	∞
found	T	F	T	F	F	F
pred	0	0	0	0	0	0

`choose` : the smallest distance among unmarked vertices is **2** (10). Final.

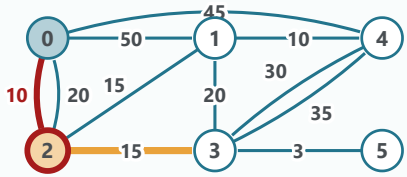
Step 2 / 10

```
1  found[v] = TRUE;  dist[v] = 0;
2  for (i = 2; i < n; i++) {              /* n-2 rounds */
▶      u = choose(dist, n, found);
4      if (u < 0) break;                  /* none left */
▶      found[u] = TRUE;
6      for (w = 0; w < n; w++)
7          if (!found[w] &&
8              dist[u] + cost[u][w] < dist[w]) {
9              dist[w] = dist[u] + cost[u][w];
10             pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = 2` `dist[u] = 10`

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	50	10	25	45	∞
found	T	F	T	F	F	F
pred	0	0	0	2	0	0

w = 3: 10 + 15 = 25 < old value → `dist[3] = 25`, `pred[3] = 2`.

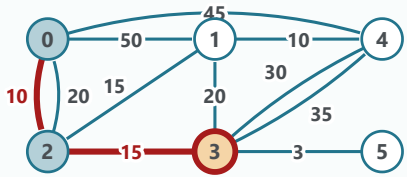
Step 3 / 10

```
1  found[v] = TRUE;  dist[v] = 0;
2  for (i = 2; i < n; i++) {            /* n-2 rounds */
3      u = choose(dist, n, found);
4      if (u < 0) break;                /* none left */
5      found[u] = TRUE;
6      for (w = 0; w < n; w++)
▶          if (!found[w] &&
▶              dist[u] + cost[u][w] < dist[w]) {
▶              dist[w] = dist[u] + cost[u][w];
▶              pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = 2` `w = 3` `dist[u] = 10`

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	50	10	25	45	∞
found	T	F	T	T	F	F
pred	0	0	0	2	0	0

`choose` : the smallest distance among unmarked vertices is **3** (25). Final.

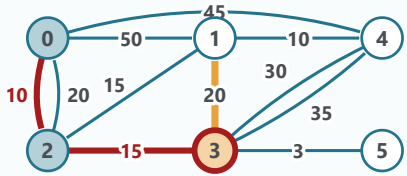
Step 4 / 10

```
1  found[v] = TRUE;  dist[v] = 0;
2  for (i = 2; i < n; i++) {            /* n-2 rounds */
▶      u = choose(dist, n, found);
4      if (u < 0) break;                /* none left */
▶      found[u] = TRUE;
6      for (w = 0; w < n; w++)
7          if (!found[w] &&
8              dist[u] + cost[u][w] < dist[w]) {
9              dist[w] = dist[u] + cost[u][w];
10             pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = 3` `dist[u] = 25`

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	45	10	25	45	∞
found	T	F	T	T	F	F
pred	0	3	0	2	0	0

w = 1: 25 + 20 = 45 < old value → `dist[1] = 45`, `pred[1] = 3`.

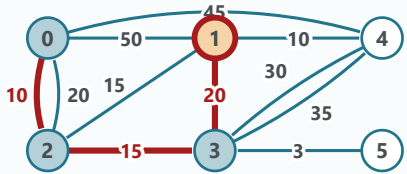
Step 5 / 10

```
 1 found[v] = TRUE;  dist[v] = 0;
 2 for (i = 2; i < n; i++) {            /* n-2 rounds */
 3     u = choose(dist, n, found);
 4     if (u < 0) break;                /* none left */
 5     found[u] = TRUE;
 6     for (w = 0; w < n; w++)
▶          if (!found[w] &&
▶              dist[u] + cost[u][w] < dist[w]) {
▶              dist[w] = dist[u] + cost[u][w];
▶              pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = 3` `w = 1` `dist[u] = 25`

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	45	10	25	45	∞
found	T	T	T	T	F	F
pred	0	3	0	2	0	0

`choose` : the smallest distance among unmarked vertices is **1** (45); 1 and 4 tie at 45, and since the test is `<` the lower index 1 wins. Final.

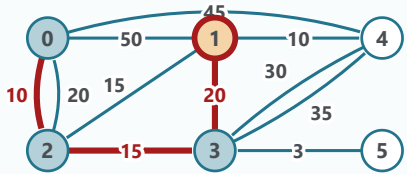
Step 6 / 10

```
1  found[v] = TRUE;  dist[v] = 0;
2  for (i = 2; i < n; i++) {            /* n-2 rounds */
▶      u = choose(dist, n, found);
4      if (u < 0) break;                /* none left */
▶      found[u] = TRUE;
6      for (w = 0; w < n; w++)
7          if (!found[w] &&
8              dist[u] + cost[u][w] < dist[w]) {
9              dist[w] = dist[u] + cost[u][w];
10             pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = 1` `dist[u] = 45`

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	45	10	25	45	∞
found	T	T	T	T	F	F
pred	0	3	0	2	0	0

No distance gets shorter through 1 (e.g. 45 + 10 = 55 > 45).

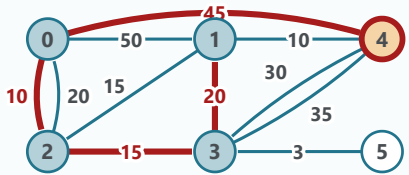
Step 7 / 10

```
1  found[v] = TRUE;  dist[v] = 0;
2  for (i = 2; i < n; i++) {            /* n-2 rounds */
3      u = choose(dist, n, found);
4      if (u < 0) break;                /* none left */
5      found[u] = TRUE;
▶      for (w = 0; w < n; w++)
▶          if (!found[w] &&
8              dist[u] + cost[u][w] < dist[w]) {
9              dist[w] = dist[u] + cost[u][w];
10             pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = 1` `dist[u] = 45`

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	45	10	25	45	∞
found	T	T	T	T	T	F
pred	0	3	0	2	0	0

`choose` : the smallest distance among unmarked vertices is **4** (45). Final.

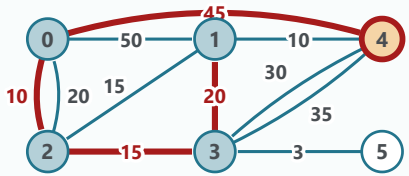
Step 8 / 10

```
1  found[v] = TRUE;  dist[v] = 0;
2  for (i = 2; i < n; i++) {            /* n-2 rounds */
▶      u = choose(dist, n, found);
4      if (u < 0) break;                /* none left */
▶      found[u] = TRUE;
6      for (w = 0; w < n; w++)
7          if (!found[w] &&
8              dist[u] + cost[u][w] < dist[w]) {
9              dist[w] = dist[u] + cost[u][w];
10             pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = 4` `dist[u] = 45`

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	45	10	25	45	∞
found	T	T	T	T	T	F
pred	0	3	0	2	0	0

No distance gets shorter through 4 (its only out-edge is <4,3> and 3 is already marked).

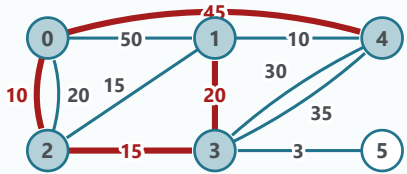
Step 9 / 10

```
1  found[v] = TRUE;  dist[v] = 0;
2  for (i = 2; i < n; i++) {            /* n-2 rounds */
3      u = choose(dist, n, found);
4      if (u < 0) break;                /* none left */
5      found[u] = TRUE;
▶      for (w = 0; w < n; w++)
▶          if (!found[w] &&
8              dist[u] + cost[u][w] < dist[w]) {
9              dist[w] = dist[u] + cost[u][w];
10             pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = 4` `dist[u] = 45`

Code Trace: `shortest_path(0, …)`



	0	1	2	3	4	5
dist	0	45	10	25	45	∞
found	T	T	T	T	T	F
pred	0	3	0	2	0	0

n − 2 = 4 rounds done. Result: 2 → 10, 3 → 25, 1 → 45, 4 → 45; **no path** to 5 (still 1000).

Step 10 / 10

```
1  found[v] = TRUE;  dist[v] = 0;
▶ for (i = 2; i < n; i++) {            /* n-2 rounds */
3      u = choose(dist, n, found);
4      if (u < 0) break;               /* none left */
5      found[u] = TRUE;
6      for (w = 0; w < n; w++)
7          if (!found[w] &&
8              dist[u] + cost[u][w] < dist[w]) {
9              dist[w] = dist[u] + cost[u][w];
10             pred[w] = u;
11         }
12 }
```

VARIABLES

`v = 0` `u = –`

Analysis, and Better Structures

The textbook algorithm

$n + n[n + n] = n + 2n^2 =$ **$O(n^2)$**, from A1, A2, A3 and A4 in turn.

Min priority queue

Keep the distances in a min heap: **log n** instead of n in A3. With adjacency lists, **degree(v)** instead of n in A4.

Improved total

A common mistake is to say O(n lg n); every relaxation also updates the heap, so the right bound is **O((n + e) log n)** (Fibonacci heap: O(e + n log n)).

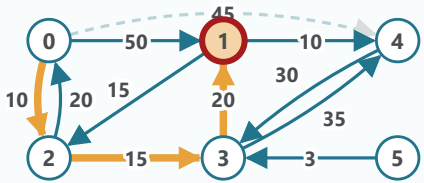
Three algorithms

Brute force $O(n^n)$ · matrix $O(n^2)$ · heap + lists O((n + e) log n). On dense graphs the $O(n^2)$ version is still good.

Instructor: "Once you know min priority queues, searching an array for the smallest element looks so clumsy, doesn't it? How absurd! How embarrassing!"

"By the time he finds the largest element in the matrix, I have the shortest paths from v0 to every vertex."

Code Trace: `print_path(pred, 0, 1)`



pred[]

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

1 ← 3 ← 2 ← 0 (traced backward)

h = 1 ≠ 0: print the predecessor's path first → `print_path(…, 3)`.

Step 1 / 8

```
1 void print_path(int pred[], int s, int h)
2 {
▶       if (h == s) {
4           printf("%d", s);
5           return;
6       }
▶       print_path(pred, s, pred[h]);   /* pred first */
8       printf(" -> %d", h);
9 }
```

VARIABLES

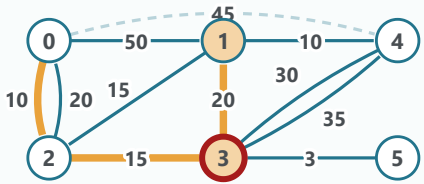
`s = 0` `h = 1` `pred[h] = 3`

CALL STACK

`print_path(1)` `main()`

OUTPUT

Code Trace: `print_path(pred, 0, 1)`



pred[]

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

1 ← 3 ← 2 ← 0 (traced backward)

h = 3 ≠ 0: print the predecessor's path first → `print_path(…, 2)`.

Step 2 / 8

```
1 void print_path(int pred[], int s, int h)
2 {
▶       if (h == s) {
4           printf("%d", s);
5           return;
6       }
▶       print_path(pred, s, pred[h]);   /* pred first */
8       printf(" -> %d", h);
9 }
```

VARIABLES

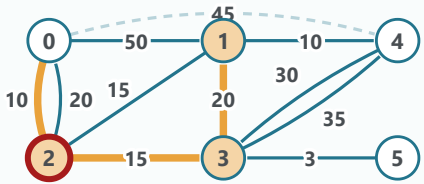
`s = 0` `h = 3` `pred[h] = 2`

CALL STACK

`print_path(3)` `print_path(1)` `main()`

OUTPUT

Code Trace: `print_path(pred, 0, 1)`



pred[]

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

1 ← 3 ← 2 ← 0 (traced backward)

h = 2 ≠ 0: print the predecessor's path first → `print_path(…, 0)`.

