Introduction to NP-completeness, Greedy Algorithms, and Dynamic Programming
Problems Solvable in Polynomial Time

- Problems solvable in polynomial time have “polynomial-time algorithms”

- On an input of size $n$, the worst case running time of a polynomial-time algorithm is $O(n^c)$ for some constant $c$
  - $O(n^{0.0000001n})$ is not polynomial
  - $O(n^{100000000})$ is polynomial $\Rightarrow$ Occurs very rarely in practice

- Polynomial-time solvable problems belong to the complexity class $P$ $\Rightarrow$ Problems that are decidable in polynomial-time on a deterministic Turing machine

- Are all problems solvable in polynomial-time?
NP-Complete Problems

- No polynomial-time algorithm has yet been discovered

- Nobody has yet been able to prove that no polynomial-time algorithm can exist for an NP-complete problem!

- Million dollar question: Is P=NP?

- Open question since 1971, currently the most important open problem in theoretical computer science
NP-Complete Problems

Many people believe that $P \neq NP$

Latest status: In August 2010, Vinay Deolalikar claimed to have proved that $P \neq NP$ but many people are still not convinced:

http://www.hpl.hp.com/personal/Vinay_Deolalikar/

Examples:
- Shortest path problem is solvable in polynomial-time
- Longest path problem is NP-complete!
Decision Problems vs. Optimization Problems

- The answer to decision problems is a simple YES or NO.

- In optimization problems, each feasible solution has an associated value and we aim to find the best feasible solution.

- Given a directed graph $G$, vertices $u$ and $v$, costs for each edge, and a number $k$:
  - **SHORTEST PATH PROBLEM (Optimization):** Find a path from $u$ to $v$ along which the total edge cost is minimum.
  - **PATH PROBLEM (Decision):** Does there exist a path from $u$ to $v$ along which the total edge cost is at most $k$?
Decision Problems vs. Optimization Problems

- For maximization problems, the decision version: Does there exist a solution with value \( \geq k \)?

- For minimization problems, the decision version: Does there exist a solution with value \( \leq k \)?

Convenient relationship between decision problems and optimization problems

- Given the answer to the optimization problem, it is easy to answer the decision problem
- Converse is also easy: Iteratively run the decision problem by updating the number \( k \) using binary search in each iteration until you find the optimal solution

Hence, if the decision problem is easy, optimization is easy as well

If we prove that the decision problem is hard, its related optimization problem is also hard
Polynomial-time Reductions

Goal: To compare the relative difficulty of different problems

“Problem X is at least as hard as problem Y” = If we had a ‘blackbox’ capable of solving X, then we could also solve Y

Can arbitrary instances of problem Y be solved using a polynomial number of standard computational steps plus a polynomial number of calls to a blackbox that solves problem X? \(\Rightarrow\) If “YES”, then we write \(Y \leq_p X\)

We read it as:

- “Y is polynomial-time reducible to X” or
- “X is at least as hard as Y” (wrt. Polynomial-time)
Suppose $Y \leq_p X$. Then:

- If $X$ can be solved in polynomial time, then $Y$ can also be solved in polynomial time.
- If $Y$ cannot be solved in polynomial time, then $X$ cannot be solved in polynomial time.

Transitivity of polynomial-time reductions:

- If $Z \leq_p Y$ and $Y \leq_p X$, then $Z \leq_p X$

Proof: Given a blackbox for $X$, show how to solve an instance of $Z$

Run the algorithm for $Z$ using a blackbox for $Y$; but each time the blackbox for $Y$ is called, simulate it in polynomial number of steps using the algorithm that solves instances of $Y$ using a blackbox for $X$. 
Polynomial-time Reductions

- Let A be a decision problem
- Instance of the problem = The input to a particular problem
- Example: An instance for the PATH problem is a particular graph G, particular vertices u and v of G, edge weights, a particular number k
- Let B be another decision problem that we know how to solve in polynomial-time
To design a polynomial time reduction from A to B (A $\rightarrow$ B), propose a procedure $f$ that transforms every instance $\alpha$ of A to some instance $\beta$ of B with the following properties:

- The transformation $f$ takes polynomial time
- The answers to both problems are the same: The answer for $\alpha$ is YES if and only if the answer for $\beta$ is also YES (YES $\rightarrow$ YES, NO $\rightarrow$ NO)

This procedure $f$ is a polynomial time reduction and gives us a way to solve problem A in polynomial time as follows:

1. Given an instance $\alpha$ of A, use the poly-time transformation $f$ to transform it to an instance $\beta$ of B
2. Run the poly-time decision algorithm for B on the instance $\beta$.
3. Use the answer for $\beta$ as the answer for $\alpha$. 

Polynomial-time Reductions
Polynomial-time Reductions

- **Independent Set (IS) Problem:** In a graph $G=(V,E)$, we say that a set of nodes $S \subseteq V$ is independent if no two nodes in $S$ are joined by an edge. Find the largest independent set.

- **Vertex Cover (VC) Problem:** A set of nodes $S \subseteq V$ is a vertex cover if every edge $e \in E$ has at least one end in $S$ (vertices do the “covering”, edges are “covered”)

Lemma: S is an IS if and only if its complement V-S is a VC

Proof: Suppose that S is an IS. Consider an arbitrary edge e = (u,v). Since S is an IS, it cannot be the case that both u and v are in S, one of them must be in V-S. Therefore, every edge has at least one end in V-S, so V-S is a VC.

Conversely, suppose that V-S is a VC. Consider any two nodes u and v in S. If they were joined by edge e, then neither end of e would lie in V-S, contradicting our assumption that V-S is a VC. Therefore, no two nodes in S are joined by an edge, and so S is an IS.

Q.E.D.
Polynomial-time Reductions

Theorem: $\text{IS} \leq_p \text{VC}$

Proof: If we have a black box to solve VC, then we can decide whether $G$ has an IS of size at least $k$ by asking the black box whether $G$ has a VC of size at most $n-k$.

