CS4335 Design and Analysis of Algorithms

Greedy Algorithms

Interval Scheduling & Partitioning

Interval Scheduling Problem: Given a set of intervals, find the maximum number of non-overlapping intervals that can be selected.

Solution: Sort the intervals by their end times. Select the interval that does not overlap with the previous one.

```
int solve(pair<int, int> intervals[], int n) {
    sort(intervals, intervals + n, [](const auto& a, const auto& b) {
        return a.second < b.second;
    });
    int ans = 0, last = INT_MIN;
    for (int i = 0; i < n; i++) {
        if (intervals[i].first >= last) {
            ans++;
            last = intervals[i].second;
        }
    }
    return ans;
}
```

Interval Partitioning Problem: Given a set of intervals, find the minimum number of resources needed to schedule all intervals.

Solution: Sort the intervals by their start times. For each interval, assign it to the resource that finishes the earliest. If no resource is available, add a new resource.

```
int solve(pair<int, int> intervals[], int n) {
    sort(intervals, intervals + n, [](const auto& a, const auto& b) {
        return a.first < b.first;
    });
    priority_queue<int, vector<int>, greater<int>> pq; // the earliest end time will be used to schedule the next interval
    for (int i = 0; i < n; i++) {
        if (!pq.empty() && pq.top() <= intervals[i].first) {
            pq.pop();
        }
        pq.push(intervals[i].second);
    }
    return pq.size();
}
```

Fractional Knapsack Problem

Given a set of items, each with a weight w_i and a value v_i, and a knapsack with a maximum weight capacity W, find the maximum total value that can be put into the knapsack, allowing fractions of items to be taken.

Solution: Greedy on the value-to-weight ratio.

```
double solve(pair<int, int> items[], int n, int W) {
    sort(items, items + n, [](const auto& a, const auto& b) {
        return (double)a.second / a.first > (double)b.second / b.first;
    });
    double ans = 0;
    for (int i = 0; i < n; i++) {
        if (W == 0) break;
        int take = min(W, items[i].first);
        ans += (double)take * items[i].second / items[i].first;
        W -= take;
    }
    return ans;
}
```

Minimize Lateness

Given n jobs, each with a deadline d_i and a processing time t_i, find the order to minimize the total lateness.

Solution: Sort the jobs by their deadlines. Schedule the jobs in the order of the sorted list.

```
int solve(pair<int, int> jobs[], int n) {
    sort(jobs, jobs + n, [](const auto& a, const auto& b) {
        return a.first < b.first;
    });
    int time = 0, lateness = 0;
    for (int i = 0; i < n; i++) {
        time += jobs[i].second;
        lateness += max(0, time - jobs[i].first);
    }
    return lateness;
}
```

Inversion: A pair of indices (i, j) is an inversion if $i < j$ and $a[i] > a[j]$.

In minimize lateness problem, if there is an inversion (i, j), then $d_i < d_j$ but j is scheduled before i. This inversion can be removed by swapping i and j.

The greedy solution is optimal because it removes all inversions.

Proof of Optimality:

- Observe. If there is an inversion, there is at least one occurence of adjacent inversions. Denote the pair as (i, j).
- Claim. Swapping two adjacent, inverted jobs reduces the number of inversions by 1; and does not increase the total lateness.
- Proof. Let l and l' be the lateness before and after the swap.

- $l'_k = l_k$ for $k \neq i, j$.
- $l'_i \leq l_i$ because $l'_i = \max(0, f'_i - d_i) \leq \max(0, f_i - d_i) = l_i$.
- $l'_j = f'_j - d_j = f_i - d_j \leq f_i - d_i = l_i$ (where f_k is the finish time of job k and $d_i \leq d_j$).
- Therefore, $l'_i + l'_j \leq l_i + l_j$.
- The total lateness is the sum of lateness of all jobs, so the total lateness is also reduced.

Minimize Average Waiting Time

Given n jobs, all arriving at time 0, each with a processing time t_i, find the order to minimize the average waiting time.

Solution: Sort the jobs by their processing times. Schedule the jobs in the order of the sorted list.

On Greedy Algorithm Design

Example 1: Given n points on a line, find the minimum number of intervals of length k that cover all points.

Solution: Greedy on the leftmost point of the interval.

Description of algorithm:

1. Sort the points by their values ascendingly.
2. Take the leftmost point x_0 as the left endpoint of the interval.
3. Find the rightmost point x_1 such that $x_1 - x_0 \leq k$.
4. Repeat step 2 and 3 until all points are covered.

