CS4335 Design and Analysis of Algorithms - Review

Algorithms: Any well-defined **computational procedure** that takes some value, or set of values, as **input** and produces some value, or set of values, as **output**.

Graph Theory

- A **graph** $G = (V, E)$ consists of a set of **vertices** V and a set of **edges** E.
- A **path** P in a graph is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$.
- A **circuit** is a path that starts and ends at the same vertex.
- **Degree** of a vertex v is the number of edges incident to v.

Representation of Graphs (Lecture 1)

Adjacency Matrix

Adjacency Matrix is a $|V| \times |V|$ matrix A where $A_{ij} = 1$ if $(v_i, v_j) \in E$ and 0 otherwise.

- Space complexity: $O(N^2)$
- Checking if $(v_i, v_j) \in E$: $O(1)$
- Finding all neighbors of a vertex: $O(N)$

Adjacency List

Adjacency List is a list of lists where each vertex v_i has a list of vertices that are adjacent to v_i.

- Space complexity: $O(N + M)$
- Checking if $(v_i, v_j) \in E$: $O(\deg(v_i))$
- Finding all neighbors of a vertex: $O(\deg(v_i))$

Euler and Hamiltonian Circuits (Lecture 1)

Euler Circuit

Given a graph $G = (V, E)$, an **Euler circuit** is a circuit that traverses every **edge** of G exactly once.

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

Pseudocode: (All code in this course is given in pseudocode that is similar to Python, but does not consider syntax, border cases, time complexity, etc.)

```
def findEulerCircuit(G):
    input: G = (V, E)
    output: E* = the Euler circuit

    E* = []
    while V is not empty: # if a node has all edges visited, remove it
        u = any vertex in V
        cur_circuit = []
        while deg(u) > 0:
            v = any neighbor of u
            E.remove((u, v))
            cur_circuit.append((u, v))
            u = v
        V.remove(u)
        merge cur_circuit into E*
    return E*
```

Challenge problem: Design an algorithm to add minimum number of edges to any graph to make it contain an Euler circuit, and prove its correctness. (The number of edges can be zero.)

Answer: By definition, a connected undirected graph has an Euler circuit if and only if every vertex has even degree. Therefore, we need to do pair up vertices with odd degree and add an edge between them.

It can be proved that the number of vertices with odd degree is always even, since every edges contributes to the degree of two vertices, and thus the total degree is always even.

```
def addEulerEdges(G):
    input: G = (V, E)
    output: E*, m

    odd_vertices = [v for v in V if deg(v) % 2 == 1]
    m = len(odd_vertices) // 2
    E* = E.copy()
    for i in range(0, len(odd_vertices), 2):
        E*.append((odd_vertices[i], odd_vertices[i+1]))
    return E*, m
```

Hamiltonian Circuit

Given a graph $G = (V, E)$, a **Hamiltonian circuit** is a circuit that traverses every **vertex** of G exactly once.

Finding a Hamiltonian circuit is an **NP-complete** problem, which means that there is no known polynomial-time algorithm to solve it.

Minimum Spanning Trees (Lecture 3)

- A subgraph T of a graph G is a **spanning tree** if T is a tree and $V(T) = V(G)$, i.e. T contains all vertices of G.
- A **minimum spanning tree** (MST) of a graph G is a spanning tree with the smallest possible sum of edge lengths.

Generic algorithm for finding an MST:

```
def GenericMST(G):
    input: G = (V, E)
    output: T = the minimum spanning tree

    T = []
    while T does not form a spanning tree:
        find an edge (u, v) that is safe to add to T
        T.append((u, v))
    return T
```

- Set A is always a subset of some MST. This property is called the **invariant property**, i.e. the total weight of the edges in A is always locally optimal.
- An edge (u, v) is **safe** to add to A if $A \cup \{(u, v)\}$ is a subset of some MST.
- A **cut** $(S, V - S)$ of a graph G is a partition of V into two sets S and $V - S$.
- An edge **crosses** the cut if one of its endpoints is in S and the other is in $V - S$.
- An **light edge crossing** the cut is the shortest edge that crosses the cut.

Theorem: If A is a subset of some MST of G, and $(S, V - S)$ is any cut that respects A (i.e. all edges in A are on the same side of the cut), then the light edge (u, v) crossing the cut is safe to add to A.

Proof: Let T be an optimal MST that contains A.

- If $(u, v) \in T$, then (u, v) is safe to add to A.
- If $(u, v) \notin T$, there must be a path P in T from u to v, and there must be another crossing edge (u', v') on the path P.
  The path can be divided into 3 parts: $u \to u'$, (u', v'), $v' \to v$.
  If we add (u, v) to T, this would create a cycle, and we can remove (u', v') from T to get a new MST T'.
  Since (u, v) is the lightest edge crossing the cut, $w(u, v) \leq w(u', v')$, and thus T' is also an optimal MST.

Therefore, the light edge crossing the cut is safe to add to A.

Kruskal's Algorithm

In Kruskal's algorithm, A is a forest of trees, and (u, v) is the lightest edge that connects two trees in A.

```
def kruskalMST(G):
    input: G = (V, E) # E = [(u, v, w), ...]
    output: T = the minimum spanning tree

    T = []
    for v in V:
        makeSet(v) # initially every vertex is a tree
    E.sort(key=lambda x: w) # by weight
    for (u, v, w) in E:
        if findSet(u) != findSet(v):
            T.append((u, v))
            union(u, v)
    return T
```

In implementation, Disjoint Set Union (DSU) is used to maintain the forest of trees.

Complexity: $O(M \log M)$ since sorting edges dominates the time complexity.

DSU is a collection of sets, each of which is a tree.

It can be initialized with `makeSet(v)` to create a set containing a single element v.

It supports two operations:

- `findSet(x)`: returns the representative of the set containing x.
- `union(x, y)`: merges the sets containing x and y.

Each taking $O(\log N)$ time. (Can be optimized to $O(\alpha(N)) \approx O(1)$ with path compression and union by rank.)