Step 3 / 8

```
1 void print_path(int pred[], int s, int h)
2 {
▶       if (h == s) {
4           printf("%d", s);
5           return;
6       }
▶       print_path(pred, s, pred[h]);   /* pred first */
8       printf(" -> %d", h);
9 }
```

VARIABLES

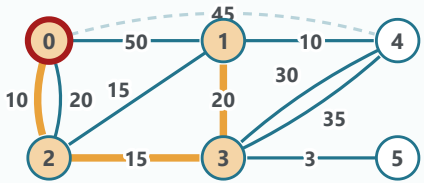
`s = 0` `h = 2` `pred[h] = 0`

CALL STACK

`print_path(2)` `print_path(3)` `print_path(1)` `main()`

OUTPUT

Code Trace: `print_path(pred, 0, 1)`



pred[]

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

1 ← 3 ← 2 ← 0 (traced backward)

h = s = 0: the source is reached → print **0** and return.

Step 4 / 8

```
1 void print_path(int pred[], int s, int h)
2 {
▶       if (h == s) {
▶           printf("%d", s);
5           return;
6       }
7       print_path(pred, s, pred[h]);   /* pred first */
8       printf(" -> %d", h);
9 }
```

VARIABLES

`s = 0` `h = 0`

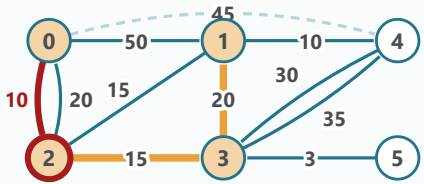
CALL STACK

`print_path(0)` `print_path(2)` `print_path(3)` `print_path(1)` `main()`

OUTPUT

```
0
```

Code Trace: `print_path(pred, 0, 1)`



pred[]

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

1 ← 3 ← 2 ← 0 (traced backward)

Back in `print_path(2)` : print " -> 2".

Step 5 / 8

```
1 void print_path(int pred[], int s, int h)
2 {
3     if (h == s) {
4         printf("%d", s);
5         return;
6     }
7     print_path(pred, s, pred[h]);   /* pred first */
▶     printf(" -> %d", h);
9 }
```

VARIABLES

`s = 0` `h = 2`

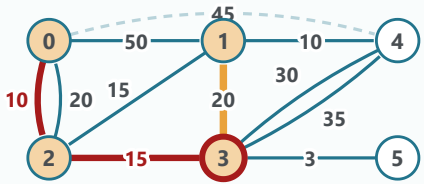
CALL STACK

`print_path(2)` `print_path(3)` `print_path(1)` `main()`

OUTPUT

```
0 -> 2
```

Code Trace: `print_path(pred, 0, 1)`



pred[]

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

1 ← 3 ← 2 ← 0 (traced backward)

Back in `print_path(3)` : print " -> 3".

Step 6 / 8

```
1 void print_path(int pred[], int s, int h)
2 {
3     if (h == s) {
4         printf("%d", s);
5         return;
6     }
7     print_path(pred, s, pred[h]);   /* pred first */
▶     printf(" -> %d", h);
9 }
```

VARIABLES

`s = 0` `h = 3`

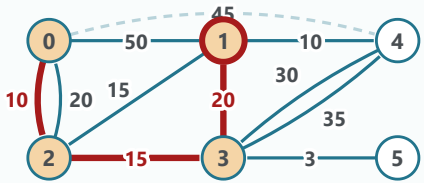
CALL STACK

`print_path(3)` `print_path(1)` `main()`

OUTPUT

```
0 -> 2 -> 3
```

Code Trace: `print_path(pred, 0, 1)`



pred[]

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

1 ← 3 ← 2 ← 0 (traced backward)

Back in `print_path(1)` : print " -> 1".

Step 7 / 8

```
1 void print_path(int pred[], int s, int h)
2 {
3     if (h == s) {
4         printf("%d", s);
5         return;
6     }
7     print_path(pred, s, pred[h]);   /* pred first */
▶     printf(" -> %d", h);
9 }
```

VARIABLES

`s = 0` `h = 1`

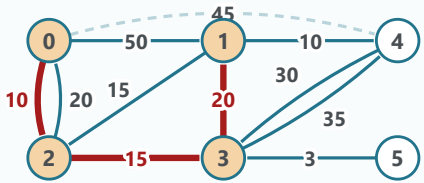
CALL STACK

`print_path(1)` `main()`

OUTPUT

```
0 -> 2 -> 3 -> 1
```

Code Trace: `print_path(pred, 0, 1)`



pred[]

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

1 ← 3 ← 2 ← 0 (traced backward)

Path: **0 -> 2 -> 3 -> 1**, cost 10 + 15 + 20 = 45. We walked back from the target; the call stack put the output in the right order.

Step 8 / 8

```
1 void print_path(int pred[], int s, int h)
2 {
3     if (h == s) {
4         printf("%d", s);
5         return;
6     }
7     print_path(pred, s, pred[h]);   /* pred first */
▶     printf(" -> %d", h);
9 }
```

VARIABLES

`s = 0` `h = 1`

CALL STACK

`main()`

OUTPUT

```
0 -> 2 -> 3 -> 1
```

Finding the Path: The Predecessor Array

Instructor: “We have the cost, but how do we find the path? Cost found, now the path! … Since we don't know which way to go, it can't be solved forward. One array is enough: the predecessor array… used backward.”

Why is one array enough?

Each vertex's shortest path is its predecessor's shortest path plus one edge (optimal substructure). The paths form a **tree**: the shortest-path tree.

Update

On a successful relaxation, `pred[w] = u`. When the cost changes, so does the predecessor.

Using it backward

pred[] walks from the target back to the source; recursion or a stack reverses it for printing.

Unreachable vertex

If `dist[h] ≥ INF`, print “no path”; pred[] is meaningless.

Software Engineering Concepts

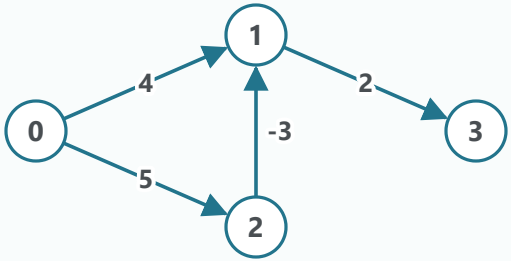
> **Instructor:** "Some of you tried to solve the midterm question with a single recursive function… Others called it as `FindWeight(…);` . What is this? 'What's wrong with it?' What does it return? 'It returns an int.' So where does it go? The return value is left hanging in the air. 'Oh, I forgot about that.'"

Solutions should be **modular, simple, readable, reusable and easy to maintain**. Keep a software engineering book at hand:

Reliability	**Usability**	**Maintainability**	**Portability**	**Integrity**
Flexibility	**Testability**	**Efficiency**	**Interoperability**	**Understandability**
Modularity	**Traceability**	**Modifiability**	**Correctness**	**Expandability**

That is also why the code in this chapter is split into small functions: `choose`, `print_path`, `find` …

Why Dijkstra Fails with Negative Edges (Extra Topic)



Vertex	Dijkstra	Correct
0	0	0
1	4	2
2	5	5
3	6	4

Dijkstra finalizes 1 at 4 (no smaller candidate). Then edge 2 → 1 (−3) reaches 1 at cost 2, but 1 is already marked. 3 stays wrong too.

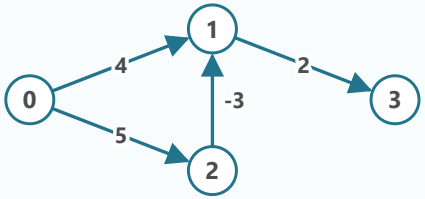
Assumption

Dijkstra assumes a chosen vertex can never be reached later by a **cheaper** path; that holds only when $w \geq 0$.

Fix

Bellman-Ford: relax every edge for $n - 1$ rounds. With a negative **cycle** shortest paths are undefined; a relaxation in round n detects it.

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	∞	∞	∞

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

d[] = ∞, source d[0] = 0.

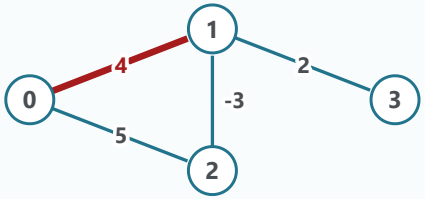
Step 1 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
▶  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
6          if (d[u] != INF && d[u] + E[j].w < d[v])
7              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;              /* -1: negative cycle */
```

VARIABLES

t = – j = –

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	4	∞	∞

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 1, 0→1 (4): d[0] + 4 = 4 is smaller → **d[1] = 4**.

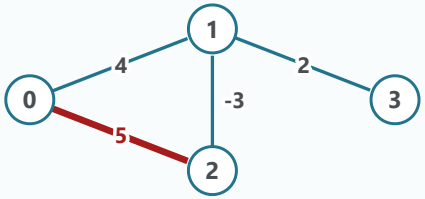
Step 2 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
           if (d[u] != INF && d[u] + E[j].w < d[v])
               d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;                 /* -1: negative cycle */
```

VARIABLES

t = 1 | j = 0 | u = 0 | v = 1

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	4	5	∞

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 1, 0→2 (5): d[0] + 5 = 5 is smaller → **d[2] = 5**.

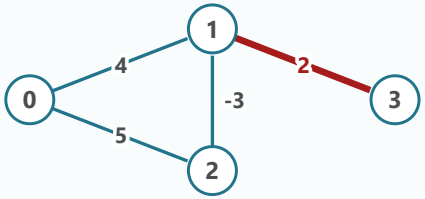
Step 3 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
▶          if (d[u] != INF && d[u] + E[j].w < d[v])
▶              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;                 /* -1: negative cycle */
```

VARIABLES

t = 1 | j = 1 | u = 0 | v = 2

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	4	5	6

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 1, 1→3 (2): d[1] + 2 = 6 is smaller → **d[3] = 6**.

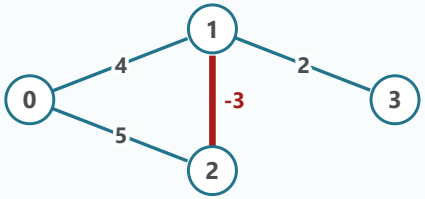
Step 4 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
▶          if (d[u] != INF && d[u] + E[j].w < d[v])
▶              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;                 /* -1: negative cycle */
```

VARIABLES

t = 1 | j = 2 | u = 1 | v = 3

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	2	5	6

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 1, 2→1 (-3): d[2] + (−3) = 2 is smaller → **d[1] = 2**.

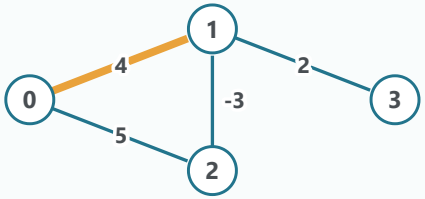
Step 5 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
▶          if (d[u] != INF && d[u] + E[j].w < d[v])
▶              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;                 /* -1: negative cycle */
```

VARIABLES

t = 1 | j = 3 | u = 2 | v = 1

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	2	5	6

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 2, 0→1 (4): 0 + 4 = 4 ≥ d[1] = 2, no change.

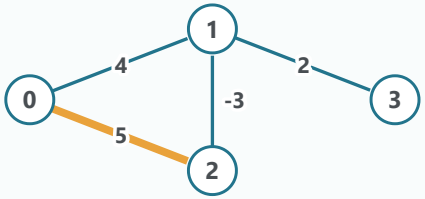
Step 6 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
▶          if (d[u] != INF && d[u] + E[j].w < d[v])
7              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;              /* -1: negative cycle */
```

VARIABLES

t = 2 | j = 0 | u = 0 | v = 1

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	2	5	6

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 2, 0→2 (5): 0 + 5 = 5 ≥ d[2] = 5, no change.

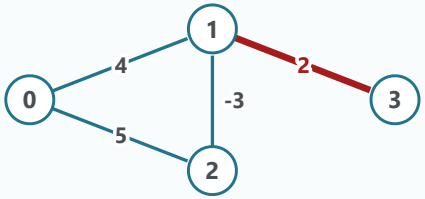
Step 7 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
▶          if (d[u] != INF && d[u] + E[j].w < d[v])
7              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;              /* -1: negative cycle */
```

VARIABLES

t = 2 | j = 1 | u = 0 | v = 2

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	2	5	4

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 2, 1→3 (2): d[1] + 2 = 4 is smaller → **d[3] = 4**.

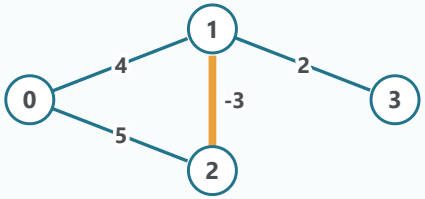
Step 8 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
▶          if (d[u] != INF && d[u] + E[j].w < d[v])
▶              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;                 /* -1: negative cycle */
```

VARIABLES

t = 2 | j = 2 | u = 1 | v = 3

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	2	5	4

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 2, 2→1 (-3): 5 + (−3) = 2 ≥ d[1] = 2, no change.