Psuedocode:

```
def solve(points, k):
    points.sort()
    ans = 0
    i = 0
    while i < len(points):
        ans += 1
        x0 = points[i]
        while i < len(points) and points[i] - x0 <= k:
            i += 1
    return ans
```

Proof of Optimality:

Denote $\{G_1, G_2, \ldots, G_m\}$ as the greedy solution and $\{O_1, O_2, \ldots, O_n\}$ as the optimal solution, where $G_i = [x_i, x_i + k]$ and $O_i = [y_i, y_i + k]$.

The greedy solution is optimal if and only if $m = n$. Therefore, we need to prove $m = n$.

Assume for all $1 \leq i < r$, $G_i = O_i$ and $G_r \neq O_r (2 \leq r \leq n)$.

Because G_r always begins at the smallest element in the remaining set, we can replace every O_k with G_k such that elements covered by O_k are also covered by G_k.

Therefore, we can keep replacing O_r with G_r until two solutions become exactly the same, during which the number of intervals remains the same.

Therefore, $m = n$ and the greedy solution is optimal.

Graph Theory

Introduction

- **Graph** $G = (V, E)$ where V is the set of vertices and E is the set of edges.
- **Path** $P = (v_1, v_2, \ldots, v_k)$ is a sequence of vertices such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$.
  - v_i and v_j could be the same (i.e., visiting the same vertex multiple times).
- **Circuit** is a path that starts and ends at the same vertex, i.e., $v_1 = v_k$.
- **Degree** of a vertex v is the number of edges incident on v.

Euler Circuit: A circuit that visits every edge exactly once.

- **Euler's Theorem**: A connected graph has an Euler circuit if and only if every vertex has even degree.

Finding an Euler circuit:

1. Starting with any vertex u in G, take an unused edge (u,v) (if there is any) incident to u
2. Do Step 1 for v and continue the process until v has no unused edge. (a circuit C is obtained)
3. If every node in C has no unused edge, stop.
4. Otherwise, select a vertex, say, u in C, with some unused edge incident to u and do Steps 1 and 2 until another circuit is obtained.
5. Merge the two circuits obtained to form one circuit
6. Goto Step 3.

It can be proved that, for a given graph G, one can always add some (>0) edges to G to create a new graph with an Euler circuit.

Representation of Graphs

1. **Adjacency Matrix**: $A_{ij} = 1$ if $(v_i, v_j) \in E$.
   - Space complexity: $O(V^2)$
   - Checking if $(v_i, v_j) \in E$: $O(1)$
   - Finding all neighbors of a vertex: $O(V)$
2. **Adjacency List**: For each vertex v_i, store a list of vertices adjacent to v_i.
   - Space complexity: $O(V + E)$
   - Checking if $(v_i, v_j) \in E$: $O(\text{degree}(v_i))$
   - Finding all neighbors of a vertex: $O(\text{degree}(v_i))$

```
typedef struct {
    int u, v, w;
} edge;
vector<vector<edge>> adj; // each vertex has a list of edges

void add_edge(int u, int v, int w) {
    adj[u].push_back({u, v, w});
    adj[v].push_back({v, u, w}); // for undirected graph
}

void print_adj_list() {
    for (int u = 0; u < adj.size(); u++) {
        cout << u << ": ";
```

```
        for (auto& e : adj[u]) {
            cout << e.v << " ";
        }
        cout << endl;
    }
}

int degree(int u) {
    return adj[u].size();
}
```

Minimum Spanning Tree

Spanning Tree: A subgraph of a graph G that is a tree containing all the vertices of G.

Minimum Spanning Tree (MST): A spanning tree of a weighted graph G with the smallest possible sum of edge weights.

Generic MST Algorithm

```
def generic_mst(G):
    a = set()
    while a does not form a spanning tree:
        find an edge (u, v) that is safe to add to a
        a = a + {(u, v)}
    return a
```

- a is a set of edges forming a spanning tree. This property is called the **invariant property**.
- An edge (u, v) is **safe** to add to a if $a + (u, v)$ does not violate the invariant.

Definitions:

- A **cut** $(S, V - S)$ of a graph G is a partition of V into two non-empty sets S and $V - S$.
- An edge **crossess** the cut $(S, V - S)$ if one of its endpoints is in S and the other is in $V - S$.
- A **light edge** crossing a cut is an edge with the smallest weight among all edges crossing the cut.

Theorem:

Given $G = (V, E)$ and a subset of MST $A \subseteq E$, If $(S, V - S)$ is a cut of G that respects A (i.e., no edge in A crosses the cut), then the light edge (u, v) crossing the cut is safe to add to A.

Kruskal's Algorithm

In Kruskal's Algorithm, set A is a forest. Initially, each vertex is a separate tree.

The safe edge (u, v) to add to A is the lightest edge that connects two trees (= connected components) in the forest.