Theorem: $\text{VC} \leq_p \text{IS}$

Proof: If we have a black box to solve IS, then we can decide whether $G$ has a VC of size at most $k$ by asking the black box whether $G$ has an IS of size at least $n-k$.

Polynomial-time reductions are used extensively in NP-completeness proofs.
The complexity class NP

- NP: Non-deterministic polynomial time
  (We cannot yet say “non-polynomial-time”)

- NP is the class of languages that are decidable by some polynomial-time non-deterministic Turing machine

- NP is the class of languages that have polynomial-time “verifiers”

- Given a candidate solution, a verifier determines in poly-time whether the candidate is a member of the language or not. It “checks” a proposed solution (different from “finding”)

14
The complexity class NP

- A verifier needs an input string and a certifier

- Example: Assume somebody gives us a graph and specifies a set of vertices. A verifier for the IS problem checks if an edge joins any pair of vertices in the provided set of vertices.

- What would the verifier for the VC problem do?

- The existence of a verifier does not give us a way to solve the problem!

- Verifying whether a given solution has a YES/NO answer is different from finding the actual answer to the problem!

- Verifying is in general easier than solving!
NP-Complete Problems

- We know for sure that $P \subseteq NP$. The question is whether $P \subset NP$ or not.

- NP-Complete problems are the hardest problems in $NP$.

- A problem is $B$ is NP-complete if it satisfies two conditions:
  1. $B$ is in $NP$
  2. Every problem $A$ in $NP$ is polynomial-time reducible to $B$

- Once we have one NP-complete problem, we may obtain others by poly-time reduction from it.

- For this, we need the first NP-complete problem!
The first NP-Complete Problem: SAT

- Literal = Either a variable or the negation of a variable
  - $x_1$ = A positive literal
  - $\overline{x_2}$ = A negative literal

- Clause = A disjunction (OR) of literals
  - $x_1 \lor x_2$ = A clause

- A Boolean formula is a conjunction (AND) of clauses
  - $(x_1 \lor \overline{x_2} \lor x_3) \land (x_1 \lor x_3) \land (x_2 \lor \overline{x_3})$ = A Boolean formula

- Satisfiability (SAT) problem: Is there some assignment of TRUE and FALSE values to the variables that will make the entire Boolean formula TRUE?

Stephen Cook and Leonard Levin proved in 1971 (Cook-Levin Theorem) that SAT is NP-complete. SAT is the first problem that was proved to be NP-complete.

Given any decision problem, the proof constructs a non-deterministic Turing machine that solves it in polynomial time → Sipser’s book

Digital hardware design of modern day computers is based on AND, OR, NOT gateways and logic. It is intuitive and not surprising that we can design a Boolean formula to satisfy a Turing machine! SAT problem is the beginning of computers!
Sometimes you may encounter the optimization version of a problem being called as "NP-hard" and its decision version as "NP-complete".

NP-hard actually means a class of problems that are, informally, "at least as hard as the hardest (NP-complete) problems in NP".
NP-Completeness Proof Techniques

- Major ones are:
  - 1. Restriction
  - 2. Local Replacement
  - 3. Component Design

- Restriction:
  - A = A known NP-complete problem
  - B = The problem that we are trying to prove to be NP-complete
    - First, we need to show that B is in NP
    - Second, we need to provide a poly-time reduction A → B
  - Restriction method means to show that A is a special case of B
Restriction Method

Set Cover (SC) Problem:

- Universal Set (U) = A set of elements \{1,2,...,m\}
- n sets S whose union comprises the universe
- SC problem is to identify the smallest number of sets whose union still contains all elements in the universe U

Example: U={1,2,3,4,5} S={{1,2,3},{2,4},{3,4},{4,5}}

Clearly, the union of all the sets in S cover all the elements in U

However, we can cover all the elements with smaller number of sets as follows:
MIN_SET_COVER = {{1,2,3},{4,5}}

Practical Use: Imagine that we have n pieces of software and a set U of m capabilities that we would like our system to have. The \(i^{th}\) piece of software includes the set \(S_i \subseteq U\) of capabilities.

In the SC problem, we aim to include the smallest possible number of these pieces of software on our system, with the property that our system will have all m capabilities
Restriction Method

We already know that VC is NP-complete. To prove that SC is NP-complete:

VC → SC by restriction method:
- Edges: Elements, Universal set: All edges
- Sets = Vertices
- VC problem is a special case of the SC problem where each element (edge) occurs in at most 2 sets

In other words: If we had a poly-time algorithm to solve SC, we would be able to use this algorithm to solve the VC problem as well. However, we know that VC is NP-complete (contradiction). Hence, SC problem is NP-complete as well.

A very useful resource: Compendium of NP optimization problems
http://www.nada.kth.se/~viggo/wwwcompendium/
Another Example for Restriction Method

KNAPSACK PROBLEM: There is a set of $n$ items $1 \leq j \leq n$. Each item $j$ has an associated profit $p_j$ and weight $w_j$.
Objective: Pick some of the items with maximal total profit while obeying that the maximum total weight of the chosen items must not exceed $W$. The coefficients are scaled to become integers.

SUBSET SUM PROBLEM: Given a set of integers, is there a non-empty subset whose sum is exactly zero?
Equivalently: Given a set of integers and an integer $s$, does any non-empty subset sum to $s$?

Which one of these two is a special case of the other problem?
More Examples for NP-Completeness
Proofs-Dominating Set Problem

Let G be a graph G=(V,E).

- Dominating Set (DS): A subset S ⊆ V is a dominating set if every vertex not in S has a neighbor in S. DS problem asks for the DS with minimum cardinality.

- Define a problem Γ which takes a graph without isolated vertices and a positive integer as input. It asks you whether there is a VC of size ≤ k.