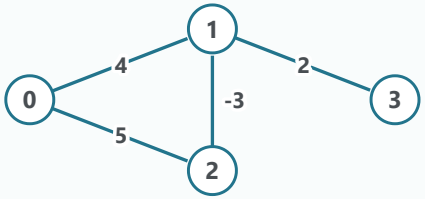
Step 9 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
▶          if (d[u] != INF && d[u] + E[j].w < d[v])
7              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)               /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;              /* -1: negative cycle */
```

VARIABLES

t = 2 | j = 3 | u = 2 | v = 1

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	2	5	4

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

Round 3: no edge relaxes (d is already final). In general n − 1 rounds are needed: a shortest path has at most n − 1 edges.

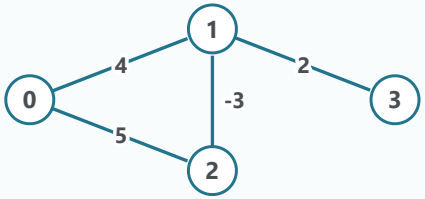
Step 10 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
▶ for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
6          if (d[u] != INF && d[u] + E[j].w < d[v])
7              d[v] = d[u] + E[j].w;
8      }
9  for (j = 0; j < e; j++)              /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
12 return 0;              /* -1: negative cycle */
```

VARIABLES

t = 3 j = –

Code Trace: Bellman-Ford (Source 0)



d[]

0	1	2	3
0	2	5	4

E[] (edge list, in this order)

0→1 (4)	0→2 (5)	1→3 (2)	2→1 (-3)

No relaxation in round n either → no negative cycle, return 0. Result d = [0, 2, 5, 4]. Cost **O(n·e)**.

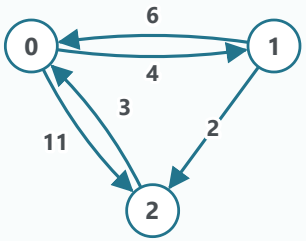
Step 11 / 11

```
1  for (i = 0; i < n; i++) d[i] = INF;
2  d[s] = 0;
3  for (t = 1; t <= n - 1; t++)          /* n-1 rounds */
4      for (j = 0; j < e; j++) {
5          u = E[j].u;  v = E[j].v;
6          if (d[u] != INF && d[u] + E[j].w < d[v])
7              d[v] = d[u] + E[j].w;
8      }
▶ for (j = 0; j < e; j++)                /* round n */
10     if (d[E[j].u] != INF &&
11         d[E[j].u] + E[j].w < d[E[j].v]) return -1;
▶ return 0;                 /* -1: negative cycle */
```

VARIABLES

t = – j = –

Code Trace: Floyd-Warshall



A (k = −1)

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

A^{-1} = the cost matrix (Horowitz example; there is no direct edge 2 → 1).

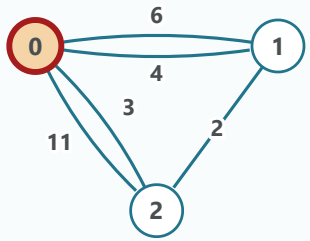
Step 1 / 8

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
          A[i][j] = cost[i][j];
4 for (k = 0; k < n; k++)              /* via vertex k
  */
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
7             if (A[i][k] + A[k][j] < A[i][j])
8                 A[i][j] = A[i][k] + A[k][j];
```

VARIABLES

k = – i = – j = –

Code Trace: Floyd-Warshall



A (k = 0)

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

k = 0: paths may now pass through vertex **0** (and lower-numbered ones).

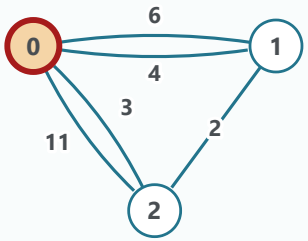
Step 2 / 8

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         A[i][j] = cost[i][j];
▶ for (k = 0; k < n; k++)              /* via vertex k
  */
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
7             if (A[i][k] + A[k][j] < A[i][j])
8                 A[i][j] = A[i][k] + A[k][j];
```

VARIABLES

k = 0 | i = – | j = –

Code Trace: Floyd-Warshall



A (k = 0)

	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

A[2][0] + A[0][1] = 3 + 4 = 7 < ∞ → **A[2][1] = 7**.

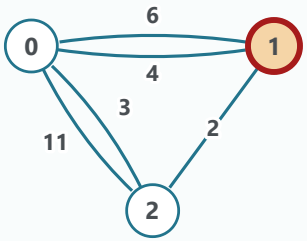
Step 3 / 8

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         A[i][j] = cost[i][j];
4 for (k = 0; k < n; k++)              /* via vertex k
  */
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
▶             if (A[i][k] + A[k][j] < A[i][j])
▶                 A[i][j] = A[i][k] + A[k][j];
```

VARIABLES

k = 0 | i = 2 | j = 1

Code Trace: Floyd-Warshall



A (k = 1)

	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

k = 1: paths may now pass through vertex **1** (and lower-numbered ones).

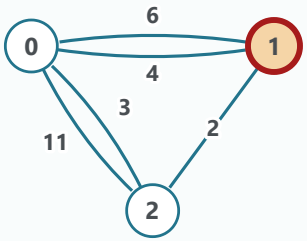
Step 4 / 8

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         A[i][j] = cost[i][j];
▶ for (k = 0; k < n; k++)              /* via vertex k
  */
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
7             if (A[i][k] + A[k][j] < A[i][j])
8                 A[i][j] = A[i][k] + A[k][j];
```

VARIABLES

k = 1 i = – j = –

Code Trace: Floyd-Warshall



A (k = 1)

	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

A[0][1] + A[1][2] = 4 + 2 = 6 < 11 → **A[0][2] = 6**.

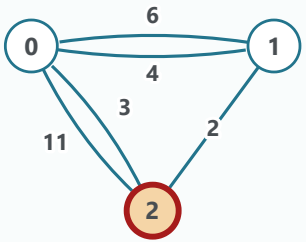
Step 5 / 8

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         A[i][j] = cost[i][j];
4 for (k = 0; k < n; k++)              /* via vertex k
  */
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
▶             if (A[i][k] + A[k][j] < A[i][j])
▶                 A[i][j] = A[i][k] + A[k][j];
```

VARIABLES

k = 1 | i = 0 | j = 2

Code Trace: Floyd-Warshall



A (k = 2)

	0	1	2
0	0	4	6
1	6	0	2
2	3	7	0

k = 2: paths may now pass through vertex **2** (and lower-numbered ones).

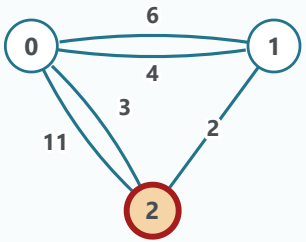
Step 6 / 8

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         A[i][j] = cost[i][j];
▶ for (k = 0; k < n; k++)              /* via vertex k
  */
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
7             if (A[i][k] + A[k][j] < A[i][j])
8                 A[i][j] = A[i][k] + A[k][j];
```

VARIABLES

k = 2 i = – j = –

Code Trace: Floyd-Warshall



A (k = 2)

	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

A[1][2] + A[2][0] = 2 + 3 = 5 < 6 → **A[1][0] = 5**.

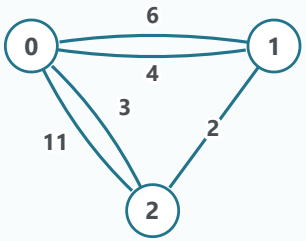
Step 7 / 8

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         A[i][j] = cost[i][j];
4 for (k = 0; k < n; k++)                /* via vertex k
  */
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
▶             if (A[i][k] + A[k][j] < A[i][j])
▶                 A[i][j] = A[i][k] + A[k][j];
```

VARIABLES

k = 2 | i = 1 | j = 0

Code Trace: Floyd-Warshall



A (k = 2), result

	0	1	2
0	0	4	6
1	5	0	2
2	3	7	0

Done: A = shortest distances between all pairs. Three nested loops → **O(n³)**; negative edges are allowed, negative cycles are not.

Step 8 / 8

```
1 for (i = 0; i < n; i++)
2     for (j = 0; j < n; j++)
3         A[i][j] = cost[i][j];
▶ for (k = 0; k < n; k++)              /* via vertex k
  */
5     for (i = 0; i < n; i++)
6         for (j = 0; j < n; j++)
7             if (A[i][k] + A[k][j] < A[i][j])
8                 A[i][j] = A[i][k] + A[k][j];
```

VARIABLES

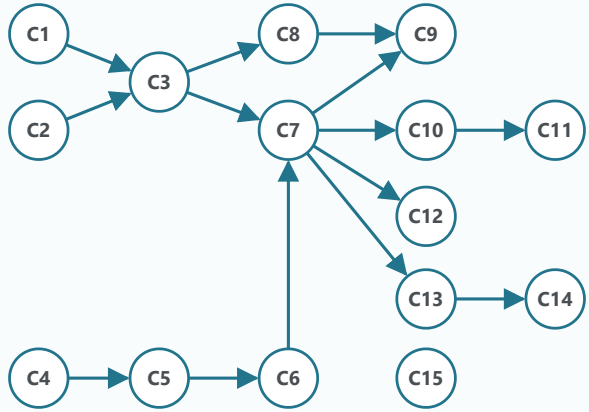
k = 3 | i = – | j = –

Shortest Paths: Which Algorithm, When?

Problem	Algorithm	Time	Condition
Unweighted, single source	BFS	$O(n + e)$	all edges equal
Single source, $w \geq 0$	Dijkstra (matrix)	$O(n^2)$	dense graph
Single source, $w \geq 0$	Dijkstra (heap + lists)	$O((n + e) \log n)$	sparse graph
Single source, negative edges	Bellman-Ford	$O(n \cdot e)$	detects negative cycles
DAG, single source	Relax in topological order	$O(n + e)$	no cycles (activity networks!)
All pairs	Floyd-Warshall	$O(n^3)$	no negative cycle
All pairs, sparse	$n \times$ Dijkstra	$O(n (n + e) \log n)$	$w \geq 0$

Activity Networks: A Project = Activities

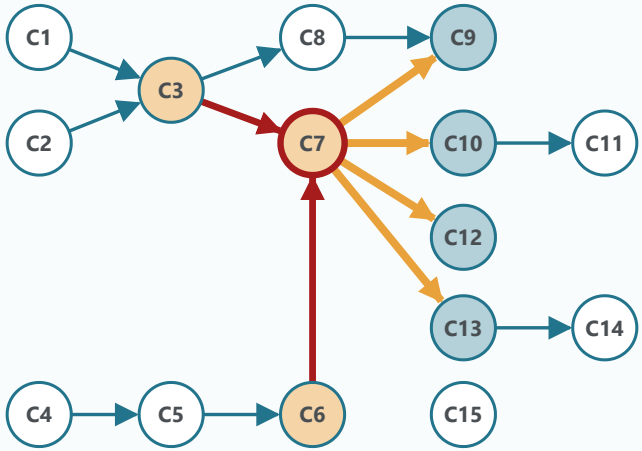
Code	Course	Prerequisites
C1	Programming I	–
C2	Discrete Math.	–
C3	Data Structures	C1, C2
C4	Calculus I	–
C5	Calculus II	C4
C6	Linear Algebra	C5
C7	Analysis of Algorithms	C3, C6
C8	Assembly Lang.	C3
C9	Operating Systems	C7, C8
C10	Programming Langs.	C7
C11	Compiler Design	C10
C12	Artificial Intelligence	C7
C13	Computational Theory	C7
C14	Parallel Algorithms	C13
C15	Numerical Analysis	–



A project ends when its subtasks (**activities**) end: graduation = courses. In construction: lay the foundation, the 2nd floor 21 days later… **When** does the project end? Which tasks are **critical**? Which have **slack**?

Activity-on-Vertex (AoV) Networks and Precedence

- **AoV** (*activity on vertex*): activities are **vertices**, the *precedence relation* gives the **edges**: <i, j> if course i is a prerequisite of j.
- C3 and C6 are **immediate predecessors** of C7.
- C9, C10, C12 and C13 are **immediate successors** of C7.
- C14 is a successor of C3, but not an **immediate** one.
- Precedence is transitive and must be acyclic: A cannot precede B while B precedes A.



Instructor: "A side note: dropping courses like operations research and numerical analysis leaves students weak in mathematics."

Definition: Topological Order

A **linear order** of the vertices in which i comes **before** j whenever i is a predecessor of j. There can be **several** topological orders; if the graph has a cycle there is none.

It goes back to the planning methods (PERT, 1958) developed during the Cold War to speed up missile (Polaris) production. Two valid topological orders of the courses (checked by program):