Prim's Algorithm

In Prim's algorithm, A is a single tree, and (u, v) is the lightest edge that connects a vertex in A to a vertex outside A.

```
def primMST(G):
    input: G = (V, E) # E = [(u, v, w), ...]
    output: T = the minimum spanning tree

    T = []
    u = any vertex in V
    S = {u} # vertices in the tree
    Q = V - {u} # vertices outside the tree
    while Q is not empty:
        (u, v, w) = shortest edge where u in S and v in Q
        T.append((u, v))
        S.add(v)
        Q.remove(v)
    return T
```

In implementation, priority queue is used to find the shortest edge in $O(\log N)$ time.

Complexity: $O((N + M) \log N)$ with priority queue. (Can be optimized to $O(M + N \log N)$ with Fibonacci heap.)

A priority queue is a data structure (implemented as a heap or balanced BST), where each item x has a key $k(x)$, and supports three operations:

- `insert(x, k)`: inserts x with key k.
- `update(x, k)`: updates the key of x to k.
- `extractMin()`: removes and returns the item with the smallest key.

All operations take $O(\log N)$ time. (In Fibonacci heap, `update` takes $O(1)$ time.)

Challenge problem 1: Does Kruskal's and Prim's algorithm work for maximum spanning tree?

Answer: Yes, they work. The invariant property that A is a subset of some maximum spanning tree still holds, the only change is to find the heaviest edge crossing the cut.

Challenge problem 2: Prove that for a graph with distinct edge weights, the minimum spanning tree is unique.

Answer: Proved by contradiction. Assume there are two different MSTs T_1 and T_2. Since they are different, there must be an edge e_1 in T_1 that is not in T_2.

Since MST is a tree, adding e_1 to T_2 will create a cycle. By removing the heaviest edge $e_2 \neq e_1$ in the cycle, we get a new MST T_2'.

The total weight of T_2' is $w(T_2) - w(e_2) + w(e_1)$. Since all edge weights are distinct, and e_2 is the heaviest edge in the cycle containing e_1, $w(e_2) > w(e_1)$, and thus $w(T_2') < w(T_2)$.

So we obtained a new MST T_2' with a smaller weight than T_2, which contradicts the assumption that T_2 is an MST. Therefore, the assumption that there are two different MSTs is false, and the MST is unique.

Single-Source Shortest Paths

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

Negative-weight cycle: A cycle in a graph where the sum of the weights of the edges is negative. If there is a negative-weight cycle reachable from the source vertex, the shortest path is not well-defined.

Theorem: A subpath of a shortest path is also a shortest path.

- Denote $\pi(v)$ as the predecessor of v in the shortest path from s to v.
- Denote $d(v)$ as the shortest path distance from s to v.

Observation. Suppose a shortest path from s to v can be decomposed to $s \to u \to v$. Then $d(v) = d(u) + w(u, v)$.

Relaxation: Given an edge (u, v), if $d(v) > d(u) + w(u, v)$, then update $d(v) = d(u) + w(u, v)$ and $\pi(v) = u$.

```
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 (Lecture 4)

Maintains a set S of vertices whose shortest path distance from s is known.

At each step, greedily selects the vertex $u \in V - S$ with the smallest $d(u)$ and relaxes all edges (u, v).

```
def dijkstra(G, s):
    input: G = (V, E) # E = [(u, v, w), ...]
    output: d, pi

    d = {v: inf for v in V}
    pi = {v: None for v in V}
    d[s] = 0
    S = set()
    Q = set(V) # a priority queue maintaining d[v] for v in (V-S)
    while Q is not empty:
        u = extractMin(Q)
        S.add(u)
        for v in neighbors(u):
            if d[v] > d[u] + w(u, v):
                d[v] = d[u] + w(u, v)
                pi[v] = u
                Q.update(v, d[v])
    return d, pi
```

To retrieve the shortest path from s to v, we can trace back from v to s using π.

Theorem: Consider the set S at any point in Dijkstra's algorithm. For any vertex $v \in S$, $d(v)$ is the shortest path distance from s to v, i.e. nodes in S are finalized.

Proof: By natural induction. Initially, $|S| = 1$ holds because $d(s) = 0$. Assume that $|S| = k > 0$ holds, now we need to prove that $|S| = k + 1$ holds.

Let node $v \in V - S$ be the node with the smallest $d(v)$, and P be a path from s to v.

- Case 1: All the nodes in P before v are in S. Since $d(v) = \min_{u \in S} d(u) + w(u, v)$, the length of $P = d(x) + w(x, v) \geq d(v)$ (where x is the last node in P before v).
- Case 2: $P = s \to \ldots \to x \to y \to \ldots \to v$, where $x \in S$ and $y \in V - S$. Since our algorithm selects v as the node with the smallest $d(v)$ and all edges have weight ≥ 0, we have $w(P) \geq d(y) \geq d(v)$.

Therefore $d(v)$ is the shortest path distance from s to v, and it can be added to S.

Therefore, the theorem holds by induction.

Time complexity: $O(N^2)$ with a simple array, $O(N \log N + M)$ with a priority queue.

Bellman-Ford Algorithm (Lecture 8)

Bellman-Ford is a DP-based algorithm that can handle graphs with negative-weight edges, and report the presence of negative-weight cycles.

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

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