- The problem Γ is NP-complete because we can easily reduce it from VC as follows:
  Given an instance G of VC, we can simply remove all isolated vertices of G and the remaining graph is an instance of Γ. Clearly, the remaining graph has a VC of size ≤ k iff the original graph has a VC of size ≤ k (VC → Γ)

- Now, we will show the reduction Γ → DS
  (Remember the transitivity in reductions: If VC → Γ and Γ → DS, then VC → DS)
More Examples for NP-Completeness
Proofs-Dominating Set Problem

Let us show that DS is NP-complete:

1. DS is in NP because we can verify in poly-time that a given subset of vertices is a DS (All needed is to check the neighbor of each vertex not in the set)

2. \( \Pi \rightarrow DS \) (Sometimes also shown as \( \Pi \sim DS \))

Let \( G \) be a graph without isolated vertices and construct a new graph \( G' \) from \( G \) as follows:

Add an extra copy of each edge of \( G \) and then subdivide one copy of each edge (i.e. replace each edge by a triangle involving a new vertex).

This construction takes \( 2m \) operations where \( m \) is the number of edges. Therefore, it is a poly-time transformation.

Claim: \( G \) has a VC of size \( \leq k \) iff \( G' \) has a DS of size \( \leq k \).

Proof: (\( \Rightarrow \)) Let \( S \) be a VC of \( G \). Let \( w \in V(G')-S \). If \( w \) is a new vertex, then \( w \) has exactly two neighbors which are the endpoints of an original edge. At least one endpoint of the edge must be in \( S \) since \( S \) is a VC of \( G \). Therefore, \( w \) has a neighbor in \( S \).
More Examples for NP-Completeness
Proofs-Dominating Set Problem

If \( w \) is an original vertex, then consider an original edge which is incident with \( w \) (such an edge exists since there is no isolated vertex in \( G \)).

Then, the other endpoint of this edge must be in \( S \) since \( w \notin S \). Hence, \( w \) has a neighbor in \( S \).

Every vertex \( w \) in \( V(G')-S \) has a neighbor in \( S \). Thus, \( S \) is a DS of \( G' \).

\((\Leftarrow)\) Let \( S \) be a DS of \( G' \). \( S=S_0 \cup S_N \) where \( S_0 \) (\( S_N \)) consists of the original (new) vertices. Let \( m \) be the number of original edges in \( G' \) where none of the endpoints of these edges are in \( S \). For each such edge, take one endpoint of the edge. The number of vertices we can choose in this way is \( \leq m \).

Now, if we take the union of this set with \( S_0 \), we get a subset of vertices of \( G \), which forms a VC for \( G \) with size \( \leq |S_0|+m \leq |S_0|+|S_N|=|S| \).

Q.E.D.
Feedback Vertex Set (FVS): A subset $V'$ of the vertex set of a directed graph (digraph) is a FVS if it contains at least one vertex from every directed cycle (circuit) in the graph.

Instance: A digraph $G$ and a positive integer $k \leq |V|$

Question: Does $G$ have a FVS of size $\leq k$?

Claim: FVS is NP-Complete.

Proof:
1. FVS is in NP: Observe that a set $S$ is a FVS iff when $S$ is removed, there is no directed cycle in the remaining graph. Therefore, we can use Tarjan’s algorithm in the remaining graph to check whether there is a circuit or not (Tarjan’s algorithm is a poly-time algorithm which finds all elementary circuits of a given digraph)
2. VC→FVS
   Let $G=(V,E)$ be an undirected graph. We can construct a digraph $G'=(V,A)$ from $G=(V,E)$ in poly-time by replacing each edge with a pair of directed edges (arcs) one in each direction.
More Examples for NP-Completeness
Proofs-Feedback Vertex Set Problem

Claim: G has a VC of size ≤ k iff G’ has a FVS of size ≤ k.

Proof:

(⇒) Let W be a VC of G. Every edge in G is incident with at least one vertex in W; hence, every arc in G’ has at least one endpoint in W. Therefore, W contains at least one vertex from every circuit in G.

(⇐) Let W be a FVS for G’. W contains at least one vertex from every circuit. In particular, it contains at least one vertex from every circuit of length 2. (A circuit of length 2 is just a pair of arcs with opposite directions between the same pair of vertices). Hence, W contains at least one endpoint of each edge.
More Examples for NP-Completeness Proofs-
Largest Common Subgraph Problem

Largest Common Subgraph (LCS):
Instance: Two graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$ and a positive integer $k$.
Question: Is there a graph $G$ with $\geq k$ edges which is a subgraph of both $G_1$ and $G_2$?

Claim: LCS problem is NP-complete.
Proof:

1. LCS is in NP: Let $G_1$ and $G_2$ be the inputs. Assume that we are given a candidate solution, a graph $H=(V^*, E^*)$ with at least $k$ edges and 1-1 mappings $f_1:V^* \rightarrow V_1$ and $f_2:V^* \rightarrow V_2$. Thus, the certificate has size $|H|+|f_1|+|f_2|$.

   If $n$ is the size of the input ($n=|G_1|+|G_2|$), then $|H|=O(n)$ since $H$ is a subgraph of the given graphs and $|f_i|=O(n)$ since both functions are just a list of vertices. Therefore, the size of the certificate is polynomial.
More Examples for NP-Completeness Proofs-
Largest Common Subgraph Problem

In order to verify this certificate:

✓ Check that \( H \) has at least \( k \) edges (\( O(n) \) time).
✓ Check that \( f_1 \) and \( f_2 \) are 1-1 (\( O(n) \) time)
✓ Check whether for any edge \((u,v)\) \( \in \) \( E^* \), \((f_1(u), f_2(v))\) \( \in \) \( E_2 \) (\( O(n^2) \) time)

Hence, the certificate takes poly-time to verify.

2. CLIQUE \( \rightarrow \) LCS

Recall that a clique in an undirected graph is a subset of its vertices such that every two vertices in the subset are connected by an edge. The CLIQUE problem aims to find the largest clique (the clique with the highest number of vertices)

Let \( H \) be an instance of the CLIQUE problem. Take \( G_1=H \) and \( G_2=K_k \) as a particular instance of the LCS problem.

It is clear that \( H \) has a clique of size \( \geq k \) iff \( H \) and \( K_k \) have a common subgraph with \( \geq k(k-1)/2 \) edges.
**Greedy Algorithms**

“Greed .... is good. Greed is right. Greed works.”