```
C1, C2, C4, C5, C3, C6, C8, C7, C10, C13, C12, C14, C15, C11, C9
C4, C5, C2, C1, C6, C3, C8, C15, C7, C9, C10, C11, C12, C13, C14
```

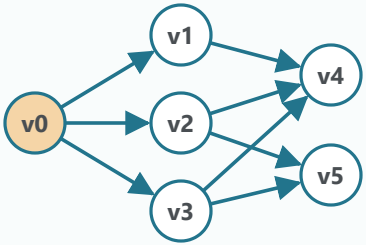
Check

For every edge <i, j>, does i come before j in the order? All 14 edges pass.

Uses

Course plans, build order (make), task scheduling, spreadsheet cells, earliest times.

Finding a Topological Order: Delete a Vertex with No Predecessor



order: —

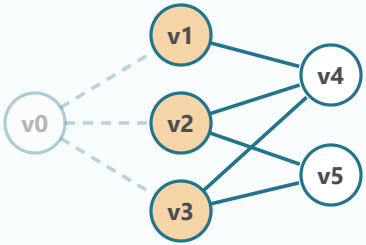
no predecessors: v0

a) v0 has no predecessor → the first vertex to choose.

Step 1 / 7

Each time, take a vertex with no predecessors, output it and delete its out-edges; repeat to the end. If vertices remain but none is free of predecessors: a **cycle**.

Finding a Topological Order: Delete a Vertex with No Predecessor



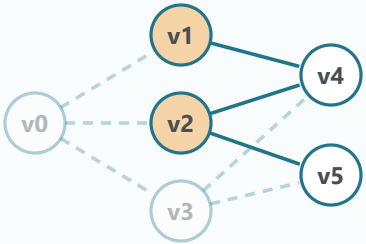
order: v0

no predecessors: v1, v2, v3

b) v0 is output and its out-edges deleted. **Any** vertex with no predecessors left (orange) may be chosen. If v3 is chosen, <v3,v4> and <v3,v5> are deleted.

Each time, take a vertex with no predecessors, output it and delete its out-edges; repeat to the end. If vertices remain but none is free of predecessors: a **cycle**.

Finding a Topological Order: Delete a Vertex with No Predecessor



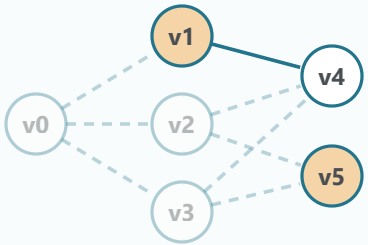
order: v0, v3

no predecessors: v1, v2

c) v3 is output and its out-edges deleted. **Any** vertex with no predecessors left (orange) may be chosen.

Each time, take a vertex with no predecessors, output it and delete its out-edges; repeat to the end. If vertices remain but none is free of predecessors: a **cycle**.

Finding a Topological Order: Delete a Vertex with No Predecessor



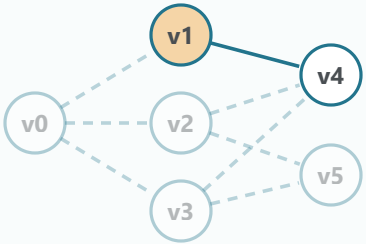
order: v0, v3, v2

no predecessors: v1, v5

d) v2 is output and its out-edges deleted. **Any** vertex with no predecessors left (orange) may be chosen.

Each time, take a vertex with no predecessors, output it and delete its out-edges; repeat to the end. If vertices remain but none is free of predecessors: a **cycle**.

Finding a Topological Order: Delete a Vertex with No Predecessor



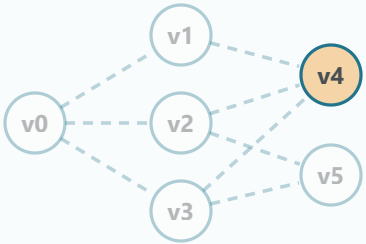
order: v0, v3, v2, v5

no predecessors: v1

e) v5 is output and its out-edges deleted. **Any** vertex with no predecessors left (orange) may be chosen.

Each time, take a vertex with no predecessors, output it and delete its out-edges; repeat to the end. If vertices remain but none is free of predecessors: a **cycle**.

Finding a Topological Order: Delete a Vertex with No Predecessor



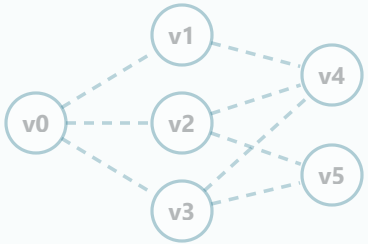
order: v0, v3, v2, v5, v1

no predecessors: v4

f) v1 is output and its out-edges deleted. **Any** vertex with no predecessors left (orange) may be chosen.

Each time, take a vertex with no predecessors, output it and delete its out-edges; repeat to the end. If vertices remain but none is free of predecessors: a **cycle**.

Finding a Topological Order: Delete a Vertex with No Predecessor



order: v0, v3, v2, v5, v1, v4

no predecessors: —

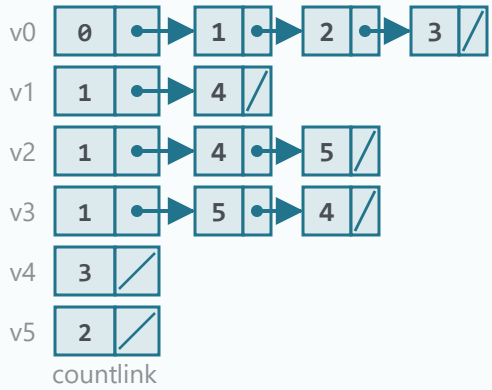
g) One possible topological order: **v0, v3, v2, v5, v1, v4**.

Each time, take a vertex with no predecessors, output it and delete its out-edges; repeat to the end. If vertices remain but none is free of predecessors: a **cycle**.

Data Structure: A Predecessor Count in the Header

```
typedef struct node *node_pointer;
typedef struct node {
    int vertex;
    node_pointer link;
} node;
typedef struct {
    int count;      /* in-degree: edges <i,j> into j */
    node_pointer link;
} hdnodes;
hdnodes graph[MAX_VERTICES];
```

- Vertices are numbered 0..n−1.
- `count` : the vertex's number of predecessors (in-edges); at 0 it can be output.
- While reading the edges: `graph[j].count++` .



Lists with counts for the v0..v5 example

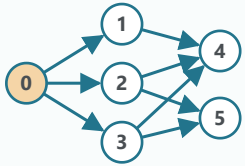
The Topological Sort Function

```
void topsort(hdnodes graph[], int n)
{
    int i, j, k, top = -1;
    node_pointer p;
    for (i = 0; i < n; i++)      /* count 0: push */
        if (!graph[i].count) {
            graph[i].count = top;  top = i; }
    for (i = 0; i < n; i++) {
        if (top == -1) { printf("cycle"); break; }
        j = top;  top = graph[top].count;
        printf("%d ", j);
        for (p = graph[j].link; p; p = p->link) {
            k = p->vertex;
            if (--graph[k].count == 0) {
                graph[k].count = top;  top = k; }
        }
    }
}
```

> **Instructor:** “There is no stack anywhere, yet we wrote comments saying ‘push’ and ‘pop’. That is exactly why comments matter. It does use a stack, but how?”

- Vertices with no predecessors left are **chained together** through their `count` fields: count now means “the vertex below me on the stack”. This resembles the **parent array** of disjoint sets.
- The pair `count--; if (!count)` is written here as `--count == 0` (same meaning).

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	1	1	1	3	2

top = 0

stack (top at left): v0

v0's count is 0 → push: `count = top` (−1: the bottom of the stack), `top = 0`.

Step 1 / 11

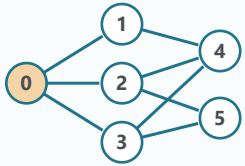
```
1  for (i = 0; i < n; i++)          /* count 0: push */
▶      if (!graph[i].count) {
▶          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

VARIABLES

`i = 0` `top = 0`

OUTPUT

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	1	1	1	3	2

top = 0

stack (top at left): v0

The first loop is done: only v0 has no predecessors. The other counts hold predecessor counts.

Step 2 / 11

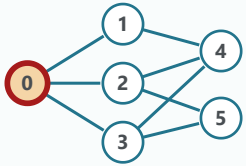
```
1  for (i = 0; i < n; i++)          /* count 0: push */
▶      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

VARIABLES

i = 6 top = 0

OUTPUT

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	1	1	1	3	2

`top = -1`

stack (top at left): empty

Pop: **j = 0**, `top = count[0] = −1`. Output v0.

Step 3 / 11

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
▶      j = top;  top = graph[top].count;
▶      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

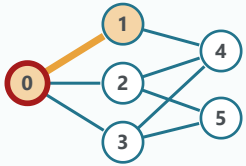
VARIABLES

`i = 0` `j = 0` `top = -1`

OUTPUT

```
0
```

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	1	3	2

top = 1

stack (top at left): v1

<0,1>: v1's count drops to 0 → push: `count[1] = -1` , `top = 1` .

Step 4 / 11

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
▶          if (--graph[k].count == 0) {
▶              graph[k].count = top;  top = k; }
12     }
13 }
```

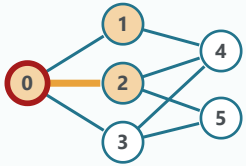
VARIABLES

i = 0 j = 0 k = 1 top = 1

OUTPUT

```
0
```

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	1	3	2

top = 2

stack (top at left): v2, v1

<0,2>: v2's count drops to 0 → push: `count[2] = 1`, `top = 2`.

Step 5 / 11

```
 1 for (i = 0; i < n; i++)          /* count 0: push */
 2     if (!graph[i].count) {
 3         graph[i].count = top;  top = i; }
 4 for (i = 0; i < n; i++) {
 5     if (top == -1) { printf("cycle"); break; }
 6     j = top;  top = graph[top].count;
 7     printf("%d ", j);
 8     for (p = graph[j].link; p; p = p->link) {
 9         k = p->vertex;
▶           if (--graph[k].count == 0) {
▶               graph[k].count = top;  top = k; }
12     }
13 }
```

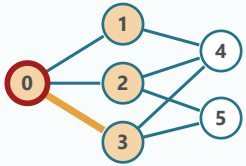
VARIABLES

`i = 0` `j = 0` `k = 2` `top = 2`

OUTPUT

```
0
```

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	3	2

top = 3

stack (top at left): v3, v2, v1

<0,3>: v3's count drops to 0 → push: `count[3] = 2`, `top = 3`.