Pseudocode:

```
def kruskal_mst(G):
    a = set()
    for each vertex v in G:
        make_set(v)
    sort the edges of G by weight
    for each edge (u, v) in G:
        if find_set(u) != find_set(v):
            a = a + {(u, v)}
            union(u, v)
    return a
```

Time complexity: $O(E \log E)$

Implementation with Disjoint Set:

```
struct edge {
    int u, v, w;
    bool operator<(const edge& e) const {
        return w < e.w;
    }
};
struct DSU {
    vector<int> parent;
    DSU(int n) {
        parent.resize(n);
        for (int i = 0; i < n; i++) {
            parent[i] = i;
        }
    }
    int find(int u) {
        return parent[u] == u ? u : parent[u] = find(parent[u]);
    }
    void unite(int u, int v) {
        parent[find(u)] = find(v);
    }
};

int kruskal_mst(vector<edge>& edges, int n) {
    sort(edges.begin(), edges.end());
    DSU dsu(n);
    int ans = 0;
    for (auto& edge : edges) {
        if (dsu.find(edge.u) != dsu.find(edge.v)) {
            dsu.unite(edge.u, edge.v);
            ans += edge.w;
        }
    }
    return ans;
}
```

Prim's Algorithm

In Prim's Algorithm, set A is a single tree.

The safe edge (u, v) to add to A is the lightest edge that connects a vertex in A to a vertex not in A.

Pseudocode:

```
def prim_mst(G):
    a = set()
    key = {v: ∞ for v in G}
    parent = {v: None for v in G}
    key[0] = 0
    Q = all vertices in G
    while Q is not empty:
        u = extract_min(Q)
        for each vertex v adjacent to u:
            if v in Q and weight(u, v) < key[v]:
                parent[v] = u
                key[v] = weight(u, v)
    return a
```

Time complexity: $O(V^2)$ with adjacency matrix, $O(E \log V)$ with adjacency list.

Implementation with Priority Queue:

```
struct node {
    int u, d;
    bool operator>(const node& n) const {
        return d > n.d;
    }
};
struct edge {
    int v, w;
};
// STL pq is a max heap which use > to compare
// to use as a min heap, use <T, vector<T>, greater<T>>
int prim_mst(vector<vector<edge>>& adj, int n) {
    vector<int> dist(n, INT_MAX);
    vector<int> parent(n, -1);
    priority_queue<node, vector<node>, greater<node>> pq;
    pq.push({0, 0});
    dist[0] = 0;

    int mst = 0;

    while (!pq.empty()) {
        node cur = pq.top();
        pq.pop();

        if (cur.d != dist[cur.u]) // skip if visited
            continue;

        mst += cur.d;
```

```
        for (edge &e : adj[cur.u]) {
            if (e.w < dist[e.v]) {
                dist[e.v] = e.w;
                parent[e.v] = cur.u;
                pq.push({e.v, e.w});
            }
        }
    }

    return mst;
}
```

To reconstruct the MST, we can store the parent array and build the MST from it.

```
void print(int u, vector<int>& parent) {
    if (parent[u] != -1) {
        print(parent[u], parent);
        cout << parent[u] << " ";
    }
}
```

Shortest Path

Given a graph $G = (V, E)$ and a source vertex s, the **shortest path** from s to v is the path with the smallest total weight.

If there are negative weight edges, Dijkstra's algorithm may not work. In this case, we can use Bellman-Ford algorithm.

Relaxation

Given π_v, the predecessor of v in the shortest path from s to v and d_v, the shortest distance from s to v:

The shortest path from s to v is the shortest path from s to π_v followed by the edge (π_v, v).

```
def relax(u, v, w):
    if d[v] > d[u] + w(u, v):
        d[v] = d[u] + w(u, v)
        pi[v] = u
```

Dijkstra's Algorithm

Greedy algorithm that finds the shortest path from a single source vertex to all other vertices.

Maintain two sets of vertices: S (final) and $V - S$ (tentative).

Notaion:

- S: set of vertices whose shortest path from s is already known. (Their distance is final.)
- $\pi[v]$: predecessor of v in the shortest path from s to v.
- $d[v]$: shortest distance from s to v.

Pseudocode:

```
def dijkstra(G, s):
    S = set()
    d = {v: ∞ for v in G}
    p = {v: None for v in G}
    d[s] = 0
    while V-S is not empty:
        u = vertex in V-S with the smallest d[u]
        S = S + {u}
        for each vertex v adjacent to u:
            relax(u, v, w)
```

Implementation with Priority Queue:

```
struct node {
    int u, d;
    bool operator>(const node& n) const {
        return d > n.d;
    }
};
struct edge {
    int v, w;
};
vector<int> dijkstra(vector<vector<edge>>& adj, int n, int s) {
    vector<int> d(n, INT_MAX);
    vector<int> pi(n, -1);
    d[s] = 0;
    priority_queue<node, vector<node>, greater<node>> pq; // node with smallest
d is on top
    pq.push({s, 0});

    while (!pq.empty()) {
        node u = pq.top();
        pq.pop();
        int _u = u.u; // current vertex
        for (auto& e : adj[_u]) {
            int v = e.v, w = e.w;
            if (d[v] > d[_u] + w) {
                d[v] = d[_u] + w;
                pi[v] = _u;
                pq.push({v, d[v]});
            }
        }
    }
    return d;
}
```