```
def bellmanFord(G, s):
    input: G = (V, E) # E = [(u, v, w), ...]
    output: d, pi

    d = {v: inf for v in V}
    pi = {v: None for v in V}
    d[s] = 0
    for i in range(1, len(V)): # at most |V| - 1 rounds
        for (u, v, w) in E:
            if d[v] > d[u] + w:
                d[v] = d[u] + w
                pi[v] = u
    for (u, v, w) in E: # the |V|-th round
        if d[v] > d[u] + w:
            return "Negative-weight cycle detected"
    return d, pi
```

Time complexity: $O(NM)$. We can terminate early if no updates to d are made in a round.

Corollary: If negative-weight circuit exists in the graph, in every iteration, the shortest path from s to some vertex v will be reduced.

Maximum Flow (Lecture 9)

A **flow network** is a directed graph $G = (V, E)$ with two special vertices s (source) and t (sink), and each edge (u, v) has a capacity $c(u, v) \geq 0$.

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

A **flow** is a function $f : V \times V \to \mathbb{R}$ that satisfies the following properties:

- **Capacity constraint**: $0 \leq f(u, v) \leq c(u, v)$ for all $u, v \in V$.
- **Flow conservation**: For all $u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$.

Net flow $f(u, v)$ is the flow from u to v. The **value** of the flow is $|f| = \sum_{v \in V} f(s, v)$.

Ford-Fulkerson Algorithm

Residual network G_f is a graph that represents the remaining capacity of the edges in the flow network.

Residual capacity is the remaining capacity of an edge in the residual network. $c_f(u, v) = c(u, v) - f(u, v)$, i.e. $c_f(u, v) = c(u, v) + f(v, u)$.

Augment path is a path from s to t in the residual network G_f, should all edges have positive residual capacity.

The residual capacity of an augment path is the smallest residual capacity of the edges in the path. $c_f(P) = \min_{(u,v) \in P} c_f(u, v)$.

```
def fordFulkerson(G, s, t):
    input: G = (V, E) # E = [(u, v, c), ...]
    output: f, |f| = f(s, t)

    f = {u: {v: 0 for v in V} for u in V}
    |f| = 0
    while there is an augment path P in G_f:
        c_f(P) = min(c_f(u, v) for (u, v) in P)
        |f| += c_f(P)
        for (u, v) in P:
            f[u][v] += c_f(P)
            f[v][u] -= c_f(P)
    return f, |f|
```

Time complexity: $O(|E| \cdot f^*)$ where f^* is the maximum flow value, assuming all capacities are integers.

The basic Ford-Fulkerson algorithm does not terminate if the capacities are real numbers, and thus the **Edmonds-Karp algorithm** is used, which always selects the shortest augment path.

EK algorithm uses BFS to find the shortest augment path ("shortest" in terms of the number of edges), and thus the time complexity is $O(VE^2)$.

Maximum Bipartite Matching

A **bipartite graph** is a graph $G = (V, E)$ where $V = X \cup Y$ and $X \cap Y = \emptyset$, and all edges are between X and Y.

A **matching** is a subset of edges $M \subseteq E$ such that no two edges share a common vertex. i.e. For each vertex $v \in V$, there is at most one edge in M incident to v.

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

It can be transformed to a maximum flow problem by adding a source vertex s connected to all vertices in X and a sink vertex t connected to all vertices in Y. The capacity of all edges is 1.

The number of edges in the maximum matching is equal to the value of the maximum flow.

More on Graphs

Midterm

Given a MST T of a graph $G = (V, E)$, and a new edge (u, v) with weight w, design an algorithm to find the new MST T'.

Solution: Adding (u, v) to T creates a cycle. The heaviest edge in the cycle can be removed to get a new MST T'.

Proof of correctness:

1. An optimal T_{opt} does not contain (u, v). T is an optimal MST for the graph $G + (u, v)$, and the total weight of T is equal to the total weight of T_{opt}.
2. An optimal T_{opt} contains (u, v). Denote $T' = T_{opt} + (u, v)$. Deleting an edge in the cycle (that is not (u, v)) will form a cut, and in T there is an crossing edge connecting the two sides of the cut. Since T is optimal for G, if the deleted egde is not (u, v), the cost of the crossing edge in T is not larger than the deleted edge. Deleting (u, v) from T' form a cut, and our algorithm selects the lightest crossing edge for such a cut, and thus the total weight of T' is not larger than T_{opt}.

Greedy

Greedy algorithms make a series of choices that are **locally optimal** at each step, with the hope of finding a global optimum. This is not always the case, and thus proving the correctness of a greedy algorithm is important.

Interval Scheduling & Interval Partitioning (Lecture 2)

Interval Scheduling

Given a set of intervals $I = \{(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)\}$, find the maximum number of non-overlapping intervals.

Greedy algorithm: Sort the intervals by their finish times and select the first interval that does not overlap with the previous one.

Proof of correctness: Proved by exchange argument (gradually transform any solution to the greedy solution without hurting its quality).

Denote $G = \{i_1, i_2, \ldots, i_k\}$ as the set of intervals selected by the greedy algorithm and $O = \{j_1, j_2, \ldots, j_m\}$ as the set of optimal intervals.

Assume $i_1 = j_1, i_2 = j_2, \dots, i_k = j_k$ and $i_{k+1} \neq j_{k+1}$.

According to the greedy algorithm, i_{k+1} is the earliest-finishing interval that does not overlap with i_k. ($f(i_{k+1}) \leq f(j_{k+1})$)Therefore, replacing j_{k+1} with i_{k+1} will not overlap with $j_{k+2} \dots j_m$, and thus does not decrease the number of selected intervals.

Therefore, by repeating the process which replaces j_o with i_o for $o = k+1, k+2, \dots, m$, we can make the optimal solution equal to the greedy solution, without decreasing the number of selected intervals.

Therefore, $n \geq m$, and thus the greedy algorithm is optimal.