Michael Douglas, Wall Street, 1980s.

Is it really good? Does it always work? Answer: Sometimes

An algorithm is “greedy” if it builds up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion.

A greedy algorithm always makes the choice that looks best at the moment.

One can often design many different greedy algorithms for the same problem, each one locally optimizing some different measure. Some of these algorithms may work, some may not.

We can also design greedy “heuristic” or “approximation algorithms” for NP-complete problems
Greedy Algorithms

Inventing greedy algorithms is easy. Proving that they work is difficult.

Two basic analysis techniques:

1. “Greedy Algorithm Stays Ahead”:
   Prove that the greedy algorithm does better than any other algorithm at each step

2. “Exchange Argument”:
   Consider any possible solution to the problem and gradually transform it to the solution found by the greedy algorithm without hurting its quality.
   Hence, the greedy algorithm must have found a solution that is at least as good as any other solution.
Interval Scheduling: The Greedy Algorithm Stays Ahead

Interval Scheduling Problem:

Given a set of requests \( \{1,2,...,n\} \), the \( i^{th} \) request corresponds to an interval of time starting at \( s(i) \) and finishing at \( f(i) \).

A subset of requests is compatible if no two of them overlap in time.

Goal: To accept as large a compatible subset as possible

Compatible sets of maximal size will be called “optimal”
Interval Scheduling: The Greedy Algorithm Stays Ahead

Some Possible Greedy Algorithms:

1. Always select the available request that starts earliest, i.e., the one with the minimal start time $s(i)$.
   
   Does NOT yield an optimal solution. If the earliest request is for a very long interval, we may have to reject a lot of requests for shorter intervals.

   Counter-example:
Some Possible Greedy Algorithms:

2. Always select the available request that requires the smallest interval of time, i.e., the one for which $f(i) - s(i)$ is the smallest.

Sounds better than the first algorithm at first glance, however, still does NOT yield an optimal solution.

Counter-example:

```
+-----------------+-----------------+-----------------+
|                  |                  |                  |
|                  |                  |                  |
+-----------------+-----------------+-----------------+
```

Time
Interval Scheduling: The Greedy Algorithm Stays Ahead

Some Possible Greedy Algorithms:

3. Always select the available request that has the fewest number of noncompatible requests, i.e., select the interval with the fewest “conflicts”

⇒ Still does NOT yield an optimal solution.
⇒ Counter-example:

Unique optimal here: Accept the four requests in the top row
Interval Scheduling: The Greedy Algorithm Stays Ahead

Some Possible Greedy Algorithms:
4. Always select the available request that finishes first, i.e., the request for which \( f(i) \) is as small as possible
   - DOES yield an optimal solution!
   - Proof (Algorithm Analysis):
     \[ A = \text{The set of intervals returned by the algorithm} \]
     \[ O = \text{Optimal set of intervals} \]
     Remark: \( A \) is a compatible set of requests—OBVIOUS
     Remark: \( |A| = |O| \), hence, \( A \) is also an optimal solution—NOT SO OBVIOUS

Show that the greedy algorithm “stays ahead” of the solution \( O \), i.e., the greedy algorithm is doing better in a step-by-step fashion.
Interval Scheduling: The Greedy Algorithm Stays Ahead

Some Possible Greedy Algorithms:

4. Cont’d: Algorithm Analysis
Let \(i_1, i_2, \ldots, i_k\) = Set of requests in A in the order they were added
Let \(j_1, j_2, \ldots, j_m\) = Set of requests in O in the order they were added

Goal: To prove that \(k = m\).
Goal: Each of the intervals of our greedy rule “stays ahead” = finishes at least as soon as the corresponding interval in O

\[\text{Remark: For all indices } r \leq k, \text{ we have } f(i_r) \leq f(j_r)\]
Proof: By induction. Base case \(r = 1\) is clearly true.
Induction hypothesis: Assume that \(f(i_{r-1}) \leq f(j_{r-1})\).

We know that \(f(j_{r-1}) \leq s(j_r)\) since O consists of compatible intervals.
Therefore, \(f(i_{r-1}) \leq s(j_r)\) \(\Rightarrow\) The interval \(j_r\) is in the set \(R\) of available intervals at the time when the greedy algorithm selects \(i_r\).
Interval Scheduling: The Greedy Algorithm Stays Ahead

Some Possible Greedy Algorithms:
4. Cont’d: Algorithm Analysis

The greedy algorithm selects the available interval with the smallest finish time. Since interval \( j_r \) is one of these intervals, we have \( f(i_r) \leq f(j_r) \).

Hence, for each \( r \), the \( r \)th interval that the greedy algorithm selects finishes at least as soon as the \( r \)th interval in \( O \) ("stays ahead").

Theorem: The greedy algorithm returns an optimal set \( A \).

Proof: By contradiction. Assume that \( A \) is not optimal. Then \( m > k \). We have \( f(i_k) \leq f(j_k) \) by the previous statement. Since \( m > k \), there is a request \( j_{k+1} \) in \( O \). This request starts after request \( j_k \) ends and hence after \( i_k \) ends. Therefore, after deleting requests that are not compatible with requests \( i_1, i_2, \ldots, i_k \), the set of possible requests \( R \) still contains \( j_{k+1} \). However, the greedy algorithm stops with request \( i_k \) and it is only supposed to stop when \( R \) is empty—Contradiction.