To reconstruct the shortest path, we can store the predecessor array and build the path from it.

```
void print(int u, vector<int>& pi) {
    if (pi[u] != -1) {
        print(pi[u], pi);
        cout << pi[u] << " ";
    }
}
```

Time complexity: $O((V+E)\log V)$, usually written as $O(E\log V)$.

Bellman-Ford Algorithm

Bellman-Ford algorithm can handle graphs with negative weight edges.

It is essentially a dynamic programming algorithm that relaxes all edges $|V|-1$ times.

Denote $OPT(i,v)$ as the shortest path P from s to v using at most i edges.

- Case 1: P uses at most $i-1$ edges.
  - $OPT(i,v) = OPT(i-1,v)$
- Case 2: P uses exactly i edges.
  - If $w \to v$ is the last edge of P, then $OPT(i,v) = OPT(i-1,w) + C_{wv}$, where $s \to w$ is a shortest path from s to w using at most $i-1$ edges.

$$OPT(i,v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min\{OPT(i-1,v), \min_{(w,v)\in E}\{OPT(i-1,w) + C_{wv}\}\} & \text{otherwise} \end{cases}$$

If there is no negative cycle, the algorithm will terminate after $|V|-1$ iterations.

Otherwise, the algorithm will detect the negative cycle in the $|V|$-th iteration.

Pseudocode:

```
def bellman_ford(G, s):
    d = {v: ∞ for v in G}
    p = {v: None for v in G}
    d[s] = 0
    for i = 1 to |V|-1:
        for each edge (u, v) in G:
            relax(u, v, w)
    for each edge (u, v) in G: # |v|-th iteration
        if d[v] > d[u] + w(u, v):
            return False
    return True
```

Time complexity: $O(VE)$

Corollary: if negative weight circuits exist, in the n-th iteration, the shortest path from s to some vertex v will be reduced.

Maximum Flow

A **flow network** $G = (V, E)$ is a directed graph where each edge (u, v) has a capacity $c(u, v) \geq 0$.

- If $(u, v) \notin E$, then $c(u, v) = 0$.

The maximum flow problem is to find the maximum flow from a source vertex s to a sink vertex t.

A flow in G is a real-valued function $f : E \to \mathbb{R}$ that satisfies the following properties:

- **Capacity constraint**: $0 \leq f(u, v) \leq c(u, v)$ for all $u, v \in V$. i.e. the flow on each edge is non-negative and does not exceed the capacity.
- **Flow conservation**: For all $u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$. i.e. the flow into a vertex equals the flow out of the vertex.

$f(u, v)$ is the **net flow** from u to v.

The value of a flow f is defined as $|f| = \sum_{v \in V} f(s, v)$, i.e. the total flow out of the source.

Residual Network

Given a flow network $G = (V, E)$ and a flow f, the **residual network** $G_f = (V, E_f)$ consists of edges with residual (remaining) capacity. Mathematically, $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$.

- For each edge $(u, v) \in E$, if $f(u, v) < c(u, v)$, then $(u, v) \in E_f$ with capacity $c_f(u, v) = c(u, v) - f(u, v)$.
- This is equivalent to $c_f(u, v) = c(u, v) + f(v, u)$.

Theorem: If f is a flow in G and f' is a flow in G_f, then $f + f'$ is a flow in G with value $|f + f'| = |f| + |f'|$.

Augment Path: Given a flow network $G = (V, E)$ and a flow f, an augment path p is a simple path from s to t in G_f.

- Residual capacity of an augment path is the min residual capacity of its edges.

$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is in } p\}$

Ford-Fulkerson Algorithm

Psuedocode:

```
def Ford-Fulkerson(G, s, t)
    for each edge (u, v) in G:
        f[u, v] = 0
        f[v, u] = 0
    while there is an augment path p in G_f:
        c_f(p) = min{c_f(u, v): (u, v) is in p}
        for each edge (u, v) in p:
            f[u, v] = f[u, v] + c_f(p)
            f[v, u] = -f[u, v]
    return f
```

Time complexity: $O(E|f^*|)$, where f^* is the maximum flow. The complexity is **not polynomial** since f^* can be arbitrarily large.