```
def intervalScheduling(I):
    input: I = [(s1, f1), (s2, f2), ..., (sn, fn)]
    output: G = the set of selected intervals

    I.sort(key=lambda x: f) # by finish time
    G = []
    prev = None
    for i in I:
        if prev is None or i.s >= prev.f: # *
            G.append(i)
            prev = i
    return G
```

(*) Since intervals are sorted by finish time, the last chosen interval will always have the latest finish time, which is the current interval i needs to check against.

Interval Partitioning

Given a set of intervals $I = \{(s_1, f_1), (s_2, f_2), \dots, (s_n, f_n)\}$, find the minimum number of classrooms needed to hold all intervals.

Greedy algorithm: Sort the intervals by their start times and assign each interval to the first classroom that is available.

Proof of correctness: Proved by structural induction (observe a property of the optimal solution and prove that the greedy solution also has this property).

Observed: greedy algorithm never schedules two overlapping intervals in the same classroom, i.e. the solution is always valid.

Let d be the number of classrooms used by the greedy algorithm.

Classroom d is allocated because we need to schedule an interval (s_i, f_i) that overlaps with all other intervals in the previous $d-1$ classrooms.

Since all intervals are sorted by start time, the other $d-1$ classrooms are assigned before this interval can use them. Therefore, all other $d-1$ classrooms are occupied by intervals starting no later than s_i.

Therefore, we have d concurrent intervals at time s_i, which means that we need at least d classrooms to hold all intervals.

```
def intervalPartitioning(I):
    input: I = [(s1, f1), (s2, f2), ..., (sn, fn)]
    output: the setup of classrooms
    I.sort(key=lambda x: s) # by start time
    classrooms = []
    for i in I:
        assigned = False
        for c in classrooms:
            if i.s >= c[-1].f: # current interval starts after the last interval
in the classroom
                c.append(i)
                assigned = True
                break
        if not assigned:
            classrooms.append([i]) # allocate a new classroom
    return classrooms
```

Minimizing Lateness & Minimizing Waiting Time (Lecture 5)

Fractional Knapsack

Given n items with weights w_i and values v_i, and a knapsack with capacity W, find the maximum value that can be put into the knapsack.

Greedy on the value-to-weight ratio: Sort the items by v_i/w_i and put as much as possible of the item with the highest ratio.

```
def fractionalKnapsack(items, W):
    input: items = [(v1, w1), (v2, w2), ..., (vn, wn)], W
    output: dict of {item: weight}, value

    items.sort(key=lambda x: v / w, reverse=True) # by value-to-weight ratio
descending
    selected = {}
    weight = 0
    value = 0
    for (v, w) in items:
        if weight + w <= W:
            selected[(v, w)] = w
            weight += w
            value += v
        else:
            selected[(v, w)] = (W - weight) / w
            value += v * (W - weight) / w
            break
    return selected, value
```

Proof of correctness: Denote $G = \{(gv_1, gw_1), (gv_2, gw_2), \ldots, (gv_n, gw_n)\}$ as the set of items selected by the greedy algorithm and $O = \{(ov_1, ow_1), (ov_2, ow_2), \ldots, (ov_m, ow_m)\}$ as the optimal set of items.

Without loss of generality, assume $gv_1 = ov_1, gv_2 = ov_2, \ldots, gv_k = ov_k$ and $gv_{k+1} \neq ov_{k+1}$. There are two cases:

- Case 1: Optimal solution uses (ov_{k+1}, ow_{k+1}) to replace (gv_{k+1}, gw_{k+1}).
  Since the greedy algorithm selects items by the value-to-weight ratio, $\frac{gv_i}{gw_i} \geq \frac{gv_{k+1}}{gw_{k+1}}$ holds for all $i \leq k$.

  Therefore, all items with higher value-to-weight ratio are selected before gv_{k+1}.
  Therefore, ov_{k+1} must have a value-to-weight ratio that is less than or equal to gv_{k+1}, and replacing ov_{k+1} with gv_{k+1} will not decrease the total value.
- Case 2: Optimal solution uses the same item but less weight $ow_{k+1} < gw_{k+1}$.
  Therefore total weight of $ov_{k+2} \dots ov_m$ has increased by that difference $d = gw_{k+1} - ow_{k+1}$, compared to that of $gv_{k+2} \dots gv_n$.
  However, since the greedy algorithm selects items by the value-to-weight ratio, we have $\frac{gv_{k+1}}{gw_{k+1}} \geq \frac{gv_{k+2}}{gw_{k+2}}, \dots, \frac{gv_n}{gw_n}$ as well as $\frac{ov_{k+1}}{ow_{k+1}} \geq \frac{ov_{k+2}}{ow_{k+2}}, \dots, \frac{ov_m}{ow_m}$.
  Therefore allocating d weight to item $k+1$ instead of $k+2, \dots, m$ will not decrease the total value.

Therefore, by replacing the items in the optimal solution with (gv_i, gw_i) for $i = k+1, k+2, \dots, n$, the total value will not decrease. By repeating this process, we can make the optimal solution equal to the greedy solution, without decreasing the total value.

Therefore, the greedy algorithm is optimal.

Minimizing Lateness

Given n jobs with processing times p_i and deadlines d_i, find the schedule that minimizes the maximum lateness. The lateness of job i is $L_i = \max(0, f_i - d_i)$, where f_i is the finish time of job i.

Greedy on the deadline: Sort the jobs by their deadlines and schedule them in the order of the deadline. In case of a tie, schedule the job with the shortest processing time first.

```
def minimizeLateness(jobs):
    input: jobs = [(p1, d1), (p2, d2), ..., (pn, dn)]
    output: schedule, max_lateness