Q.E.D.
Scheduling to Minimize Lateness: An Exchange Argument

- We have a single resource and a set of $n$ requests

- More flexible requests: Instead of $s(i)$ and $f(i)$, request $i$ has a deadline $d(i)$

- Each request $i$ requires a contiguous time interval of length $t(i)$

- A request $i$ is late if it misses its deadline, i.e., if $f(i) > d(i)$. Lateness of such a request $= l(i) = f(i) - d(i)$

- The goal is to minimize the maximum lateness, $L = \max_i l(i)$

- There are several natural greedy approaches for this problem
Scheduling to Minimize Lateness: An Exchange Argument

- **Approach-1**: Schedule the jobs in order of increasing length \( t(i) \) in order to get the short jobs out of the way quickly.
  - NOT optimal!
  - It completely ignores the deadlines—too simplistic
  - Counter-example: \( t_1=1, \quad d_1=100, \quad t_2=10, \quad d_2=10 \)

- **Approach-2**: Schedule the jobs in order of increasing slack = \( d(i) - t(i) \).
  - Still NOT optimal!
  - Counter-example: \( t_1=1, \quad d_1=2, \quad t_2=10, \quad d_2=10 \)

- **Approach-3**: Earliest Deadline First (EDF)
  - Optimal! Although it never look at the lengths of the jobs...
  - Do the homework that has the earliest deadline
  - Why is it optimal?
Scheduling to Minimize Lateness: An Exchange Argument

**EARLIEST DEADLINE FIRST (EDF):**

- Observe that the schedules that EDF produces has no gaps (no idle time)
- Observe also that there is an optimal schedule with no idle time

How do we prove that EDF is optimal?

- Start by an optimal schedule \( O \)
- Gradually transform (modify) \( O \) preserving its optimality at each step
- Finally transform it into a schedule identical to the schedule \( A \) found by EDF
  - This method is referred to as “exchange argument”

**Inversion:** A schedule \( A' \) has an inversion if a job \( i \) with deadline \( d(i) \) is scheduled before another job \( j \) with earlier deadline \( d(j) < d(i) \)

By definition, the schedule \( A \) produced by EDF has no inversions
Scheduling to Minimize Lateness: An Exchange Argument

EARLIEST DEADLINE FIRST (EDF):

Theorem: All schedules with no inversions and no idle time have the same maximum lateness

Proof:
- If two different schedules have neither inversions nor idle time, then they might not produce exactly the same order of jobs but they can only differ in the order in which jobs with identical deadlines are scheduled.

- Consider such a deadline $d$. In both schedules, the jobs with deadline $d$ are all scheduled consecutively (after all jobs with earlier deadlines and before all jobs with later deadlines).

- Among the jobs with deadline $d$, the last one has the greatest lateness and this lateness does not depend on the order of the jobs (remember that $l(i) = f(i) - d(i)$).

- Hence, they all have the same maximum lateness.
Scheduling to Minimize Lateness: An Exchange Argument

**EARLIEST DEADLINE FIRST (EDF):**

**Theorem:** There is an optimal schedule that has no inversions and no idle time

**Proof:**
- **Remark-1:** If $O$ (optimal schedule) has an inversion, then there is a pair of jobs $i$ and $j$ such that $j$ is scheduled immediately after $i$ and has $d_j < d_i$

  ➤ Indeed, consider an inversion in which a job $a$ is scheduled sometime before a job $b$, and $d_a > d_b$. If we advance in the scheduled order of jobs from $a$ to $b$ one at a time, there has to come a point at which the deadline we see decreases for the first time.

  This corresponds to a pair of consecutive jobs that form an inversion.
Scheduling to Minimize Lateness: An Exchange Argument

**EARLIEST DEADLINE FIRST (EDF):**

- **Remark-2:** After swapping $i$ and $j$, we get a schedule with one less inversion.
  
  $\Rightarrow$ The pair $(i,j)$ formed an inversion in $O$. This inversion is eliminated by the swap and no new inversions are created.

- **Remark-3:** The new swapped schedule has a maximum lateness no larger than that of $O$.
  
  $\Rightarrow$ If we can prove this, then we are done.

The initial schedule can have at most $\binom{n}{2}$ inversions (if all pairs are inverted).

If Remark-3 is true, then after at most $\binom{n}{2}$ swaps, we get an optimal schedule with no inversions.
Scheduling to Minimize Lateness: An Exchange Argument

EARLIEST DEADLINE FIRST (EDF):

- Proof of Remark-3:

  - Assume that each request is scheduled for the time interval \([s(r), f(r)]\) and has lateness \(l'_r\). Let \(L' = \max_r l'_r\) denote the maximum lateness of this schedule.

  - Let \(O\) denote the swapped schedule and \(s(r), f(r), l(r), L\) denote the corresponding quantities in the swapped schedule.

  - Consider the consecutively inverted jobs \(i\) and \(j\). After they are swapped, job \(j\) will get finished earlier. Hence, its lateness does not increase.

  - The only question is: The lateness of job \(i\) may have increased. Does this increase raise the maximum lateness of the whole schedule?

- After the swap, job \(i\) finishes at time \(f(j)\), when job \(j\) was finished in the schedule \(O\). If job \(i\) is late in this new schedule, its lateness is \(\bar{l}(i) = \bar{f}(i) - d(i) = f(j) - d(i)\)
Scheduling to Minimize Lateness: An Exchange Argument

EARLIEST DEADLINE FIRST (EDF):

Proof of Remark-3:

- The crucial point is that i CANNOT be more late in the new schedule $\bar{O}$ than j was in the old schedule $O$.

- If $d(i) > d(j)$, then $I'(i) = f(j) - d(i) < f(j) - d(j) = l'_j$

- Since the lateness of the old schedule $O$ was $L' \geq I_j > \bar{I}_i$, the swap does not increase the maximum lateness of the schedule.

- Therefore, the schedule $A$ produced by the greedy algorithm has optimal maximum lateness $L$.

$\Rightarrow$ We proved that an optimal schedule with no inversions exists. We then proved that all schedules with no inversions have the same maximum lateness. Therefore, the schedule obtained by the greedy algorithm is optimal.
More Examples: Minimum Spanning Tree (MST) Problem

- Kruskal and Prim algorithms are both greedy algorithms and they both produce an optimal solution.

- Proofs are again based on “exchange argument”s.

Recall:
- Given a connected, undirected graph, a “spanning tree” of that graph is a subgraph that is a tree and connects all the vertices together.
- An MST is a spanning tree with minimum total edge weights.
- Unless the graph is a very simple graph, it will have exponentially many different spanning trees. Therefore, finding the MST in poly-time is NOT obvious.