Implementation:

```
struct edge {
    int v, c, f;
};
int n;
vector<vector<edge>> adj;
vector<bool> vis;

int dfs(int u, int t, int f) {
    if (u == t) return f;
    vis[u] = true;
    for (auto& e : adj[u]) {
        if (!vis[e.v] && e.c - e.f > 0) { // not visited and residual capacity >
0
            int df = dfs(e.v, t, min(f, e.c - e.f)); // find the minimum
residual capacity
            if (df > 0) {
                e.f += df;
                for (auto& re : adj[e.v]) {
                    if (re.v == u) {
                        re.f -= df;
                        break;
                    }
                }
                return df;
            }
        }
    }
    return 0;
}

int ford_fulkerson(int s, int t) {
    int max_flow = 0;
    while (true) {
        fill(vis.begin(), vis.end(), false);
        int flow = dfs(s, t, INT_MAX);
        if (flow == 0) break;
        max_flow += flow;
    }
    return max_flow;
}
```

Edmonds-Karp Algorithm: Ford-Fulkerson with BFS to find the augment path.

Time complexity: $O(VE^2)$

BFS finds the shortest (in terms of number of edges) augment path.

Maximum Bipartite Matching

A **bipartite graph** $G = (V, E)$ is a graph whose vertices can be divided into two disjoint sets $V = X \cup Y$ such that every edge connects a vertex in X to a vertex in Y, i.e. every edge $(u, v) \in E$ has $u \in X$ and $v \in Y$.

A **maximum bipartite matching** is a matching with the largest possible number of edges.

This problem can be solved using the maximum flow algorithm.

To form the corresponding flow network $G' = (V', E')$ of a bipartite graph $G = (V, E)$:

- Create a source vertex s and connect it to all vertices in X, from s to x.
- Create a sink vertex t and connect it to all vertices in Y, from y to t.
- Duplicat all edges in G, only in the direction from X to Y.
- All edges in G' have capacity 1.

The maximum flow in G' is the maximum bipartite matching in G.

Pseudocode:

```
def bipartite_matching(G, X, Y, s, t):
    G0 = create_flow_network(G, X, Y, s, t)
    f = Ford-Fulkerson(G0, s, t)
    return f

def create_flow_network(G, X, Y, s, t):
    G0 = create_empty_graph()
    for x in X:
        add_edge(s, x, 1)
    for (x, y) in G:
        add_edge(x, y, 1)
    for y in Y:
        add_edge(y, t, 1)
    return G0
```

Divide and Conquer

Merge Sort

Psuedocode:

```
def sort(A):
    if len(A) > 1:
        mid = len(A) // 2
        L = A[:mid]
        R = A[mid:]
        sort(L)
        sort(R)
        merge(L, R, A)
```

Time complexity: $O(n \log n)$

Auxiliary space: $O(n)$, i.e., merge sort is not in-place.

Proof of time complexity:

$$T(n) \geq T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + O(n)$$

where $T(n)$ is the time complexity of sorting an array of size n.

- $T(\lceil n/2 \rceil)$ and $T(\lfloor n/2 \rfloor)$ are the time complexities of sorting two halves.
- $O(n)$ is the time complexity of merging two sorted halves.

Let $n = 2^k$. The recurrence relation becomes:

$$
\begin{aligned}
T(n) &= n + 2T(\frac{n}{2}) \\
&= n + 2(\frac{n}{2} + 2T(\frac{n}{4})) \\
&= n + n + 4T(\frac{n}{4}) \\
&= n + n + n + 8T(\frac{n}{8}) \\
&= \ldots \\
&= kn = n \log n
\end{aligned}
$$

Proof by telescoping:

$$
\begin{aligned}
\frac{T(n)}{n} &= \frac{T(n/2)}{n/2} + \frac{n}{n} \\
&= \frac{T(n/4)}{n/4} + 1 + 1 \\
&= \ldots \\
&= 1 + 1 + \ldots + 1 = \log n
\end{aligned}
$$

Counting Inversions

Given an array A, an inversion is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. Find the number of inversions in A.

Solution: Merge sort with inversion counting.

Psuedocode:

```
def sort(A):
    if len(A) > 1:
        mid = len(A) // 2
        L = A[:mid]
        R = A[mid:]
        inv = sort(L) + sort(R) + merge(L, R, A)
    return inv
```

Implementation:

```
int merge(vector<int>& L, vector<int>& R, vector<int>& A) {
    int inv = 0;
    int i = 0, j = 0, k = 0;
    while (i < L.size() && j < R.size()) {
        if (L[i] <= R[j]) {
            A[k++] = L[i++];
        } else {
            A[k++] = R[j++];
            inv += L.size() - i; // each of a[i], a[i+1], ..., a[L.size()-1]
creates an inversion with a[j]
        }
    }
    while (i < L.size()) {
        A[k++] = L[i++];
    }
    while (j < R.size()) {
        A[k++] = R[j++];
```

```
    }
    return inv;
}

int count_inversions(vector<int>& A) {
    if (A.size() <= 1) return 0;
    int mid = A.size() / 2;
    vector<int> L(A.begin(), A.begin() + mid);
    vector<int> R(A.begin() + mid, A.end());
    int inv = count_inversions(L) + count_inversions(R);
    inv += merge(L, R, A);
    return inv;
}
```

Karatsuba Multiplication

Problem: compute the product of two large integers a and b faster than the naive $O(n^2)$ algorithm.

