# Traveling Salesman

Find a minimum-cost tour in a complete graph.

# Knapsack

Given $0 \leq W$, $[1 \leq v_1, \ldots, v_N]$, and $[1 \leq w_1, \ldots, w_N]$, maximize $\sum^N v_i x_i$ with $\sum^N w_i x_i \leq W$.

## 0/1 Knapsack

When $0 \leq x_i \leq 1$.

| | |
|---|---|
| **Time** | $O(NW)$ |
| **Space** | $O(W)$ (store $\text{dp}[n]$ and $\text{dp}[n-1]$) |

### Solution

Let $\text{dp}[n][w]$ be the solution when $W = w$ and $N = n$.

- $\text{dp}[0][w] = 0$
- $\text{dp}[n][w] = \max\left(\text{dp}[n-1][w],\ \text{dp}[n-1][w - w_i] + v_i\right)$ (for $w \geq w_i$)

## Bounded Knapsack

When $0 \leq x_i \leq k$.

| | |
|---|---|
| **Time** | $O(kNW)$ |
| **Space** | $O(W)$ (store $\text{dp}[n]$ and $\text{dp}[n-1]$) |

### Solution

Solve the 0/1 knapsack problem where each $v_i$, $w_i$ implicitly appears $k$ times ($N' = kN$).

## Unbounded Knapsack

When $0 \leq x_i < \infty$.

| | |
|---|---|
| **Time** | $O(W)$ |
| **Space** | $O(W)$ |

### Solution

Let $\text{dp}[w]$ be the solution when $W = w$.

- $\text{dp}[0] = 0$
- $\text{dp}[w] = \max\limits_{1 \leq i \leq N} \{\text{dp}[w - w_i] + v_i\}$ (for $w \geq w_i$)

# BFS

| | |
|---|---|
| **Time** | $\lvert V\rvert + \lvert E\rvert$ |
| **Space** | $\lvert V\rvert$ |

## Data Structures

| Name | Type | Initial Value |
|---|---|---|
| `front` | `Queue<Vertex>` | `[start]` |
| `seen` | `Set<Vertex>` | `{start}` |

## Algorithm

```
while (!front.empty()) {
    Vertex u = front.top();
    front.pop();

    // Visit u

    for (Vertex v : E[u]) {
        if (seen.has(v)) continue;
        seen.add(v);

        // See u → v

        front.push(v);
    }
}
```

## Results

- `seen` is the set of vertices connected to `start`.

# Prim's Algorithm

| | |
|---|---|
| **Time** | $(\lvert E\rvert + \lvert V\rvert) \log \lvert V\rvert$ |
| **Space** | $\lvert V\rvert^2$ |

## Data Structures

| Name | Type | Initial Value |
|---|---|---|
| `front` | `PriorityQueue<(Weight, Vertex)>` | `[(0, start)]` |
| `visited` | `Set<Vertex>` | `{}` |
| `parent` | `Map<Vertex, Vertex>` | `{}` |
| `cost` | `Map<Vertex, Weight>` | `{}` |
| `tree` | `Map<Vertex, List<(Weight, Vertex)>>` | `{}` |

## Algorithm

```
while (!front.empty()) {
    (Weight w, Vertex u) = front.top();
    front.pop();

    if (visited.has(u)) continue;
    visited.add(u);

    // Visit u

    if (parent.has(u)) {
        tree[u].push((w, parent[u]));
        tree[parent[u]].push((w, u));

        // Connect parent[u] to u
    }

    for ((Vertex v, Weight x) : E[u]) {
        if (!cost.has(v) || cost[v] > x) {
            cost[v] = x;
            parent[v] = u;

            // Relax u → v

            front.push((x, v));
        }
    }
}
```

## Results

- `tree` is **some** MST of `start`'s connected component.

## Notes

- Fails on directed graphs.

# DFS

| | |
|---|---|
| **Time** | $\|V\| + \|E\|$ |
| **Space** | $\|V\|$ |

## Data Structures

| Name | Type | Initial Value |
|---|---|---|
| backtrack | Stack<Vertex> | [start] |
| visited | Set<Vertex> | {} |

## Algorithm

```
while (!backtrack.empty()) {
    Vertex u = backtrack.top();

    if (!visited.has(u)) {
        visited.add(u);

        // Start visiting u
    } else {
        // Backtrack to u
    }

    bool follow = false;
    for (Vertex v : E[u]) {
        if (visited.has(v)) continue;

        // Follow u → v

        backtrack.push(v);
        follow = true;
        break;
    }
    if (follow) continue;

    // Finish visiting u

    backtrack.pop();
}
```

## Results

- `visited` is the set of vertices connected to `start`.