Step 6 / 11

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
▶          if (--graph[k].count == 0) {
▶              graph[k].count = top;  top = k; }
12     }
13 }
```

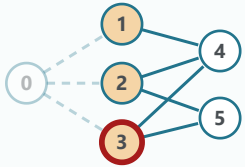
VARIABLES

`i = 0` `j = 0` `k = 3` `top = 3`

OUTPUT

```
0
```

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	3	2

top = 2

stack (top at left): v2, v1

Pop: **j = 3**, `top = count[3] = 2`. Output v3.

Step 7 / 11

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
▶      j = top;  top = graph[top].count;
▶      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

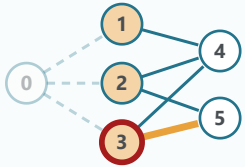
VARIABLES

`i = 1` `j = 3` `top = 2`

OUTPUT

```
0 3
```

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	3	1

top = 2

stack (top at left): v2, v1

<3,5>: v5's predecessor count drops by 1 → 1; not ready yet.

Step 8 / 11

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
▶          k = p->vertex;
▶          if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

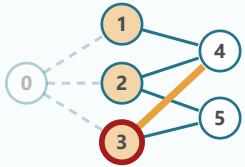
VARIABLES

i = 1 | j = 3 | k = 5 | top = 2

OUTPUT

```
0 3
```

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	2	1

top = 2

stack (top at left): v2, v1

<3,4>: v4's predecessor count drops by 1 → 2; not ready yet.

Step 9 / 11

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
▶          k = p->vertex;
▶          if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

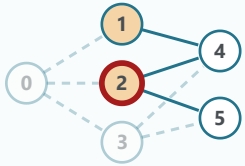
VARIABLES

i = 1 | j = 3 | k = 4 | top = 2

OUTPUT

```
0 3
```

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	2	1

top = 1

stack (top at left): v1

Pop: **j = 2**, `top = count[2] = 1`. Output v2.

Step 10 / 11

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
▶      j = top;  top = graph[top].count;
▶      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

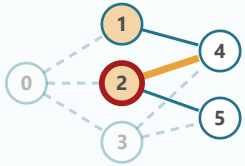
VARIABLES

`i = 2` `j = 2` `top = 1`

OUTPUT

```
0 3 2
```

Code Trace: `topsort(graph, 6)`



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	1	1

top = 1

stack (top at left): v1

<2,4>: v4's predecessor count drops by 1 → 1; not ready yet.

Step 11 / 11

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
▶          k = p->vertex;
▶          if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

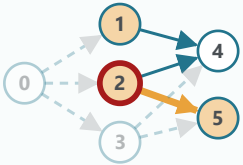
VARIABLES

i = 2 | j = 2 | k = 4 | top = 1

OUTPUT

```
0 3 2
```

Code Trace: `topsort()` (Continued)



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	1	1

top = 5

stack (top at left): v5, v1

<2,5>: v5's count drops to 0 → push: `count[5] = 1`, `top = 5`.

Step 1 / 6

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
▶          if (--graph[k].count == 0) {
▶              graph[k].count = top;  top = k; }
12     }
13 }
```

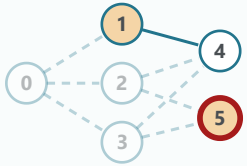
VARIABLES

`i = 2` `j = 2` `k = 5` `top = 5`

OUTPUT

```
0 3 2
```

Code Trace: `topsort()` (Continued)



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	1	1

top = 1

stack (top at left): v1

Pop: **j = 5**, `top = count[5] = 1`. Output v5.

Step 2 / 6

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
▶      j = top;  top = graph[top].count;
▶      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

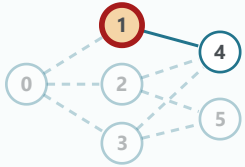
VARIABLES

`i = 3` `j = 5` `top = 1`

OUTPUT

```
0 3 2 5
```

Code Trace: `topsort()` (Continued)



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	1	1

`top = -1` stack (top at left): empty

Pop: **j = 1**, `top = count[1] = −1`. Output v1.

Step 3 / 6

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
▶      j = top;  top = graph[top].count;
▶      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

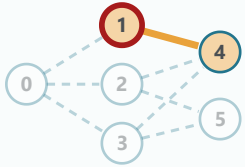
VARIABLES

`i = 4` `j = 1` `top = −1`

OUTPUT

```
0 3 2 5 1
```

Code Trace: `topsort()` (Continued)



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	-1	1

top = 4

stack (top at left): v4

<1,4>: v4's count drops to 0 → push: `count[4] = -1`, `top = 4`.

Step 4 / 6

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
▶          if (--graph[k].count == 0) {
▶              graph[k].count = top;  top = k; }
12     }
13 }
```

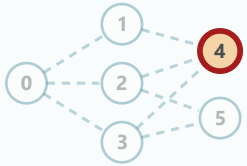
VARIABLES

`i = 4` `j = 1` `k = 4` `top = 4`

OUTPUT

```
0 3 2 5 1
```

Code Trace: `topsort()` (Continued)



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	-1	1

`top = -1`

stack (top at left): empty

Pop: **j = 4**, `top = count[4] = −1`. Output v4.

Step 5 / 6

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
5      if (top == -1) { printf("cycle"); break; }
▶      j = top;  top = graph[top].count;
▶      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

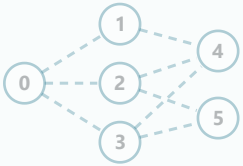
VARIABLES

`i = 5` `j = 4` `top = −1`

OUTPUT

```
0 3 2 5 1 4
```

Code Trace: `topsort()` (Continued)



graph[].count

v0	v1	v2	v3	v4	v5
-1	-1	1	2	-1	1

`top = -1` stack (top at left): empty

Every vertex is output: **0 3 2 5 1 4**. With a cycle, some round would see `top == -1` and print “cycle”.

Step 6 / 6

```
1  for (i = 0; i < n; i++)          /* count 0: push */
2      if (!graph[i].count) {
3          graph[i].count = top;  top = i; }
4  for (i = 0; i < n; i++) {
▶      if (top == -1) { printf("cycle"); break; }
6      j = top;  top = graph[top].count;
7      printf("%d ", j);
8      for (p = graph[j].link; p; p = p->link) {
9          k = p->vertex;
10         if (--graph[k].count == 0) {
11             graph[k].count = top;  top = k; }
12     }
13 }
```

VARIABLES

`i = 6` `top = -1`

OUTPUT

```
0 3 2 5 1 4
```

Cycle Detection and Analysis

It finds cycles too

A cycle means A and B each precede the other, which is an error. “With a cycle the project never ends.” If the stack is empty in some round while vertices remain, there is a cycle.

Analysis

The first for is O(n), the second O(n); the inner for runs d_j times (out-degree). Total $O(\Sigma d_i + n)$ = **$O(n + e)$**.

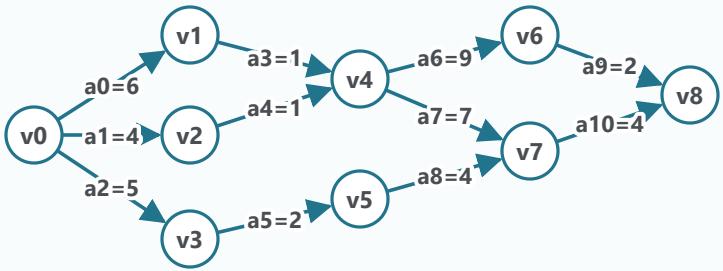
What decides the order?

The push order follows the order of the adjacency lists; with a queue (Kahn) you would get a different but still valid order.

Graph structure

Depending on how the graph is connected, e ranges from O(n) to $O(n^2)$.

Activity-on-Edge (AoE) Networks



v0 start, v8 finish: 11 activities (edges), 9 events (vertices)

AoE (activity on edge/arc)

Unlike AoV, **directed edges are activities** (with durations) and **vertices are events**.

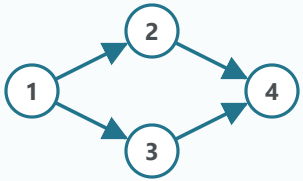
Event

A milestone the project must reach: “the work that must be finished is expressed by a vertex”.

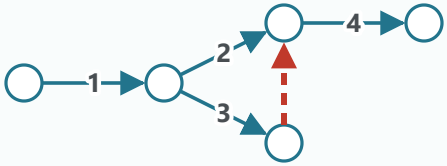
Parallelism

a0, a1 and a2 start in parallel; a6 and a7 start once a3 and a4 finish; a10 needs a7 and a8 finished.

Graph Drawing: From AoV to AoE, Dummy Activities



AoV: 4 waits for both 2 and 3



dummy arc, duration 0

In AoV, activity 4 waits for 2 and 3; in AoE this needs a dummy edge of cost 0.

Dummy activity

An edge of cost 0. E.g. if a6 and a7 above had to wait for a3, a4 **and a5**, the dummy activity <v5, v4> would be needed.

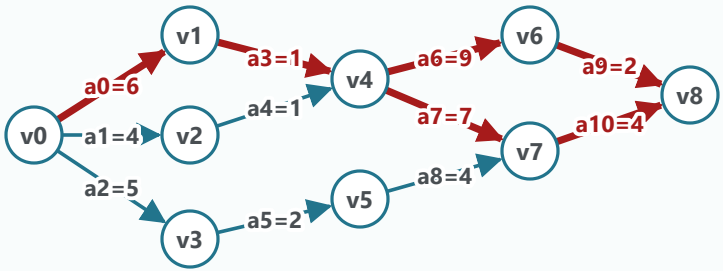
Minimum arc problem

Drawing the graph with the **fewest** dummy activities is NP-hard, a research topic with thesis-level work on it.

Methods

PERT (*Program Evaluation and Review Technique*, 1958), **CPM** (*Critical Path Method*, late 1950s), RAMPS (*Resource Allocation and Multi-Project Scheduling*).

The Critical Path



Critical paths: v0-v1-v4-v6-v8 and v0-v1-v4-v7-v8 (length 18)

- Summing activity durations, the **longest** path from v0 to v8 is the **critical path**. There may be more than one.
- Here 6 + 1 + 9 + 2 = 18 and 6 + 1 + 7 + 4 = 18 → the project ends **at the earliest in 18** days.
- If its prerequisites finish as fast as possible, **when does** an activity start? **How late** can it start without delaying the project?
- To the crews on critical tasks: “Finish in 4 days, don't let it slip to 5. If it's done in 3, I'll be happy.” An operations research problem.

Instructor: “I asked for earliest and latest start times on the 2nd midterm, and I'll ask again. I give this and say ‘Run it.’ Everyone thinks they understood it, but 80% cannot run it.”

Earliest Start Times

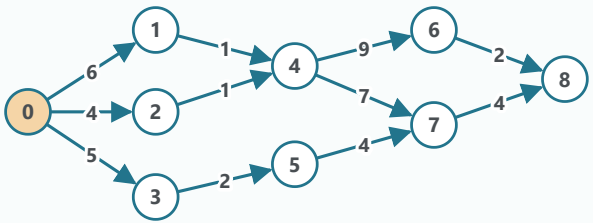
- If a_i = <k, m>, then `early(i) = earliest[k]`: the activity starts when the event at its tail occurs.
- E.g. a1: k = v0 → 0; a5: k = v3 → 5 (a2); a8: k = v5 → 5 + 2 = 7.
- Two paths lead to v4: a3 ends at the earliest at 6 + 1 = 7, a4 at 4 + 1 = 5. Both must finish → wait for the **longer** (7).
- `earliest[m] = max { earliest[k] + dur(<k,m>) }`, k ∈ P(m) (immediate predecessors).
- The array starts at **0**: day 0 of the project (e.g. "May 8"); with 10, everything would shift by 10.
- Compute in **topological order**: the predecessors' values must be known first.

```
/* in topsort, before decrementing count: */
if (earliest[k] < earliest[j] + p->duration)
    earliest[k] = earliest[j] + p->duration;
```



A `duration` field is added to the list node; the value is updated when a **longer path** is found.

Code Trace: earliest[] (in Topological Order)



earliest[]

0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0

stack (top at left): 0

Start: earliest[] = 0; the first loop pushed v0, which has no predecessors.

Step 1 / 11

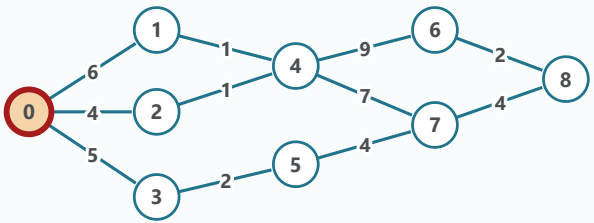
```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

VARIABLES

j = – top = 0

OUTPUT

Code Trace: earliest[] (in Topological Order)



earliest[]

0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0

stack (top at left): empty

v0 is popped and output (earliest[0] = 0 is now final).