- Three greedy algorithms that find the optimal solution in poly-time:
  - Kruskal’s algorithm
  - Prim’s algorithm
  - Reverse-Delete algorithm
More Examples: Minimum Spanning Tree (MST) Problem

- **Kruskal’s Algorithm:**
  - Start with an empty graph. Successively insert edges in order of increasing cost.
  - Insert each edge as long as it does not create a cycle when added to the already inserted edges.
  - If the edge would create a cycle, discard it and continue.

- **Prim’s Algorithm:**
  - Start with a root node $s$ and greedily grow a tree from $s$ outward.
  - At each step, add the node that can be attached as cheaply as possible to the partial tree that we already have.

- **Reverse-Delete Algorithm:**
  - Backward version of Kruskal’s algorithm.
  - Start with the complete graph and begin deleting edges in order of decreasing cost.
  - Delete each edge $e$ as long as the new graph is not disconnected.
More Examples: Minimum Spanning Tree (MST) Problem

Analyzing the algorithms:

- Assume w.l.o.g. that all edge costs are distinct \( \Rightarrow \) This assumption can be easily eliminated

When is it safe to include an edge in the MST?

Remark (“Cut Property”): Let \( S \) be any subset of nodes that is neither empty nor equal to all of \( V \). Let edge \( e=(v,w) \) be the minimum cost edge with one end in \( S \) and the other end in \( V-S \). Then every MST contains the edge \( e \).

Proof: Let \( T \) be a spanning tree that does not contain \( e \). We need to show that \( T \) does not have the minimum possible cost.

\( \Rightarrow \) Using an exchange argument:

- Identify an edge \( e' \) in \( T \) that is more expensive than \( e \).
- Show that exchanging \( e \) for \( e' \) results in another spanning tree.
- This resulting spanning tree will then be cheaper than \( T \), as desired.
More Examples: Minimum Spanning Tree (MST) Problem

Proof (cont’d):

Find an edge that can be successfully exchanged with e.

Recall that ends of e are v and w. T is a spanning tree so there must be a path P in T from v to w.

Starting at v, suppose that we follow the nodes of P in sequence. There is a first node w’ on P that is in V-S.

Let v’∈S be the node just before w’ on P and let e’=(v’,w’) be the edge joining them. Thus, e’ is an edge with one end in S and the other end in V-S.

If we exchange e for e’, we get a set of edges T’. We claim that T’ is a spanning tree

Clearly, (V,T’) is connected since (V,T) is connected and any path in (V,T) that used the edge e’ can now be rerouted in (V,T’) to follow the portion of P from v’ to v, then the edge e, and then the portion of P from w to w’.

Also, (V,T’) is acyclic: The only cycle in (V,T’∪{e’}) is the one composed of e and the path P, and this cycle is absent in (V,T’) due to the deletion of e’.
More Examples: Minimum Spanning Tree (MST) Problem

Proof (cont’d):

Edge e’ has one end in S and the other end in V-S. However, edge e is the cheapest edge with this property, so $c_e < c_{e'}$. (The inequality is strict since we assumed w.l.o.g. that no two edges have the same cost). Thus, the total cost of $T'$ is less than that of $T$, as desired.

Q.E.D.

Optimality of Kruskal’s and Prim’s Algorithms:

Both algorithms only include an edge when it is justified by the Cut Property.

Theorem: Kruskal’s algorithm produces an MST.

Proof: Consider any edge $e=(v,w)$ added by Kruskal. Let $S$ be the set of nodes to which $v$ has a path just before $e$ is added. Clearly, $v \in S$ but $w \notin S$ since adding $e$ does not create a cycle.

Moreover, no edge from $S$ to $V-S$ has been encountered yet since any such edge could have been added without creating a cycle and hence would have been added by Kruskal’s algorithm.
More Examples: Minimum Spanning Tree (MST) Problem

Proof (cont’d):

Thus, e is the cheapest edge with one end in S and the other in V-S and therefore by the Cut Property, it belongs to every MST.

So, if we show that the output (V,T) of Kruskal’s algorithm is in fact a spanning tree of G, then we will be done.

Clearly, (V,T) contains no cycles since the algorithm is designed to avoid cycles.

If (V,T) were not connected, then there would exist a non-empty subset of nodes S (not equal to all of C) such that there is no edge from S to V-S. However, this contradicts the behavior of the algorithm: Since G is connected, we know that there is at least one edge between S and V-S and the algorithm will add the first of these that it encounters.

Q.E.D.

Theorem: Prim’s algorithm produces an MST.

Proof: We will show that Prim’s algorithm only adds edges belonging to every MST.

In each iteration, there is a set S ⊆ V on which a partial spanning tree has been constructed.

In each iteration, a node v and edge e are added that minimize the quantity

\[ \min_{e=(u,v): u \in S} C_e \]
More Examples: Minimum Spanning Tree (MST) Problem

Proof (cont’d):
- By definition, e is the cheapest edge with one end in S and the other end in V-S. Therefore, by the Cut Property, it is in every MST.
- It is also obvious that Prim’s algorithm produces a spanning tree (in each iteration, a partial spanning tree is constructed and the algorithm continues until all vertices are spanned)
- Hence, Prim’s algorithm produces an MST.

Remark (“Cycle Property”): Assume that all edge costs are distinct. Let C be any cycle in G and let edge e=(v,w) be the most expensive edge belonging to C. Then e does not belong to any MST of G.

Proof: Let T be a spanning tree that contains e. We can show that T cannot be minimum as follows (again with “exchange argument”):

- The edges of C other than e form, by definition, a path P with one end at v and the other at w. If we follow P from v to w, we begin in S and end up in V-S. Therefore, there is some edge e’ on P that crosses from S to V-S.
- If we exchange e for e’, the resulting graph T’ will be connected and will have no cycles. Hence, T’ will be a spanning tree. Since e is the most expensive edge on the cycle C and e’ belongs to C, e’ must be cheaper than e and hence T’ is cheaper than T.