# Floyd-Warshall

| | |
|---|---|
| **Time** | $\lvert V\rvert^3$ |
| **Space** | $\lvert V\rvert^2$ |

## Data Structures

| Name | Type | Initial Value |
|---|---|---|
| `next?` | `Map<(Vertex, Vertex), Vertex>` | `{E(u, v): v}` |
| `dist` | `Map<(Vertex, Vertex), Distance>` | `{E(u, v): w, V(v, v): 0}` |

## Algorithm

```
for (Vertex m : V) {
    for (Vertex u : V) {
        for (Vertex v : V) {
            if (!dist.has((u, m)) || !dist.has((m, v))) continue;

            if (dist[u, v] > dist[u, m] + dist[m, v]) {
                dist[u, v] = dist[u, m] + dist[m, v];
                next[u, v] = next[u, m];

                // Relax u → v through m
            }
        }
    }
}

for (Vertex v : V) {
    if (dist[v, v] < 0) {
        return false; // Negative cycle detected
    }
}

return true;
```

## Results

- `dist[u, v]` is the distance from `u` to `v` (if they are connected).
- `next[u, v]` is the second vertex on **some** shortest path from `u` to `v` (if they are connected and distinct).

## Notes

- Johnson’s Algorithm is faster for sparse graphs.
- Fails on negative cycles (detected).

# Bellman-Ford

| | |
|---|---|
| **Time** | $\lvert V\rvert \cdot \lvert E\rvert$ |
| **Space** | $\lvert V\rvert$ |

## Data Structures

| Name | Type | Initial Value |
|---|---|---|
| `prev?` | `Map<Vertex, Vertex>` | `{}` |
| `dist` | `Map<Vertex, Distance>` | `{start: 0}` |

## Algorithm

```
for (|V| - 1) {
    for ((Vertex u, Vertex v, Weight w) : E) {
        if (!dist.has(u)) continue;

        if (!dist.has(v) || dist[v] > dist[u] + w) {
            dist[v] = dist[u] + w;
            prev[v] = u;

            // Relax u → v
        }
    }
}

// Extra iteration
for ((Vertex u, Vertex v, Weight w) : E) {
    if (dist.has(u) && dist[v] > dist[u] + w) {
        return false; // Negative cycle detected
    }
}

return true;
```

## Results

- `dist[v]` is the distance from `start` to `v` (if they are connected).
- `prev[v]` is the penultimate vertex on **some** shortest path from `start` to `v` (if they are connected).

## Notes

- The extra iteration will relax some vertex iff a negative cycle is reachable from `start`.
- `|V| - 1` extra iterations will relax `v` iff there is a negative cycle between `start` and `v`.
- Fails on negative cycles (detected).

# Johnson's Algorithm

| | |
|---|---|
| **Time** | $(\lvert E\rvert + \lvert V\rvert)\lvert V\rvert \log \lvert V\rvert$ |
| **Space** | $\lvert V\rvert^2$ |

## Data Structures

| Name | Type | Initial Value |
|---|---|---|
| `adjusted` | `Map<Vertex, List<(Weight, Vertex)>>` | `G + {q: [V(0, v)]}` |
| `height` | `Map<Vertex, Distance>` | `{}` |
| `prev?` | `Map<Vertex, Map<Vertex, Vertex>>` | `{}` |
| `dist` | `Map<Vertex, Map<Vertex, Distance>>` | `{}` |

## Algorithm

```
if (!BellmanFord(adjusted, &height, q)) return false;
adjusted.remove(q);

// Reweighting
for (Vertex u : V) {
    for ((Weight w, Vertex v) : adjusted[u]) {
        w += height[u] - height[v];
    }
}

// Repeated Dijkstra
for (Vertex v : V) {
    (dist[v], prev[v]) = Dijkstra(adjusted, v);
}

return true;
```

## Results

- `dist[u][v] - height[u] + height[v]` is the distance from `u` to `v` (if they are connected).
- `prev[u][v]` is the penultimate vertex on **some** shortest path from `u` to `v` (if they are connected).

## Notes

- Bellman-Ford & reweighting can be skipped for graphs with non-negative edges.
- Fails on negative cycles (detected during Bellman-Ford).