To multiply two N-digit numbers:

- Divide each number into two halves:
  - $a = w \cdot 10^{N/2} + x, b = y \cdot 10^{N/2} + z$
- Perform N/2-digit multiplications:
  - $p = w \cdot y, q = x \cdot z, r = (w + x)(y + z)$
- Compute the product:
  - $a \cdot b = p \cdot 10^N + (r - p - q) \cdot 10^{N/2} + q$

Proof:

$$a \cdot b = w \cdot y \cdot 10^N + (w \cdot z + x \cdot y) \cdot 10^{N/2} + x \cdot z$$

Time complexity: $O(n^{\log_2 3}) \approx O(n^{1.585})$

$$T(N) \leq 3T(N/2) + O(N)$$

On Divide and Conquer

Example 1: Given an array, find the largest and second largest elements using divide and conquer.

```
pair<int, int> solve(vector<int>& A, int l, int r) {
    if (l == r) {
        return {A[l], INT_MIN};
    }
    int mid = l + (r - l) / 2;
    auto L = solve(A, l, mid);
    auto R = solve(A, mid + 1, r);
    int largest = max(L.first, R.first);
    int second = max(min(L.first, R.first), max(L.second, R.second));
    return {largest, second};
}
```

i.e.

```
    if (L.first > R.first && R.first > L.second) second = R.first;
    if (L.first > R.first && R.first < L.second) second = L.second;
    if (L.first < R.first && L.first > R.second) second = L.first;
    if (L.first < R.first && L.first < R.second) second = R.second;
```

Example 2: Given an array, find the maximum subarray sum using divide and conquer.

```
struct result {
    int sum, prefix, suffix, total;
};
result solve(vector<int>& A, int l, int r) {
    if (l == r) {
        return {A[l], A[l], A[l], A[l]};
    }
    int mid = l + (r - l) / 2;
    auto L = solve(A, l, mid);
    auto R = solve(A, mid + 1, r);
    int sum = L.sum + R.sum;
    int prefix = max(L.prefix, L.sum + R.prefix);
    int suffix = max(R.suffix, R.sum + L.suffix);
    int total = max({L.total, R.total, L.suffix + R.prefix});
    return {sum, prefix, suffix, total};
}
```

Dynamic Programming

Binary Choice

Notation. $OPT(j)$ is the optimal solution for the subproblem j.

- Case 1: OPT selects j.
  - can't select incompatible jobs $p(j)+1, p(j)+2, \ldots, j-1$
  - must include optimal solution for $1, 2, \ldots, p(j)$
- Case 2: OPT does not select j.
  - must include optimal solution for $1, 2, \ldots, j-1$

$$OPT(j) = \max\{OPT(p(j)) + v_j, OPT(j-1)\}$$

Weighted Interval Scheduling

Given a set of intervals, each with a start time s_i, end time f_i, and value v_i, find the maximum total value of non-overlapping intervals.

Pseudocode:

```
def WIS(intervals):
    sort intervals by end time
    compute p[j] for 1 <= j <= n
    M[0] = 0
    for j = 1 to n:
        M[j] = max(v[j] + M[p[j]], M[j-1])
        if M[j] == M[j-1]:
            B[j] = 0
```

```
        else:
            B[j] = 1
    m = n # backtracking
    while m > 0:
        if B[m] == 1:
            print m
            m = p[m]
        else:
            m = m - 1
    return M[n]
```

$p(n)$ can be computed in $O(n)$ total time using two pointers.

Time complexity: $O(n \log n)$

Implementation:

```
void print_solution(vector<pair<int, int>>& intervals, vector<int>& p,
vector<int>& B, int m) {
    if (m == 0) return;
    if (B[m] == 1) {
        print_solution(intervals, p, B, p[m]);
        cout << m << " ";
    } else {
        print_solution(intervals, p, B, m - 1);
    }
}

int weighted_interval_scheduling(vector<pair<int, int>>& intervals) {
    sort(intervals.begin(), intervals.end(), [](const auto& a, const auto& b) {
        return a.second < b.second;
    });
    int n = intervals.size();
    vector<int> p(n + 1);
    for (int j = 1; j <= n; j++) {
        int i = j - 1;
        while (i >= 0 && intervals[i].second > intervals[j - 1].first) {
            i--;
        }
        p[j] = i + 1;
    }
    vector<int> M(n + 1), B(n + 1);
    M[0] = 0;
    for (int j = 1; j <= n; j++) {
        M[j] = max(intervals[j - 1].second + M[p[j]], M[j - 1]);
        B[j] = M[j] == M[j - 1] ? 0 : 1;
    }
    print_solution(intervals, p, B, n);
    return M[n];
}
```