    jobs.sort(key=lambda x: (d, p)) # by deadline, then processing time
    schedule = []
    finish = 0
    max_lateness = 0
    for (p, d) in jobs:
        finish += p
        lateness = max(0, finish - d)
        max_lateness = max(max_lateness, lateness)
        schedule.append((p, d, finish, lateness))
    return schedule, max_lateness
```

Proof of correctness: Structural induction + exchange argument.

An **inversion** in schedule S is a pair of jobs i and j such that i is scheduled after j but $d_i < d_j$.

It can be observed that if a schedule has inversions, it must contain adjacent inversions, i.e. i is scheduled after $j = i - 1$ but $d_i < d_j$. (Proof: swapping an adjacent inversion pair eliminates the inversion. If there are no adjacent inversions, it is not possible to eliminate inversions by swapping, and thus there are no inversions.)

By swapping two adjacent inversion jobs i and j, the number of inversions is reduced by 1 and the maximum lateness is not increased. Proof:

- Let l be the lateness before swapping, and l' be the lateness after swapping.
- $l_k = l'_k$ for all jobs $k \neq i, j$.

- $l'_i < l_i$ since i is now scheduled earlier and its finish time is advanced, $l'_i = \max(0, f'_i - d_i) < \max(0, f_i - d_i) = l_i$.
- Therefore if the swap increases the maximum lateness, it must be due to $l'_j > l_i$. Now let's proove $l'_j \leq l_i$: $l'_j = \max(0, f'_j - d_j) = \max(0, f_i - d_j) \leq \max(0, f_i - d_i) = l_i$

Therefore, by swapping adjacent inversion pairs, the maximum lateness is not increased.

Back to the problem: Denote G as the greedy schedule and O as the optimal schedule. By the algorithm, G does not contain any inversions.

If there are adjacent inversions in O, we can swap them to get a new schedule O' with fewer inversions and no greater maximum lateness. We can repeat the step until O has no inversions, and thus O is equal to G.

Therefore, the greedy algorithm is optimal.

Minimizing Waiting Time

Given n jobs with processing times p_i, find the schedule that minimizes the total waiting time. The waiting time of job i is $W_i = \sum_{j=1}^{i-1} p_j$.

Greedy on the processing time: Sort the jobs by their processing times and schedule them in the order of the processing time.

```
def minimizeWaitingTime(jobs):
    input: jobs = [p1, p2, ..., pn]
    output: schedule, waiting

    jobs.sort() # by processing time
    schedule = []
    waiting = 0
    for p in jobs:
        waiting += p
        schedule.append((p, waiting))
    return schedule, waiting
```

Proof of correctness (core only): An inversion in schedule S is a pair of jobs i and j such that i is scheduled after j but $p_i < p_j$.

G contains no inversions. If there are adjacent inversions p_j, p_i in O, we can swap them to get a new schedule O' with fewer inversions and no greater total waiting time. Since $W_j + W_i = 2(p_1 + p_2 + \ldots + p_{j-1}) + p_j$, $W'_i + W'_j = 2(p_1 + p_2 + \ldots + p_{j-1}) + p_i$, and $p_i < p_j$, we have $W'_i + W'_j \leq W_j + W_i$. Since all other jobs are not affected, the total waiting time is not increased.

By repeating the process, we can make O equal to G, and thus the greedy algorithm is optimal.

Divide and Conquer

At each level of recursion, three steps are performed:

- **Divide**: The problem is divided into a number of subproblems that are smaller instances of the same problem.
- **Recur**: The subproblems are solved recursively.
- **Conquer**: The solutions to the subproblems are combined to solve the original problem.

Merge Sort (Lecture 6)

```
def mergeSort(A):
    input: A = [a1, a2, ..., an]
    output: sorted A

    if len(A) <= 1:
        return A
    mid = len(A) // 2
    L = mergeSort(A[:mid])
    R = mergeSort(A[mid:])
    return merge(L, R)

def merge(L, R):
    output: merged L and R

    i = j = 0
    merged = []
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            merged.append(L[i])
            i += 1
        else:
            merged.append(R[j])
            j += 1
    if i < len(L):
        merged.extend(L[i:])
    if j < len(R):
        merged.extend(R[j:])
    return merged
```

Counting Inversions (Lecture 6)

Based on Merge Sort, we can count the number of inversions in an array.

```
def countInversions(A):
    input: A = [a1, a2, ..., an]
    output: sorted A, number of inversions

    if len(A) <= 1:
        return A, 0
    mid = len(A) // 2
    L, invL = countInversions(A[:mid])
    R, invR = countInversions(A[mid:])
    merged, inv = mergeAndCount(L, R)
    return merged, inv + invL + invR

def mergeAndCount(L, R):
    output: merged L and R, number of inversions

    i = j = 0
    merged = []
    inv = 0
    while i < len(L) and j < len(R):
```

```python
        if L[i] <= R[j]:
            merged.append(L[i])
            i += 1
        else:
            merged.append(R[j])
            j += 1
            inv += len(L) - i # L[i...mid] > R[j], each count as an inversion
    if i < len(L):
        merged.extend(L[i:])
    if j < len(R):
        merged.extend(R[j:])
    return merged, inv
```

Time Complexity

Recurrence relation: A recursive function can be described by a recurrence relation, which describes the time complexity of the function in terms of the time complexity of its subproblems.

The recurrence relation of Merge Sort is: $T(n) = 2T(n/2) + O(n)$, or more specifically:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + O(n) & \text{o.w.} \end{cases}$$

Prove by induction that $T(n) = O(n \log n)$. Denote $n = 2^k$,

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

Another way is to use telescoping series:

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

There are $\log n$ number 1s in the series and the last term is zero.

Other examples (some use Master Theorem):

- $T(n) = T(n/2) + O(1) \Rightarrow T(n) = O(\log n)$
- $T(n) = T(n/2) + O(n) \Rightarrow T(n) = O(n)$ ($\sum_{i=0}^{\log n} n/2^i = 2n = O(n)$)
- $T(n) = 2T(n/2) + O(1) \Rightarrow T(n) = O(n)$
- $T(n) = 2T(n/2) + O(n) \Rightarrow T(n) = O(n \log n)$
- $T(n) = 2T(n/2) + O(n^2) \Rightarrow T(n) = O(n^2)$

More on Divide and Conquer

Midterm

(1) Find the largest and the second largest elements in an array in $O(n)$ time.