Step 2 / 11

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

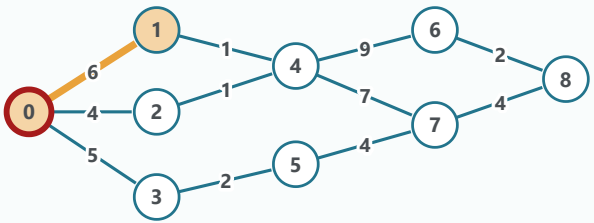
VARIABLES

`j = 0` `top = -1`

OUTPUT

```
0
```

Code Trace: earliest[] (in Topological Order)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	0	0	0	0	0	0	0

stack (top at left): 1

<0,1> (duration 6): 0 + 6 = 6 > 0 → **earliest[1] = 6**; v1 is pushed.

Step 3 / 11

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
▶          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

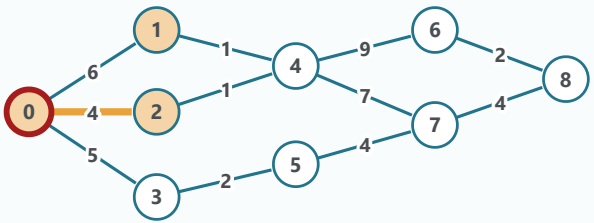
VARIABLES

j = 0 k = 1 top = 1

OUTPUT

```
0
```

Code Trace: earliest[] (in Topological Order)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	0	0	0	0	0	0

stack (top at left): 2, 1

<0,2> (duration 4): 0 + 4 = 4 > 0 → **earliest[2] = 4**; v2 is pushed.

Step 4 / 11

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
▶          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

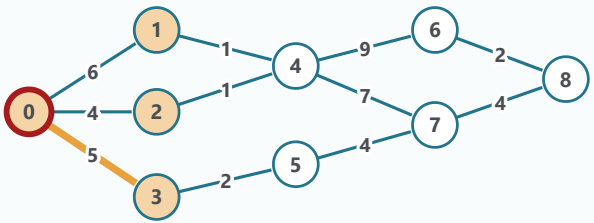
VARIABLES

j = 0 k = 2 top = 2

OUTPUT

```
0
```

Code Trace: earliest[] (in Topological Order)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	0	0	0	0	0

stack (top at left): 3, 2, 1

<0,3> (duration 5): 0 + 5 = 5 > 0 → **earliest[3] = 5**; v3 is pushed.

Step 5 / 11

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
▶          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

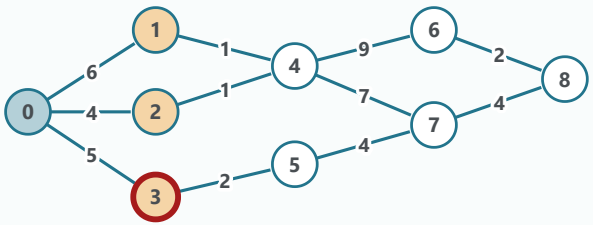
VARIABLES

j = 0 k = 3 top = 3

OUTPUT

```
0
```

Code Trace: earliest[] (in Topological Order)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	0	0	0	0	0

stack (top at left): 2, 1

v3 is popped and output (earliest[3] = 5 is now final).

Step 6 / 11

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

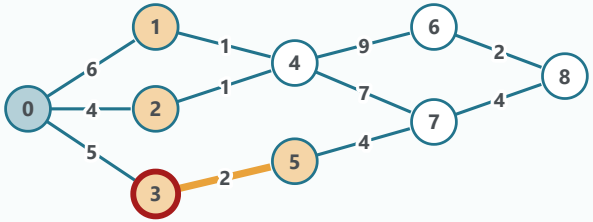
VARIABLES

`j = 3` `top = 2`

OUTPUT

```
0 3
```

Code Trace: earliest[] (in Topological Order)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	0	7	0	0	0

stack (top at left): 5, 2, 1

<3,5> (duration 2): 5 + 2 = 7 > 0 → **earliest[5] = 7**; v5 is pushed.

Step 7 / 11

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
▶          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

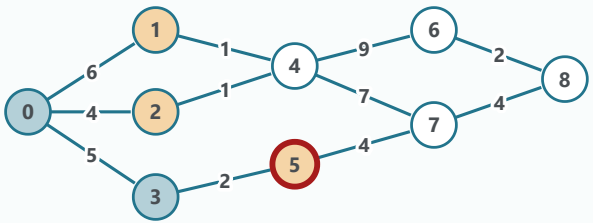
VARIABLES

j = 3 | k = 5 | top = 5

OUTPUT

```
0 3
```

Code Trace: earliest[] (in Topological Order)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	0	7	0	0	0

stack (top at left): 2, 1

v5 is popped and output (earliest[5] = 7 is now final).

Step 8 / 11

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

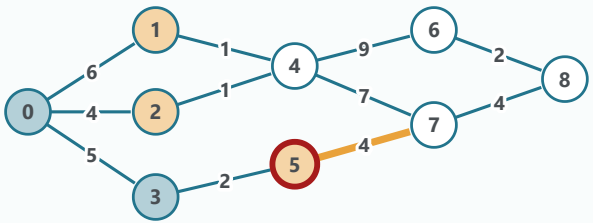
VARIABLES

`j = 5` `top = 2`

OUTPUT

```
0 3 5
```

Code Trace: earliest[] (in Topological Order)



earliest[]

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

stack (top at left): 2, 1

<5,7> (duration 4): 7 + 4 = 11 > 0 → **earliest[7] = 11**.

Step 9 / 11

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
8          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

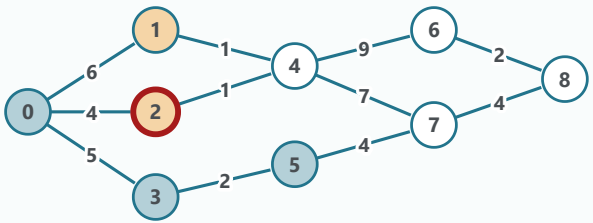
VARIABLES

j = 5 | k = 7 | top = 2

OUTPUT

```
0 3 5
```

Code Trace: earliest[] (in Topological Order)



earliest[]

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

stack (top at left): 1

v2 is popped and output (earliest[2] = 4 is now final).

Step 10 / 11

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

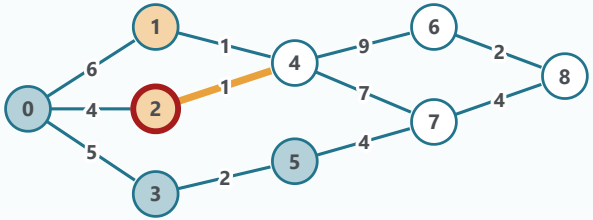
VARIABLES

`j = 2` `top = 1`

OUTPUT

```
0 3 5 2
```

Code Trace: earliest[] (in Topological Order)



earliest[]

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

stack (top at left): 1

<2,4> (duration 1): 4 + 1 = 5 > 0 → **earliest[4] = 5**.

Step 11 / 11

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
8          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

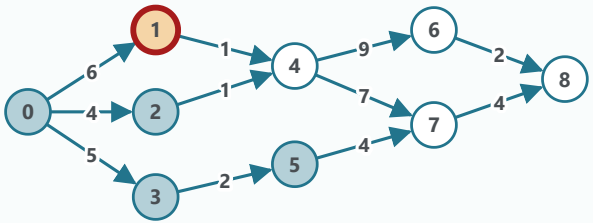
VARIABLES

j = 2 | k = 4 | top = 1

OUTPUT

```
0 3 5 2
```

Code Trace: earliest[] (Continued)



earliest[]

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

stack (top at left): empty

v1 is popped and output (earliest[1] = 6 is now final).

Step 1 / 10

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

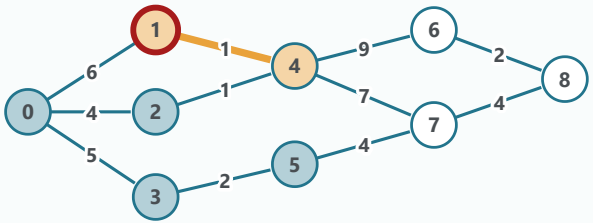
VARIABLES

`j = 1` `top = -1`

OUTPUT

```
0 3 5 2 1
```

Code Trace: earliest[] (Continued)



earliest[]

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

stack (top at left): 4

<1,4> (duration 1): 6 + 1 = 7 > 5 → **earliest[4] = 7**; v4 is pushed.

Step 2 / 10

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
▶          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

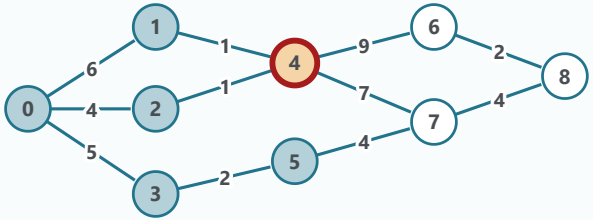
VARIABLES

j = 1 | k = 4 | top = 4

OUTPUT

```
0 3 5 2 1
```

Code Trace: earliest[] (Continued)



earliest[]

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

stack (top at left): empty

v4 is popped and output (earliest[4] = 7 is now final).

Step 3 / 10

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

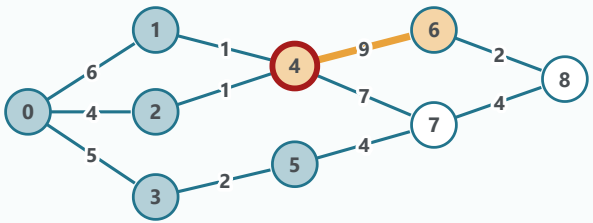
VARIABLES

`j = 4` `top = -1`

OUTPUT

```
0 3 5 2 1 4
```

Code Trace: earliest[] (Continued)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	7	7	16	11	0

stack (top at left): 6

<4,6> (duration 9): 7 + 9 = 16 > 0 → **earliest[6] = 16**; v6 is pushed.

Step 4 / 10

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
▶          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

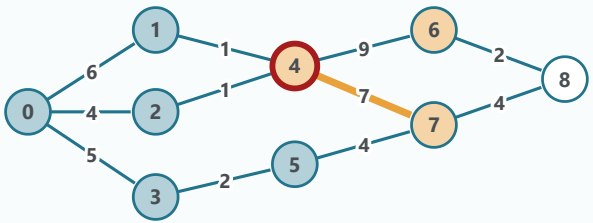
VARIABLES

j = 4 k = 6 top = 6

OUTPUT

```
0 3 5 2 1 4
```

Code Trace: earliest[] (Continued)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	7	7	16	14	0

stack (top at left): 7, 6

<4,7> (duration 7): 7 + 7 = 14 > 11 → **earliest[7] = 14**; v7 is pushed.

Step 5 / 10

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
▶          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

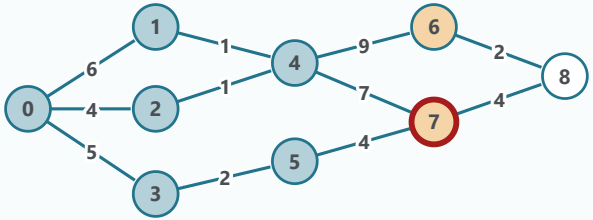
VARIABLES

j = 4 k = 7 top = 7

OUTPUT

```
0 3 5 2 1 4
```

Code Trace: earliest[] (Continued)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	7	7	16	14	0

stack (top at left): 6

v7 is popped and output (earliest[7] = 14 is now final).

Step 6 / 10

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

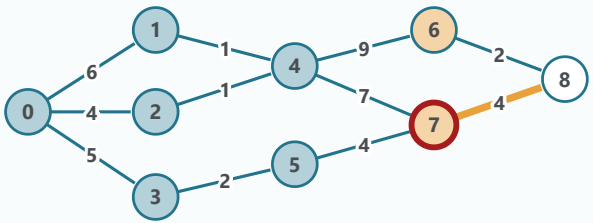
VARIABLES

`j = 7` `top = 6`

OUTPUT

```
0 3 5 2 1 4 7
```

Code Trace: earliest[] (Continued)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	7	7	16	14	18

stack (top at left): 6

<7,8> (duration 4): 14 + 4 = 18 > 0 → **earliest[8] = 18**.