# Dijkstra’s Algorithm

| | |
|---|---|
| **Time** | $(\|E\| + \|V\|) \log \|V\|$ |
| **Space** | $\|V\|^2$ |

## Data Structures

| Name | Type | Initial Value |
|---|---|---|
| `front` | `PriorityQueue<(Distance, Vertex)>` | `[(0, start)]` |
| `visited` | `Set<Vertex>` | `{}` |
| `prev?` | `Map<Vertex, Vertex>` | `{}` |
| `dist` | `Map<Vertex, Distance>` | `{start: 0}` |

## Algorithm

```
while (!front.empty()) {
    (Distance d, Vertex u) = front.top();
    front.pop();

    if (visited.has(u)) continue;
    visited.add(u);

    // Visit u

    for ((Vertex v, Weight w) : E[u]) {
        Distance r = d + w;
        if (!dist.has(v) || dist[v] > r) {
            dist[v] = r;
            prev[v] = u;

            // Relax u → v

            front.push((r, v));
        }
    }
}
```

## Results

- `dist[v]` is the distance from `start` to `v` (if they are connected).
- `prev[v]` is the penultimate vertex on **some** shortest path from `start` to `v` (if they are connected).

## Notes

- Fails on graphs with negative edges (use Bellman-Ford).

# Topological Sort

| | |
|---|---|
| **Time** | $\lvert V\rvert + \lvert E\rvert$ |
| **Space** | $\lvert V\rvert$ |

## Data Structures

| Name | Type | Initial Value |
|---|---|---|
| `sorted` | `Set<Vertex>` | `{}` |
| `topo` | `List<Vertex>` | `[]` |

## Algorithm

```
for (Vertex start : V) {
    if (sorted.has(start)) continue;

    Set<Vertex> visited = {};
    Stack<Vertex> backtrack = [start];

    while (!backtrack.empty()) {
        Vertex u = backtrack.top();
        visited.add(u);

        bool follow = false;
        for (Vertex v : E[u]) {
            if (sorted.has(v)) continue;
            if (visited.has(v)) return false; // Cycle detected

            backtrack.push(v);
            follow = true;
            break;
        }
        if (follow) continue;

        sorted.add(u);
        topo.push(u);
        backtrack.pop();
    }
}

return true;
```

## Results

- `i < j` implies there is no path from `topo[i]` to `topo[j]` (reverse topological order).

## Notes

- Impossible with cycles (detected).

# C++ Tricks

- `set` is better than `priority_queue` for Prim's and Dijkstra's:

| | `set<T>` | `priority_queue<T, vector<T>, greater<T>>` |
|---|---|---|
| **Insert** | `q.insert(x)` | `q.push(x)` |
| **Top** | `*q.begin()` | `q.top()` |
| **Pop** | `q.erase(q.begin())` | `q.pop()` |
| **Delete** | `q.erase(q.find(x))` | N/A |

# C++ Prelude

```cpp
#include <bits/stdc++.h>
using namespace std;

#define DEBUG 1
#define dbg(x) (DEBUG ? _d((#x), (x)) : (x))
#define mod(x, m) ((((x) % (m)) + (m)) % (m))
#define _f (k ? '\n' + string(f, ' ') : "")

template <class T> auto _s(T x,...) -> decltype(to_string(x)) { return to_string(x); }
string _s(char x,...) { return string("'") + x + "'"; }
string _s(string x,...) { return '"' + x + '"'; }

template <class P, class Q> string _s(pair<P, Q> x, int f=0, int k=0) {
    return _f + '(' + _s(x.first) + ", " + _s(x.second) + ')';
}

template <class T> auto _s(T x, int f=0, int k=0) -> decltype(end(x), string()) {
    string s; int i = 0; auto b = begin(x), e = end(x);
    while (b != e) s += _s(*b++, f+1, i++), s += (b == e ? "" : ", ");
    return _f + '[' + s + ']';
}

template <class T> T& _d(string s, T&& x) {
    cout << s + " = " + _s(x, s.size() + 3) + '\n'; return x;
}

typedef vector<int> vi;
typedef pair<int, int> ii;
typedef int64_t i64;
typedef uint64_t u64;

const double pi = 2 * acos(0.0);
const double dinf = 1.0 / 0.0;
const int inf = numeric_limits<int>::max() >> 2;
const long linf = numeric_limits<i64>::max() >> 2;
```

# Java Prelude

# Speed Estimates

# Sample Graphs

```
10
0 0 (no vertices)
9 0 (no edges)
1 1 (self-loop)
0 0 5
2 2 (parallel edges)
0 1 0
0 1 2
3 3 (cycle)
0 1 0
1 2 2
2 0 4
4 2 (disconnected)
0 1 0
2 3 2
3 3 (non-consecutive)
9 3 0
7 3 2
7 9 4
3 3 (negative edge)
0 1 -2
1 2 4
2 0 0
3 3 (negative cycle)
0 1 -4
1 2 2
2 0 0
5 10 (complete graph)
0 1
0 2
0 3
0 4
1 2
1 3
1 4
1 5
2 3
2 4
2 5
3 4
3 5
4 5
```

## Other possibilities

- Dense or sparse
- Multiple solutions (e.g., shortest paths, MSTs, topological sorts)