```
def findLargest(A):
    input: A = [a1, a2, ..., an]
    output: largest, second_largest

    if len(A) == 1:
        return A[0], None
    if len(A) == 2:
        return max(A), min(A)
    mid = len(A) // 2
    L1, L2 = findLargest(A[:mid])
    R1, R2 = findLargest(A[mid:])
    if L1 > R1:
        largest = L1
        second_largest = max(L2, R1)
    else:
        largest = R1
        second_largest = max(R2, L1)
    return largest, second_largest
```

Times of comparison: Each recursive call compares 2 times, i.e.

$$\begin{aligned} G(n) &= 2G(n/2) + C \\ &= 2(2G(n/4) + C) + C \\ &= 2^k G(n/2^k) + (1 + 2 + 4 + \ldots + 2^{k-1})C \\ &= 2^k G(1) + (2^k - 1)C \\ &= nG(1) + (n-1)C \\ &= O(n) \end{aligned}$$

(2) Given an array of distinct numbers, it is single-peaked, i.e. there exists $1 \le p \le n$ such that $A[1] < A[2] < \ldots < A[p]$ and $A[p] > A[p+1] > \ldots > A[n]$. Design an algorithm to find p in $O(\log n)$ time.

```
def findPeak(A, i, j):
    input: A = [a1, a2, ..., an], interval [i, j]
    output: p

    if i == j:
        return i
    mid = (i + j) // 2
    if A[mid] < A[mid+1]:
        return findPeak(A, mid+1, j)
    else:
        return findPeak(A, i, mid)
```

Non-recursive version:

```
def findPeak(A):
    input: A = [a1, a2, ..., an]
    output: p

    i = 0
    j = len(A) - 1
    while i < j:
        mid = (i + j) // 2
        if A[mid] < A[mid+1]:
            i = mid + 1
        else:
            j = mid
    return i
```

Time complexity: $O(\log n)$

$$\begin{aligned} G(n) &= G(n/2) + O(1) \\ &= G(n/4) + O(1) + O(1) \\ &= G(n/2^k) + kO(1) \\ &= G(1) + \log n O(1) \\ &= O(\log n) \end{aligned}$$

(3) Given an array, $A[i]$ and $A[j]$ forms a significant inversion if $i < j$ and $A[i] > 3 \cdot A[j]$. Design an algorithm to count the number of significant inversions in $O(n \log n)$ time.

First write a merge sort algorithm that counts the number of normal inversions.

```
def count(A):
    input: A = [a1, a2, ..., an]
    output: number of significant inversions, sorted A

    if len(A) <= 1:
        return 0, A

    mid = len(A) // 2
    x, L = count(A[:mid])
    y, R = count(A[mid:])
    cnt, merged = mergeCount(L, R)
    return x + y + cnt, merged

def mergeCount(A, B):
    input: A, B
    output: number of significant inversions, merged A and B

    cnt = 0
    i = j = 0
    merged = []
    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            merged.append(A[i])
            i += 1
        else:
            merged.append(B[j])
            j += 1
            cnt += len(A) - i
    if i < len(A):
        merged.extend(A[i:])
    if j < len(B):
        merged.extend(B[j:])
    return cnt, merged
```

By modifying the mergeCount function to count the number of significant inversions, we can get the result.

```
def mergeCount(A, B):
    input: A, B
    output: number of significant inversions, merged A and B

    cnt = 0
    i = j = k = 0
    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            merged.append(A[i])
            i += 1
        else: # A[i] > B[j]
            merged.append(B[j])
            j += 1
            while k < len(A) and A[k] <= 3 * B[j]:
                k += 1
            cnt += len(A) - k
    if i < len(A):
        merged.extend(A[i:])
    if j < len(B):
        merged.extend(B[j:])
    return cnt, merged
```

It is only possible for `A[i] > 3 * B[j]` to happen if `A[i] > B[j]`, so we only check this at the `else` branch.

Then we use a pointer `k` to find the first element satisfying `A[k] > 3 * B[j]`, so all of `A[k...len(A)]` forms significant inversions with `B[j]`, giving a total of `len(A) - k` significant inversions.

Time complexity: $O(n \log n)$. Merge count is $O(n)$ as it goes through all elements in A and B once. The recurrence relation is $T(n) = 2T(n/2) + O(n)$.

Dynamic Programming

Weighted Interval Scheduling (Lecture 7)

Given a set of intervals $I = \{(s_1, f_1, v_1), (s_2, f_2, v_2), \dots, (s_n, f_n, v_n)\}$, where v_i is the value of interval i, find the maximum total value of non-overlapping intervals.

Denote $P(i)$ as the largest index $j < i$ such that interval j does not overlap with interval i.

Binary choice: For each interval i, we can either include it or exclude it. This will influence all intervals after i.

Denote $OPT(i)$ as the maximum total value of non-overlapping intervals ending at interval i (either including or excluding interval i).

$$OPT(i) = \begin{cases} 0 & \text{if } i = 0 \\ \max(OPT(i-1), v_i + OPT(P(i))) & \text{o.w.} \end{cases}$$

To restore the solution, we can use a separate array B to store the choices made at each step. If $B[i] = 1$, interval i is included in the solution.

```
def weightedIntervalScheduling(I):
    input: I = [(s1, f1, v1), (s2, f2, v2), ..., (sn, fn, vn)]
    output: max value

    I.sort(key=lambda x: f) # by finish time
    P = [0] * len(I)
    for i in range(1, len(I)):
        for j in range(i-1, -1, -1):
            if I[j].f <= I[i].s:
                P[i] = j
                break
    dp = [0] * len(I)
    B = [0] * len(I)
    dp[0] = I[0].v
    for i in range(1, len(I)):
        dp[i] = max(dp[i-1], I[i].v + dp[P[i]])
        B[i] = 1 if dp[i] == I[i].v + dp[P[i]] else 0
    return dp[-1], B
```

An alternative way is to directly restore the solution by tracing back from the last interval.

```
    i = len(I) - 1
    while i >= 0:
        if dp[i] == dp[i-1]: # not chosen
            i -= 1
        else:
            print(I[i])
            i = P[i]
```

Note the $P[]$ array can be precomputed in $O(n \log n)$ time (which is sorting time), or $O(n)$ extra time.