Step 7 / 10

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
▶          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
8          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

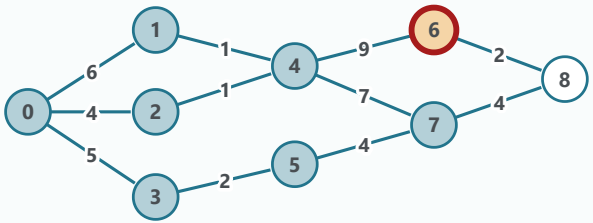
VARIABLES

j = 7 | k = 8 | top = 6

OUTPUT

```
0 3 5 2 1 4 7
```

Code Trace: earliest[] (Continued)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	7	7	16	14	18

stack (top at left): empty

v6 is popped and output (earliest[6] = 16 is now final).

Step 8 / 10

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

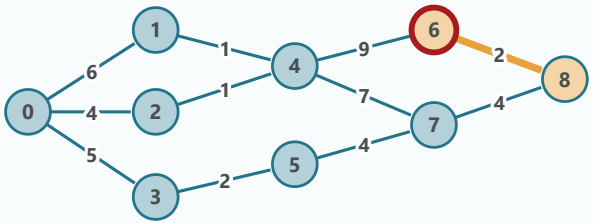
VARIABLES

`j = 6` `top = -1`

OUTPUT

```
0 3 5 2 1 4 7 6
```

Code Trace: earliest[] (Continued)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	7	7	16	14	18

stack (top at left): 8

<6,8> (duration 2): 16 + 2 = 18 ≤ 18, no change; v8 is pushed.

Step 9 / 10

```
1  j = top;  top = graph[top].count;   /* pop */
2  printf("%d ", j);
3  for (p = graph[j].link; p; p = p->link) {
▶      k = p->vertex;
▶      if (earliest[k] < earliest[j] + p->duration)
6          earliest[k] = earliest[j] + p->duration;
7      if (--graph[k].count == 0) {
▶          graph[k].count = top;  top = k;  /* push */
9      }
10 }
```

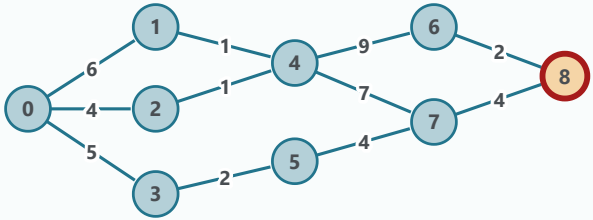
VARIABLES

`j = 6` `k = 8` `top = 8`

OUTPUT

```
0 3 5 2 1 4 7 6
```

Code Trace: earliest[] (Continued)



earliest[]

0	1	2	3	4	5	6	7	8
0	6	4	5	7	7	16	14	18

stack (top at left): empty

v8 is popped and output (earliest[8] = 18 is now final).

Step 10 / 10

```
▶ j = top;  top = graph[top].count;   /* pop */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;
      if (earliest[k] < earliest[j] + p->duration)
          earliest[k] = earliest[j] + p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;  /* push */
      }
  }
```

VARIABLES

`j = 8` `top = -1`

OUTPUT

```
0 3 5 2 1 4 7 6 8
```

Worksheet for the Earliest Start Times

earliest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
Initial	0	0	0	0	0	0	0	0	0	0
Output v0	0	6	4	5	0	0	0	0	0	3, 2, 1
Output v3	0	6	4	5	0	7	0	0	0	5, 2, 1
Output v5	0	6	4	5	0	7	0	11	0	2, 1
Output v2	0	6	4	5	5	7	0	11	0	1
Output v1	0	6	4	5	7	7	0	11	0	4
Output v4	0	6	4	5	7	7	16	14	0	7, 6
Output v7	0	6	4	5	7	7	16	14	18	6
Output v6	0	6	4	5	7	7	16	14	18	8
Output v8	0	6	4	5	7	7	16	14	18	–

Which values are final? “Only the earliest start times of vertices already output are final; from then on they never change.” E.g. when v5 was processed, [7] became 11; the longer path from v4 later made it 14. The project ends at the earliest in **18** days.

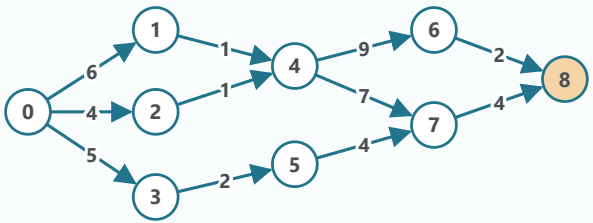
Latest Start Times

- A **backward** pass: reverse every arrow and work in the topological order of that graph (v8 first, v0 last). Structure: the **inverse adjacency lists**, built in O(n + e).
- Last vertex: `latest[n-1] = earliest[n-1]` = 18. The array starts filled with 18.
- a_i = <k, m>: `late(i) = latest[m] - dur`. v7 at the latest 18 − 4 = 14, v6 at the latest 18 − 2 = 16.
- With several successors take the **smallest**: `latest[k] = min { latest[m] - dur }`. E.g. v0: 6 − 6 = 0 via v1, 6 − 4 = 2 via v2, 8 − 5 = 3 via v3 → **0**. Taking the largest (3) would delay v1 and push the project to 21.

```
if (latest[k] > latest[j] - p->duration)
    latest[k] = latest[j] - p->duration;
```

Since a9 = 2, the latest time of v6 is day **16** (as in the table), and both paths give 7 for v4; the min rule makes no difference there. The v0 example shows the min rule better.

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
18	18	18	18	18	18	18	18	18

stack (top at left): 8

Start: latest[] = 18; v8, the only vertex with no predecessors in the reversed graph, is on the stack.

Step 1 / 10

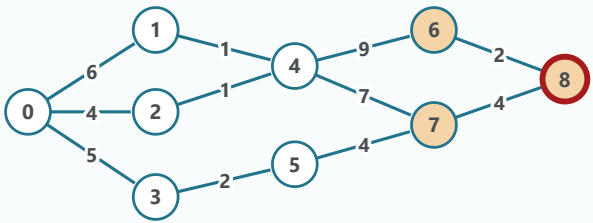
```
▶ j = top;  top = graph[top].count;   /* reversed list */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
      k = p->vertex;                        /* k: pred */
      if (latest[k] > latest[j] - p->duration)
          latest[k] = latest[j] - p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;
      }
  }
```

VARIABLES

j = – top = 8

OUTPUT

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
18	18	18	18	18	18	16	14	18

stack (top at left): 7, 6

v8 is popped (latest[8] = 18 final). Predecessors: latest[6] = 16, latest[7] = 14. Pushed: v6, v7.

Step 2 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list
  */
  printf("%d ", j);
  for (p = graph[j].link; p; p = p->link) {
▶     k = p->vertex;                      /* k: pred */
▶     if (latest[k] > latest[j] - p->duration)
▶         latest[k] = latest[j] - p->duration;
      if (--graph[k].count == 0) {
          graph[k].count = top;  top = k;
      }
  }
```

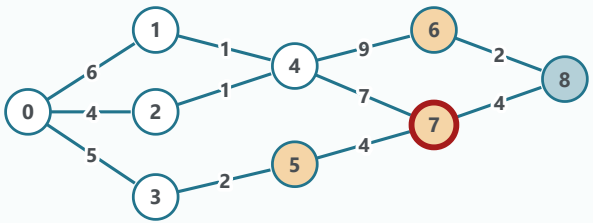
VARIABLES

`j = 8` `top = 7`

OUTPUT

```
8
```

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
18	18	18	18	7	10	16	14	18

stack (top at left): 5, 6

v7 is popped (latest[7] = 14 final). Predecessors: latest[4] = 7, latest[5] = 10. Pushed: v5.

Step 3 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list */
2 printf("%d ", j);
3 for (p = graph[j].link; p; p = p->link) {
▶     k = p->vertex;                      /* k: pred */
▶     if (latest[k] > latest[j] - p->duration)
▶         latest[k] = latest[j] - p->duration;
7     if (--graph[k].count == 0) {
8         graph[k].count = top;  top = k;
9     }
10 }
```

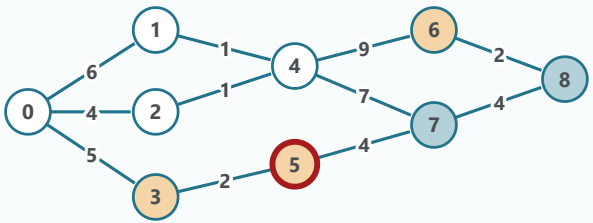
VARIABLES

j = 7 top = 5

OUTPUT

```
8 7
```

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
18	18	18	8	7	10	16	14	18

stack (top at left): 3, 6

v5 is popped (latest[5] = 10 final). Predecessors: latest[3] = 8. Pushed: v3.

Step 4 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list */
2 printf("%d ", j);
3 for (p = graph[j].link; p; p = p->link) {
▶     k = p->vertex;                      /* k: pred */
▶     if (latest[k] > latest[j] - p->duration)
▶         latest[k] = latest[j] - p->duration;
7     if (--graph[k].count == 0) {
8         graph[k].count = top;  top = k;
9     }
10 }
```

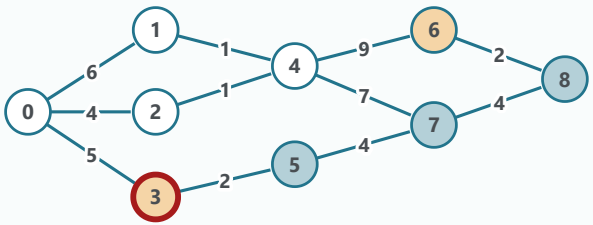
VARIABLES

j = 5 top = 3

OUTPUT

```
8 7 5
```

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
3	18	18	8	7	10	16	14	18

stack (top at left): 6

v3 is popped (latest[3] = 8 final). Predecessors: latest[0] = 3.

Step 5 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list */
2 printf("%d ", j);
3 for (p = graph[j].link; p; p = p->link) {
▶     k = p->vertex;                      /* k: pred */
▶     if (latest[k] > latest[j] - p->duration)
▶         latest[k] = latest[j] - p->duration;
7     if (--graph[k].count == 0) {
8         graph[k].count = top;  top = k;
9     }
10 }
```

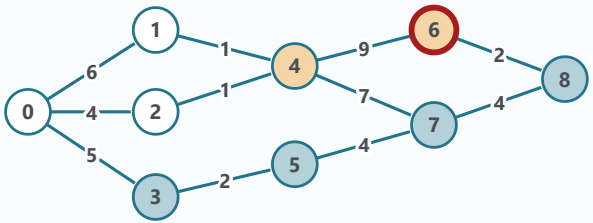
VARIABLES

j = 3 top = 6

OUTPUT

```
8 7 5 3
```

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
3	18	18	8	7	10	16	14	18

stack (top at left): 4

v6 is popped (latest[6] = 16 final). [4] does not change. Pushed: v4.

Step 6 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list */
2 printf("%d ", j);
3 for (p = graph[j].link; p; p = p->link) {
▶     k = p->vertex;                        /* k: pred */
▶     if (latest[k] > latest[j] - p->duration)
6         latest[k] = latest[j] - p->duration;
7     if (--graph[k].count == 0) {
8         graph[k].count = top;  top = k;
9     }
10 }
```

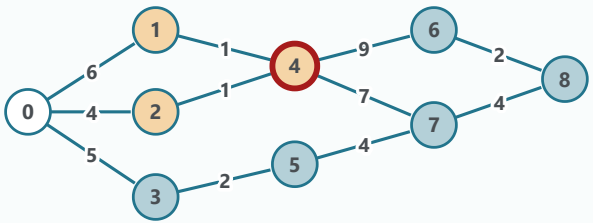
VARIABLES

j = 6 top = 4

OUTPUT

```
8 7 5 3 6
```

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
3	6	6	8	7	10	16	14	18

stack (top at left): 2, 1

v4 is popped (latest[4] = 7 final). Predecessors: latest[1] = 6, latest[2] = 6. Pushed: v1, v2.

Step 7 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list */
2 printf("%d ", j);
3 for (p = graph[j].link; p; p = p->link) {
▶     k = p->vertex;                      /* k: pred */
▶     if (latest[k] > latest[j] - p->duration)
▶         latest[k] = latest[j] - p->duration;
7     if (--graph[k].count == 0) {
8         graph[k].count = top;  top = k;
9     }
10 }
```

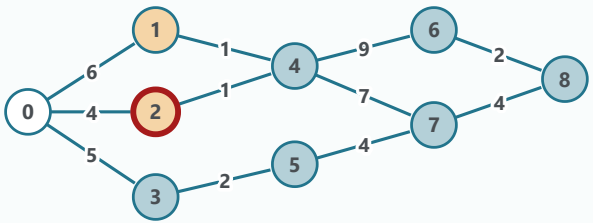
VARIABLES

j = 4 top = 2

OUTPUT

```
8 7 5 3 6 4
```

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
2	6	6	8	7	10	16	14	18

stack (top at left): 1

v2 is popped (latest[2] = 6 final). Predecessors: latest[0] = 2 (changed).