Manhattan Tourist Problem

Given a grid with weights on edges, find the maximum-weight path from $(0, 0)$ to (n, m) that only moves right or down.

$$s_{i,j} = \max\{s_{i-1,j} + d(\{i-1, j\}, \{i, j\}), s_{i,j-1} + d(\{i, j-1\}, \{i, j\})\}$$

Time complexity: $O(nm)$

Implementation with backtracking:

```
void print_solution(vector<vector<int>>& s, vector<vector<int>>& d, int i, int
j) {
    if (i == 0 && j == 0) return;
    if (i > 0 && s[i][j] == s[i - 1][j] + d[i - 1][j]) {
        print_solution(s, d, i - 1, j);
        cout << "↓";
    } else {
        print_solution(s, d, i, j - 1);
        cout << "→";
    }
}

int manhattan_tourist(int n, int m, vector<vector<int>>& d) {
    vector<vector<int>> s(n + 1, vector<int>(m + 1));
    for (int i = 1; i <= n; i++) {
        s[i][0] = s[i - 1][0] + d[i - 1][0];
    }
    for (int j = 1; j <= m; j++) {
        s[0][j] = s[0][j - 1] + d[0][j - 1];
    }
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            s[i][j] = max(s[i - 1][j] + d[i - 1][j], s[i][j - 1] + d[i][j - 1]);
        }
    }
    print_solution(s, d, n, m);
    return s[n][m];
}
```

0-1 Knapsack Problem

Given a set of items, each with a weight w_i and a value v_i, and a knapsack with a maximum weight capacity W, find the maximum total value that can be put into the knapsack.

Denote $OPT(i, w)$ as the optimal solution for the subproblem i with weight w.

- Case 1: OPT does not select item i.
  - $OPT(i, w) = OPT(i-1, w)$
- Case 2: OPT selects item i.
  - Inherited from the optimal solution for $i - 1$ with weight at most $w - w_i$.

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{OPT(i-1, w), v_i + OPT(i-1, w-w_i)\} & \text{otherwise} \end{cases}$$

Time complexity: $O(nW)$

Implementation:

```
int knapsack(int W, vector<int>& w, vector<int>& v) {
    int n = w.size();
    vector<vector<int>> dp(n + 1, vector<int>(W + 1));
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= W; j++) {
            if (w[i - 1] > j) {
                dp[i][j] = dp[i - 1][j];
            } else {
                dp[i][j] = max(dp[i - 1][j], v[i - 1] + dp[i - 1][j - w[i -
1]]); // v[] is 0-indexed
            }
        }
    }
    return dp[n][W];
}
```

How to find the items selected?

```
void print_solution(vector<vector<int>>& dp, vector<int>& w, vector<int>& v, int
i, int w) {
    if (i == 0 || w == 0) return;
    if (dp[i][w] == dp[i - 1][w]) {
        print_solution(dp, w, v, i - 1, w);
    } else {
        print_solution(dp, w, v, i - 1, w - w[i - 1]);
        cout << i << " ";
    }
}
```

Longest Palindromic Subsequence

A **palindromic subsequence** is a subsequence that reads the same forwards and backwards.

Given a string S, find the length of the longest palindromic subsequence.

Denote $d(i, j)$ as the length of the longest palindromic subsequence of $S[i, j]$.

$$d(i,j) = \begin{cases} 1 & \text{if } i = j \\ 2 & \text{if } i = j - 1 \text{ and } S[i] = S[j] \\ 1 & \text{if } i = j - 1 \text{ and } S[i] \neq S[j] \\ d(i+1, j-1) + 2 & \text{if } S[i] = S[j] \\ \max\{d(i+1, j), d(i, j-1)\} & \text{otherwise} \end{cases}$$

Note the order of iteration.

Time complexity: $O(n^2)$

Pseudocode with backtracking:

```
def LPS(A, n):
    for i = 1 to n:
        d[i][i] = 1
```

```
        d[i][i+1] = 2 if A[i] == A[i+1] else 1

    for j = 2 to n-1:
        for i = n-j to 1:
            k = j + i
            d[i][k] = d[i+1][k-1] + 2 if A[i] == A[k] else max(d[i+1][k], d[i]
[k-1])
            if d[i][k] == d[i+1][k-1] + 2:
                b[i][k] = 0
            if d[i][k] == d[i][k-1]: # tail is removed
                b[i][k] = 2
            if d[i][k] == d[i+1][k]: # head is removed
                b[i][k] = 3

    p = 1, q = n
    l1 = [], l2 = []
    while p <= q:
        if b[p][q] == 0:
            p += 1
            q -= 1
            l1.push_back(A[p])
            l2.push_head(A[q])
        if b[p][q] == 2:
            q -= 1
        if b[p][q] == 3:
            p += 1
    return d[1][n], l1 + l2
```

Longest palindromic subsequence can also be solved using LCS. The answer is the LCS of S and S^R (i.e. the reverse of S).