Q.E.D.
More Examples: Minimum Spanning Tree (MST) Problem

Theorem: The reverse-delete algorithm produces an MST.

Proof: Consider any edge $e=(v,w)$ removed by Reverse-Delete. At the time that $e$ is removed, $e$ lies on a cycle $C$. Since it is the first edge encountered by the algorithm in decreasing order of edge costs, it must be the most expensive edge on $C$. Thus, by the cycle property, $e$ does not belong to any MST.

So, if we show that the output $(V,T)$ of Reverse-Delete is an MST, we will be done. Clearly, $T$ is connected since the algorithm never removes an edge when it will disconnect the graph.

Suppose (for contradiction) that $(V,T)$ contains a cycle $C$. Consider the most expensive edge on $C$. This edge would be removed by the algorithm since its removal does not disconnect the graph. Contradiction. Therefore, $(V,T)$ does not contain a cycle. Hence, the theorem follows.

Q.E.D.
More Examples: Minimum Spanning Tree (MST) Problem

- Eliminating the assumption that all edge costs are distinct:
  - Take the graph instance and perturb all edge costs by different extremely small numbers (\( \varepsilon_1, \varepsilon_2, \varepsilon_3 \) etc.)

  This way, any two costs that differed originally will still have the same relative order since the perturbations are too small.

  Perturbations effectively serve simply as “tie breakers” to resolve comparisons among costs that used to be equal.

  Any MST for the new, perturbed instance is also an MST for the original instance.

- Observe that the above is actually a poly-time reduction!
Dynamic Programming

- In some problems, there is no natural greedy algorithm that works

- Dynamic programming is not “programming” but an “algorithm design technique”

- Intuition is from “divide and conquer”, essentially the opposite of greedy strategy

- Dynamic programming does NOT ALWAYS give the optimal solution

Method:
- Implicitly explore the space of all possible solutions
- By carefully decomposing things into a series of subproblems and
- By building up correct solutions to larger and larger subproblems
Weighted Interval Scheduling: A Recursive Procedure

- Each interval has a certain value (weight) and we aim to accept a set of maximum value

- The greedy algorithm (EDF) does not work here

- Indeed, no natural greedy algorithm is known for this problem

Recall the notation:
- \( n \) requests labeled 1, 2, ..., \( n \)
- Each request \( i \) has start time \( s(i) \) and finish time \( f(i) \)
- Each interval has a value/weight \( v_i \)

Suppose that the requests are sorted in order of nondecreasing finish time \( f_1 \leq f_2 \leq ... \leq f_n \)
Weighted Interval Scheduling: A Recursive Procedure

Define \( p(j) \) for an interval \( j = \) Largest index \( i < j \) such that intervals \( i \) and \( j \) are disjoint

\( \Rightarrow \) \( i \) is the leftmost interval that ends before \( j \) begins

\( \Rightarrow \) \( P(j) = 0 \) if no request \( i < j \) is disjoint from \( j \)

Consider an optimal solution \( O \). Either the last interval \( n \) belongs to \( O \) or it does not.

- If \( n \in O \), then:
  - No interval indexed strictly between \( p(n) \) and \( n \) can belong to \( O \)
  - \( O \) must include an optimal solution to the problem consisting of requests \( \{1, 2, \ldots, p(n)\} \)

- If \( n \notin O \), then \( O \) is simply equal to the optimal solution of the problem consisting of requests \( \{1, 2, \ldots, n-1\} \)
Weighted Interval Scheduling: A Recursive Procedure

For any \(1 \leq j \leq n\):

- \(\text{OPT}(j) = \max(v_j + \text{OPT}(p_j), \text{OPT}(j-1))\)
- Request \(j\) belongs to an optimal solution on the set \(\{1, 2, \ldots, j\}\) if and only if \(v_j + \text{OPT}(p_j) \geq \text{OPT}(j-1)\)

Consider the following pseudocode:

\[
\text{Compute}_\text{Opt}(j) \\
\text{If } j = 0 \text{ then} \\
\quad \text{Return } 0 \\
\text{Else} \\
\quad \text{Return } \max(v_j + \text{Compute}_\text{Opt}(p(j)), \text{Compute}_\text{Opt}(j-1)) \\
\text{Endif}
\]

If we really implemented the algorithm as above, it would take exponential time due to repeated & unnecessary calls for recursive procedures.
Weighted Interval Scheduling: A Recursive Procedure

Consider this example:

\[ v_1 = 2 \]
\[ v_2 = 4 \]
\[ v_3 = 4 \]
\[ v_4 = 7 \]
\[ v_5 = 2 \]
\[ v_6 = 1 \]

\[ p(1) = 0 \]
\[ p(2) = 0 \]
\[ p(3) = 1 \]
\[ p(4) = 0 \]
\[ p(5) = 3 \]
\[ p(6) = 3 \]

The tree of subproblems grows very quickly:
Memoizing the recursion:

Memoization = A technique in which partial results are recorded (forming a memo) and then can be reused later without having to recompute them.

Observe that our recursive algorithm is really only solving n+1 different subproblems: Compute_Opt(0), Compute_Opt(1), ..., Compute_Opt(n).

We can eliminate this redundancy by storing the value of Compute_Opt in a globally accessible place the first time we compute it and then access it whenever needed. The new pseudocode is:

```plaintext
M_Compute_Opt(j)
   If j=0 then
      Return 0
   Else if M[j] is not empty then
      Return M[j]
   Else
      Define M[j]=max(vj+M_Compute_Opt(p(j)), M_Compute_Opt(p(j-1)))
      Return M[j]
   Endif
```

Assuming that the intervals are sorted by their finish time, the new running time is O(n).
In the weighted interval scheduling problem, we can directly compute the entries in M by an iterative algorithm rather than using memoized recursion:

\[ \text{Iterative\_Compute\_Opt} \]
\[ M[0] = 0 \]
\[ \text{For } j = 1, 2, \ldots, n \]
\[ M[j] = \max(v_j + M[p(j)], M[j-1]) \]
\[ \text{Endfor} \]

When designing a dynamic programming algorithm we have to make sure that:

- There are only a polynomial number of subproblems
- The solution to the original problem can be easily computed from the solutions to the subproblems
- There is a natural ordering on subproblems from “smallest” to “largest” together with an easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems
In the weighted interval scheduling problem, the recurrence was based on a *binary* choice: Either the interval n belonged to an optimal solution or it did not.