```
    S = I sorted by start time
    F = I sorted by finish time
    s_ptr = n-1
    f_ptr = n-1
    while f_ptr >= 0:
        if S[s_ptr].start >= F[f_ptr].finish:
            f_ptr -= 1
        else:
            P[s_ptr] = f_ptr
            s_ptr -= 1
    while s_ptr >= 0:
        P[s_ptr] = -1 # predecessor does not exist
        s_ptr -= 1
```

Manhattan Tourist Problem (Lecture 7)

Given a grid G with $n+1$ rows and $m+1$ columns. Each edge has a weight, and the only allowed moves are right and down. Find the maximum weight path from $(0,0)$ to (n,m).

The binary choice is between the path going right and the path going down.

$$OPT(i,j) = \begin{cases} 0 & \text{if } i=0, j=0 \\ OPT(i,j-1) + w(i,j-1,\downarrow) & \text{if } i=0, j>0 \\ OPT(i-1,j) + w(i-1,j,\rightarrow) & \text{if } i>0, j=0 \\ \max(OPT(i,j-1) + w(i,j-1,\downarrow), OPT(i-1,j) + w(i-1,j,\rightarrow)) & \text{o.w.} \end{cases}$$

```
def MTP(G):
    input: G. since the structure too complex, we use a function w(i, j, 
down|right) to get the weight
    output: path, max weight

    n = len(G) - 1
    m = len(G[0]) - 1
    dp = [[0] * (m+1) for _ in range(n+1)]
    B = [[''] * (m+1) for _ in range(n+1)]
    for i in range(1, n+1):
        dp[i][0] = dp[i-1][0] + w(i-1, 0, down)
        B[i][0] = 'down'
    for j in range(1, m+1):
        dp[0][j] = dp[0][j-1] + w(0, j-1, right)
        B[0][j] = 'right'
    for i in range(1, n+1):
        for j in range(1, m+1):
            v_down = dp[i][j-1] + w(i, j-1, down)
            v_right = dp[i-1][j] + w(i-1, j, right)
            if v_down > v_right:
                dp[i][j] = v_down
                B[i][j] = 'down'
            else:
                dp[i][j] = v_right
                B[i][j] = 'right'
    return dp[-1][-1], B
```

Backtracking:

```
    i = len(B) - 1
    j = len(B[0]) - 1
    path = []
    while i > 0 or j > 0:
        path.append((i, j))
        if B[i][j] == 'down':
            i -= 1
        else:
            j -= 1
    path.append((0, 0))
    return path[::-1] # reverse
```

0-1 Knapsack Problem (Lecture 8)

Given n items with weights w_i and values v_i, and a knapsack with capacity W, find the maximum total value that can be put into the knapsack.

We need to include a second dimension in the DP table to represent the remaining capacity of the knapsack.

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

```
def knapsack(W, items):
    input: W, items = [(v1, w1), (v2, w2), ..., (vn, wn)]
    output: max value

    n = len(items)
    items.insert(0, (0, 0)) # make 1-indexed
    dp = [[0] * (W+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(1, W+1):
            if items[i-1].w > w:
                dp[i][w] = dp[i-1][w]
            else:
                dp[i][w] = max(dp[i-1][w], items[i].v + dp[i-1][w-items[i].w])
    return dp[-1][-1]
```

Backtracking:

```
    i = n
    w = W
    selected = []
    while i > 0 and w > 0:
        if dp[i][w] == dp[i-1][w]:
            i -= 1
        else:
            selected.append(i)
            w -= items[i].w
            i -= 1
    return selected[::-1] # reverse
```

More DP Problems

Point Selection (Lecture 7)

Given a list of points $x_1, x_2, \ldots, x_n$ on a line, each having a value v_i, find the maximum total value of points such that no two selected points are less or equal to k distance apart.

Denote $p(i)$ as the largest index $j < i$ such that $x_i - x_j > k$.

```
def pointSelection(X, k):
    input: X = [(x1, v1), (x2, v2), ..., (xn, vn)], k
    output: max value

    X.sort(key=lambda x: x) # by position
    P = [0] * len(X)
    for i in range(1, len(X)):
        for j in range(i-1, -1, -1):
            if X[i].x - X[j].x > k:
                P[i] = j
                break
    dp = [0] * len(X)
    dp[0] = X[0].v
    for i in range(1, len(X)):
        dp[i] = max(dp[i-1], X[i].v + dp[P[i]])
    return dp[-1]
```

Similarly, the $P[]$ array can be precomputed in $O(n \log n)$ time.

```
    ptr = 0
    p[0] = -1
    for i in range(1, n):
        if X[i].x - X[ptr].x > k:
            while ptr < i and X[i].x - X[ptr + 1].x > k:
                ptr += 1
            P[i] = ptr
        else:
            P[i] = -1 # no predecessor
```

Coin Change (Lecture 7)

Given a set of coin denominations $c_1, c_2, \ldots, c_n$ and a target amount A, find the minimum number of coins needed to make up the amount.

Invariant property: after optimally giving the first coin, the remaining amount is also optimally solved.

$$OPT(A) = \begin{cases} 0 & \text{if } A = 0 \\ 1 + \min_{c \in \text{coins}} \{OPT(A - c)\} & \text{o.w.} \end{cases}$$

```
def coinChange(A, coins):
    input: A, coins = [c1, c2, ..., cn]
    output: min number of coins

    dp = [0] * (A+1)
    B = [0] * (A+1)
    for i in range(1, A+1):
        dp[i] = float('inf')
        for c in coins:
            if i - c >= 0:
                val = 1 + dp[i-c]
                if val < dp[i]:
                    dp[i] = val
                    B[i] = c
    return dp[-1]
```

Note it would be impossible to retrieve the solution only with the DP table, so $B[]$ is used to store the last coin used to make up the amount.

```
    amount = A
    selected = []
    while amount > 0:
        selected.append(B[amount])
        amount -= B[amount]
    return selected
```

Longest Palindromic Subsequence (Lecture 9)

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

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

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

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

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

Note the order of the DP table is important, as the value of $d(i, j)$ depends on $d(i+1, j)$ and $d(i, j-1)$. i.e. j in increasing order and i in decreasing order.