Other Problems

Longest Common Subsequence: Given two strings A and B, find the length of the longest common subsequence.

Solution: Denote $d(i,j)$ as the length of the longest common subsequence of $A[1,i]$ and $B[1,j]$.

$d(i,j) = d(i-1,j-1)+1$ if $A[i] = B[j]$, otherwise $d(i,j) = \max\{d(i-1,j), d(i,j-1)\}$.

Backtracking: if $A[i] = B[j]$, then $A[i]$ and $B[j]$ are in the LCS, and goto $(i-1,j-1)$. Otherwise, if $d(i,j) = d(i-1,j)$, goto $(i-1,j)$, otherwise goto $(i,j-1)$.

Edit Distance: Given two strings A and B, find the minimum number of operations to convert A to B.

Solution: Find the LCS of A and B, then the edit distance is $|A|+|B|-2\cdot|LCS|$.

Longest Increasing Subsequence: Given an array A, find the length of the longest increasing subsequence.

Solution: Denote $d(i)$ as the length of the longest increasing subsequence ending at $A[i]$.

$d(i) = \max\{d(j)+1 : j < i \text{ and } A[j] < A[i]\}$

Backtracking: store the predecessor of each element. if $d(i) = d(p[i])+1$, then i is in the LIS, and goto $p[i]$. Otherwise, goto $i-1$.

Coin Change: Given a set of coin denominations (infinite supply) and a target amount, find the minimum number of coins needed to make up the amount.

Solution: Denote $d(i)$ as the minimum number of coins needed to make up the amount i.

$d(i) = \min\{d(i-c)+1 : c \text{ is a coin denomination}\}$

Time complexity: $O(n \cdot |C|)$, where n is the target amount and C is the set of coin denominations.

Subset Sum: Given a set of integers and a target sum, find if there is a subset that sums to the target.

Solution: Denote $d(i,j)$ as true if there is a subset of $A[1,i]$ that sums to j.

$d(i,j) = d(i-1,j) \text{ or } d(i-1, j-A[i])$

Note that the space complexity can be reduced to $O(n)$.

Let $G=(V,E)$ be a undirected chain, where $V=\{v_1,v_2,\ldots,v_n\}$ contains n nodes and $E=\{(v_i,v_{i+1})|i=1,2,\ldots,n-1\}$. Distance between v_i and v_{i+1} is $d(v_i,v_{i+1}) \geq 0$. Each node v_i has a weight $w(v_i) \geq 0$.

An independent set $V' \subseteq V$ is a subset of V such that for any pair of nodes $v_i \in V'$ and $v_j \in V'$, $(v_i,v_j) \notin E$. i.e. no two nodes in V' are adjacent. The weight of the independent set $w(V')$ is the total weight of the nodes in V'.

Design a DP algorithm that finds the maximum weight independent set V' in G such that for any pair of nodes v_i and v_j in V', $d(v_i,v_j) \geq L$, where L is input.

Analysis The goal is to find a subset of nodes such that

- No adjacent nodes are selected.
- The distance between any two consecutively selected nodes is at least L.
- The total weight of the selected nodes is maximized.

Denote $dp(i)$ as the maximum weight independent set using first i nodes.

Denote $p(i)$ as the last node before v_i such that $i-p(i) \geq 2$ and $d(v_{p(i)},v_i) \geq L$.

$$dp(i) = \begin{cases} 0 & \text{if } i=0 \\ w(v_1) & \text{if } i=1 \\ \max\{dp(i-1), w(v_i)+dp(p(i))\} & \text{otherwise} \end{cases}$$

```
maxWeightIndependentSet(G, L)
    INPUT: G=(V, E), L
    OUTPUT: V'=max weight independent set, w(V')
    dp[0..n] = 0
    p[0..n] = 0
    dp[1] = w(v[1])

    for i = 2 to n:

        # find p[i]
        dist = d(v[i-1], v[i])
        for j = i-2 to 1:
            dist += d(v[j], v[j+1])
            if dist >= L:
```

```
            p[i] = j
            break

    dp[i] = max{dp[i-1], w(v[i]) + dp[p[i]]}


# backtracking
V' = []
w = dp[n]
while n > 0:
    if dp[n] == dp[n-1]:
        n -= 1
    else:
        V'.push(v[n])
        n = p[n]

return V', w
```