Step 8 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list */
2 printf("%d ", j);
3 for (p = graph[j].link; p; p = p->link) {
▶     k = p->vertex;                      /* k: pred */
▶     if (latest[k] > latest[j] - p->duration)
▶         latest[k] = latest[j] - p->duration;
7     if (--graph[k].count == 0) {
8         graph[k].count = top;  top = k;
9     }
10 }
```

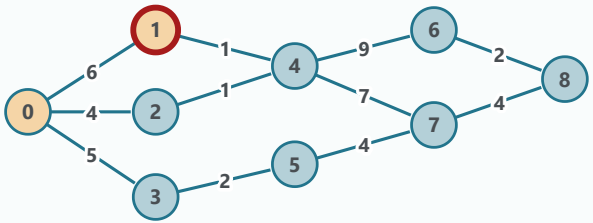
VARIABLES

j = 2 top = 1

OUTPUT

```
8 7 5 3 6 4 2
```

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
0	6	6	8	7	10	16	14	18

stack (top at left): 0

v1 is popped (latest[1] = 6 final). Predecessors: latest[0] = 0 (changed). Pushed: v0.

Step 9 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list */
2 printf("%d ", j);
3 for (p = graph[j].link; p; p = p->link) {
▶     k = p->vertex;                      /* k: pred */
▶     if (latest[k] > latest[j] - p->duration)
▶         latest[k] = latest[j] - p->duration;
7     if (--graph[k].count == 0) {
8         graph[k].count = top;  top = k;
9     }
10 }
```

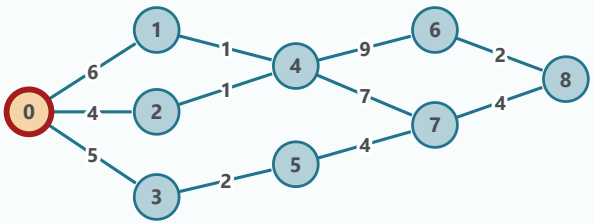
VARIABLES

j = 1 top = 0

OUTPUT

```
8 7 5 3 6 4 2 1
```

Code Trace: latest[] (on the Reversed Graph)



latest[]

0	1	2	3	4	5	6	7	8
0	6	6	8	7	10	16	14	18

stack (top at left): empty

v0 is popped (latest[0] = 0 final). No predecessors (the start event).

Step 10 / 10

```
▶ j = top;  top = graph[top].count;   /* reversed list */
2 printf("%d ", j);
3 for (p = graph[j].link; p; p = p->link) {
4     k = p->vertex;                        /* k: pred */
5     if (latest[k] > latest[j] - p->duration)
6         latest[k] = latest[j] - p->duration;
7     if (--graph[k].count == 0) {
8         graph[k].count = top;  top = k;
9     }
10 }
```

VARIABLES

j = 0 top = −1

OUTPUT

```
8 7 5 3 6 4 2 1 0
```

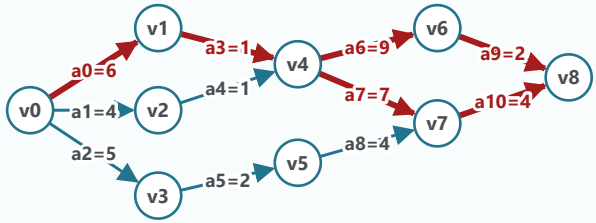
Worksheet for the Latest Start Times

latest	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	Stack
Initial	18	18	18	18	18	18	18	18	18	8
Output v8	18	18	18	18	18	18	16	14	18	7, 6
Output v7	18	18	18	18	7	10	16	14	18	5, 6
Output v5	18	18	18	8	7	10	16	14	18	3, 6
Output v3	3	18	18	8	7	10	16	14	18	6
Output v6	3	18	18	8	7	10	16	14	18	4
Output v4	3	6	6	8	7	10	16	14	18	2, 1
Output v2	2	6	6	8	7	10	16	14	18	1
Output v1	0	6	6	8	7	10	16	14	18	0
Output v0	0	6	6	8	7	10	16	14	18	–

[0] was updated three times: 8 − 5 = 3 from v3, 6 − 4 = 2 from v2, 6 − 6 = **0** from v1.

Slack and Critical Activities

Act.	Edge	Dur.	Earliest	Latest	Slack
a0	(0,1)	6	0	0	**0: critical**
a1	(0,2)	4	0	2	2
a2	(0,3)	5	0	3	3
a3	(1,4)	1	6	6	**0: critical**
a4	(2,4)	1	4	6	2
a5	(3,5)	2	5	8	3
a6	(4,6)	9	7	7	**0: critical**
a7	(4,7)	7	7	7	**0: critical**
a8	(5,7)	4	7	10	3
a9	(6,8)	2	16	16	**0: critical**
a10	(7,8)	4	14	14	**0: critical**



thick red edges: slack 0 (critical)

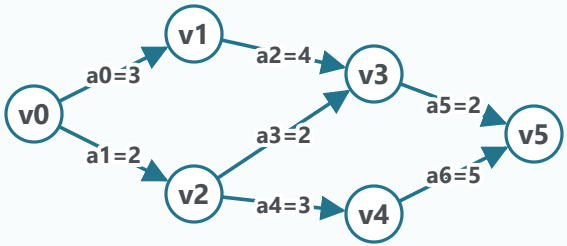
- Slack = latest − earliest (a_i = <k,m>: latest[m] − dur − earliest[k]). E.g. a4: earliest 4, latest 6 → it can slip **2 days**.
- An activity with slack 0 is **critical**: it must start at its earliest time. Project-management software charts the slack.
- This is **total float**: the 3 days of slack on the path a2-a5-a8 are shared; if a2 or a5 uses them, a8 becomes critical.

Let the Program Find the Critical Path (Extra Topic)

```
/* after computing earliest[] and latest[] */
for (i = 0; i < n; i++)
    for (p = graph[i].link; p; p = p->link) {
        int early = earliest[i];
        int late  = latest[p->vertex] - p->duration;
        printf("<%d,%d> early=%d late=%d slack=%d%s\n",
               i, p->vertex, early, late, late - early,
               late == early ? " CRITICAL" : "");
    }
```

- The critical activities form one (or more) paths from v0 to v8: the **critical path**.
- The whole computation: forward pass O(n + e) + transposed lists O(n + e) + backward pass O(n + e) = **O(n + e)**.
- In a DAG the “longest path” is easy in topological order; in a general graph the longest simple path is NP-hard.
- **Free float**: slack usable without delaying the earliest time of any successor; 0 for a2 and a5, 3 for a8.

Exam Question: Run the Earliest / Latest Times



1. Write the topological order and the stack at every step.
2. Fill in the earliest[] and latest[] tables.
3. Find the slack of every activity; mark the critical path.

> **Instructor:** "'Which topics will be on the exam?' Anything we covered. It can come from Data Structures 1 too."

▶ **Solution**

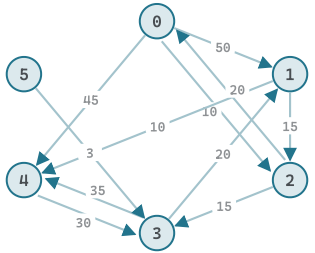
Lab: Dijkstra

shortest_path(0, …): 6 vertices

Input (directed edges, e.g. 0>1:50) `0>1:50, 0>2:10, 0>4:45, 1>2:15, 1>4:10, 2>0:20, 2>3:15, 3>:` Apply

CODE

```
 1 for (i = 0; i < n; i++) {        /* A1 */
 2     found[i] = FALSE;
 3     dist[i] = cost[v][i];
 4     pred[i] = v;
 5 }
 6 found[v] = TRUE;  dist[v] = 0;
 7 for (i = 2; i < n; i++) {     /* n-2 rounds …
 8   u = choose(dist, n, found);
 9   if (u < 0) break; /* none left */
10   found[u] = TRUE;
11   for (w = 0; w < n; w++)
12     if (!found[w] && dist[u] +
13         cost[u][w] < dist[w]) {
14       dist[w] = dist[u] +
```



dist

pred

OUTPUT (none yet)

`shortest_path(0, …)` : the source is 0. A missing edge costs `INF = 1000` (∞). The number next to a vertex is its current `dist` value.

SUMMARY

Complexity Summary

Algorithm	Structure	Time
Brute force (one target)	all paths	$O(n^n)$
Dijkstra (basic)	cost matrix + arrays	$O(n^2)$
Dijkstra (improved)	min heap + adjacency lists	$O((n + e) \log n)$
Printing the path	predecessor array	O(path length)
Bellman-Ford	edge list	$O(n \cdot e)$
Floyd-Warshall	$n \times n$ matrix	$O(n^3)$
Topological order	lists with counts + embedded stack	$O(n + e)$
Earliest / latest times, slack	topological order + inverse lists	$O(n + e)$

SUMMARY

Common Mistakes

1. Using Dijkstra on a graph with negative edges.
2. Choosing INT_MAX for ∞ and overflowing in `dist[u] + cost`.
3. Taking the **min** for earliest times and the **max** for latest times (it is the other way round!).
4. Computing a latest time before its successors are final: the order must be the topological order of the reversed graph (or the forward order reversed).
5. Starting the topological order at an arbitrary vertex; treating a vertex as final before its predecessors are processed.
6. Trying to build the path forward instead of using the predecessor array backward.

Exercises

1. On the directed sample graph, which distances would shortest_path find from source 2?
2. Why does the outer loop of shortest_path run n − 2 rounds? What if it ran n − 1?
3. In the step-by-step example (v0…v6), if edge v3–v6 cost 15 instead of 2, in what order would the vertices join S?
4. If topsort used a queue instead of a stack, what would it print for the v0..v5 example?
5. In the AoE example, if a8 took 8 instead of 4, what would the project length and critical path be?
6. In the Floyd-Warshall example, why did A[1][0] change in round k = 2?

▶ **Answers / hints**

SUMMARY

Chapter Summary

Shortest paths

Brute force $O(n^n)$; Dijkstra is greedy and needs $w \geq 0$: $O(n^2)$ or $O((n + e) \log n)$.

The path

The predecessor array alone is enough; print it backward (recursion/stack).

Generalizations

Negative edges: Bellman-Ford $O(ne)$. All pairs: Floyd-Warshall $O(n^3)$.

AoV and topological order

Counts + an embedded stack; $O(n + e)$; it also detects cycles.

AoE and times

Forward: earliest with max; backward (reversed graph): latest with min. Slack = late − early.

Critical path

The activities with zero slack; they set the project's length.

SUMMARY

References

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., §6.4 Shortest Paths, §6.5 Activity Networks.
- T. H. Cormen et al., *Introduction to Algorithms*, 4th ed., Section 20.4 (Topological Sort), Chapters 22 (Single-Source Shortest Paths) and 23 (All-Pairs).
- R. Sedgewick, *Algorithms in C, Part 5*, Chapter 21 (Shortest Paths).

← Chapter 17: Minimum Spanning Trees

Course page

Chapter 19: Basic Sorting Algorithms

The End

Chapter 18: Shortest Paths and Activity Networks

Next chapter: Chapter 19: Basic Sorting Algorithms · Course page