```
def longestPalindromicSubsequence(S):
    input: S
    output: the subsequence, the length

    n = len(S)
    dp = [[0] * n for _ in range(n)]
    B = [[0] * n for _ in range(n)]
    for i in range(n):
        dp[i][i] = 1
        if i < n-1:
            dp[i][i+1] = 2 if S[i] == S[i+1] else 1
    for j in range(1, n-1): # 1 to n-2 (i.e. exclude 0 and n-1)
        for i in range(j-1, -1, -1): # j-1 to 0
            if S[i] == S[i+j]:
                dp[i][i+j] = dp[i+1][i+j-1] + 2
                B[i][i+j] = 1 # include both
            else:
                dp[i][i+j] = max(dp[i+1][i+j], dp[i][i+j-1])
                B[i][i+j] = 2 if dp[i+1][i+j] > dp[i][i+j-1] else 3 # 2=skip
left, 3=skip right
    p, q = 0, n-1
    s1, s2 = "", ""
    while p <= q:
        if B[p][q] == 1:
            s1 += S[p]
            s2 += S[q]
            p += 1
            q -= 1
        elif B[p][q] == 2:
            p += 1
        else:
            q -= 1
    s = s1 + s2[::-1]
    return s, dp[0][-1]
```

Longest Increasing Subsequence

Given a sequence $A = [a_1, a_2, \ldots, a_n]$, find the length of the longest increasing subsequence.

Denote $d(i)$ as the length of the longest increasing subsequence ending at a_i (a_i must be included).

$$d(i) = \begin{cases} 1 & \text{if } i = 0 \\ \max(d(j) + 1) & \text{if } \exists j < i, a_j < a_i \\ 1 & \text{o.w.} \end{cases}$$

```
def LIS(A):
    input: A
    output: the subsequence, the length

    n = len(A)
    dp = [1] * n
    B = [0] * n
    for i in range(1, n):
        for j in range(i):
            if A[i] > A[j] and dp[j] + 1 > dp[i]:
                dp[i] = dp[j] + 1
                B[i] = j
    length = max(dp)
    idx = dp.index(length)
    s = []
    while idx >= 0:
        s.append(A[idx])
        idx = B[idx]
    return s[::-1], length
```

Longest Common Subsequence

Given two sequences $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_m]$, find the length of the longest common subsequence.

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

$$d(i, j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ d(i-1, j-1) + 1 & \text{if } A[i] = B[j] \\ \max(d(i-1, j), d(i, j-1)) & \text{o.w.} \end{cases}$$

```
def LCS(A, B):
    input: A, B
    output: the subsequence, the length

    n = len(A)
    m = len(B)
    A.insert(0, None) # make 1-indexed
    B.insert(0, None)
    dp = [[0] * (m+1) for _ in range(n+1)]
    B = [[0] * (m+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for j in range(1, m+1):
            if A[i] == B[j]:
                dp[i][j] = dp[i-1][j-1] + 1
                B[i][j] = 1 # include both
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
                B[i][j] = 2 if dp[i-1][j] > dp[i][j-1] else 3 # 2=skip left,
3=skip right

    i, j = n, m
    s = []
    while i > 0 and j > 0:
        if B[i][j] == 1:
            s.append(A[i])
            i -= 1
            j -= 1
        elif B[i][j] == 2:
            i -= 1
        else:
            j -= 1
    return s[::-1], dp[n][m]
```

Largest Sum Contiguous Subarray

Given an array $A = [a_1, a_2, \ldots, a_n]$, find the contiguous subarray with the largest sum.

Denote $d(i)$ as the largest sum contiguous subarray ending at a_i (a_i must be included).

$$d(i) = \begin{cases} a_1 & \text{if } i = 1 \\ \max(d(i-1) + a_i, a_i) & \text{o.w.} \end{cases}$$

```
def largestSumSubarray(A):
    input: A
    output: the subarray, the sum

    dp = [0] * len(A)
    dp[0] = A[0]
    for i in range(1, len(A)):
        dp[i] = max(dp[i-1] + A[i], A[i])
    max_sum = max(dp)
    idx = dp.index(max_sum)
    s = []
    while idx >= 0:
        s.append(A[idx])
        if dp[idx] == A[idx]: # if d(i) = a_i, this is the start
            break
        idx -= 1
    return s[::-1], max_sum
```

Edit Distance

Given two strings A and B, find the minimum number of operations to convert A to B. The operations include insertion, deletion, and substitution.

Solution: Let $|LCS(A, B)|$ be the length of the longest common subsequence of A and B. The edit distance is $|A| + |B| - 2 \times |LCS(A, B)|$.

Subset Sum

Given a set of integers S and a target sum T, find if there is a subset of S that sums to T.

Solution: Let $d(i, j)$ be true if there is a subset of $S[1..i]$ that sums to j. Then $d(i, j) = d(i-1, j) \vee d(i-1, j-S[i])$.

```
def subsetSum(S, T):
    input: S, T
    output: the subset, or None if not found

    n = len(S)
    dp = [[False] * (T+1) for _ in range(n+1)]
    B = [[0] * (T+1) for _ in range(n+1)]
    for i in range(n+1):
        dp[i][0] = True
    for i in range(1, n+1):
        for j in range(1, T+1):
            if j - S[i-1] >= 0:
                dp[i][j] = dp[i-1][j] or dp[i-1][j-S[i-1]]
                B[i][j] = 1 if dp[i-1][j-S[i-1]] else 2
            else:
                dp[i][j] = dp[i-1][j]
                B[i][j] = 2
    if not dp[n][T]:
        return None
    subset = []
    i, j = n, T
    while i > 0 and j > 0:
        if B[i][j] == 1:
            subset.append(S[i-1])
            j -= S[i-1]
        i -= 1
    return subset[::-1]
```