Here: The recurrence will involve multiway choices. At each step, we have a polynomial number of possibilities to consider for the structure of the optimal solution.

**Least Squares Problem:** Suppose our data consists of a set $P$ of $n$ points in the plane, denoted by $(x_1,y_1), (x_2,y_2), \ldots, (x_n,y_n)$ and suppose $x_1 \leq x_2 \leq \ldots \leq x_n$.

Given a line $L$ defined by the equation $y=ax+b$, the “error” of $L$ wrt. $P$ is the sum of its squared distances to the points in $P$:

$$\text{Error}(L, P) = \sum_{i=1}^{n} \left( y_i - ax_i - b \right)^2$$

There is a formula for the line with minimum error.
Segmented Least Squares: Multi-way Choices

- We often have data that look like these:

- Any single line would have a terrible error. Therefore, we wish to say something like: “The points lie roughly on a sequence of 2 lines”

- Segmented Least Squares Problem Formulation:

  Let us use \( p_i \) to denote the point \((x_i, y_i)\). Each segment is a subset of \( O \) that represents a contiguous set of x-coordinates, i.e., it is a subset of the form \( \{p_i, p_{i+1}, \ldots, p_{j-1}, p_j\} \) for some indices \( i \leq j \).
Segmented Least Squares: Multi-way Choices

Segmented Least Squares Problem Formulation:

For each segment $S$ in the partition, we compute the line minimizing the error wrt. the points in $S$

The penalty of a partition is defined to be a sum of the following:

i) The number of segments into which we partition $P$, times a given fixed multiplier $C > 0$

ii) For each segment, the error value of the optimal line through that segment

Goal: Find a partition with minimum penalty

There are exponentially many possible partitions and initially it is not clear that we can find an efficient algorithm
Segmented Least Squares: Multi-way Choices

Some Useful Observations:

1. The last point \( p_n \) belongs to a single segment in the optimal partition and that segment begins at some earlier point \( p_i \).

2. If we knew the identity of the last segment \( p_i, \ldots, p_n \), then we could remove those points from consideration and recursively solve the problem on the remaining points \( p_1, \ldots, p_{i-1} \).

Let \( \text{OPT}(i) = \text{Optimum solution for the points } p_1, \ldots, p_i \)
and \( e_{ij} = \text{Minimum error of any line wrt. } p_i, p_{i+1}, \ldots, p_j \)

Then:

If the last segment of the optimal partition is \( p_i, \ldots, p_n \), then the value of the optimal solution is \( \text{OPT}(n) = e_{in} + C + \text{OPT}(i-1) \) where \( C \) is some constant.

We can use the same observation for the subproblem consisting of the points \( p_1, \ldots, p_j \).

We see that to get \( \text{OPT}(j) \) we should find the best way to produce a final segment \( p_i, \ldots, p_j \) together with an optimal solution \( \text{OPT}(i-1) \) for the remaining points.
Segmented Least Squares: Multi-way Choices

In other words:

For the subproblem on the points $p_1,...,p_j$

$$OPT(j) = \min_{1 \leq i \leq j} (e_{ij} + C + OPT(i-1))$$

and the segment $p_i,...,p_j$ is used in an optimum solution for the subproblem if and only if the minimum is obtained using index $i$.

We can now design the pseudocode to implement the above result:

```
Segmented_Least_Squares(n)
Array M[0,...,n]
Set M[0]=0
For all pairs $i \leq j$
    Compute the least squares error $e_{ij}$ for the segment $p_i,...,p_j$
Endfor
For $j=1,2,...,n$
    Use the above recurrence to compute $M[j]$
Endfor
Return $M[n]$
```
Segmented Least Squares: Multi-way Choices

We can then implement below to find the segments:

\[ \text{Find\_Segments}(j) \]
\[ \text{If } j=0 \text{ then} \]
\[ \quad \text{Output nothing} \]
\[ \text{Else} \]
\[ \quad \text{Find an } i \text{ that minimizes } e_{ij} + C + M[i-1] \]
\[ \quad \text{Output the segment } \{p_i, \ldots, p_j\} \text{ and the result of } \text{Find\_Segments}(i-1) \]
\[ \text{Endif} \]

Running time:

- Computing the values of all least square errors \( e_{ij} \)
  - There are \( O(n^2) \) pairs. For each pair, the formula to compute \( e_{ij} \) takes \( O(n) \) time.
  - Thus, the total running time to compute \( e_{ij} \) values is \( O(n^3) \)
- Once all the \( e_{ij} \) values have been determined
  - The algorithm has \( n \) iterations for \( j=1,\ldots,n \)
  - For each \( j \), we need to determine the minimum value in the recurrence to fill in the array entry \( M[j] \). This takes time \( O(n) \) for each \( j \)
  - Thus, the total running time is \( O(n^2) \) once all the \( e_{ij} \) values have been determined
Subset Sums and Knapsacks: Adding a Variable

- Recall the subset sum problem:
  - Given a set of integers and an integer $W$, does any non-empty subset sum to $W$?
  - Given a set of requests $\{1,2,...,n\}$ and a single machine (resource), we can use this resource only for the period between time 0 and time $W$ for some number $W$. Each request corresponds to a job that requires time $w_i$ to process. If our goal is to process jobs so as to keep the machine as busy as possible up to the “cut-off” $W$, which jobs should we choose?

- A dynamic programming approach to the subset sum problem:

Let $OPT(i) =$ Best possible solution using a subset of the requests $\{1,2,...,i\}$

To find out the value for $OPT(n)$, we not only need the value of $OPT(n-1)$ but we also need the best solution we can get using a subset of the first $n-1$ items and total allowed weight $W-w_n$.

We will use many more subproblems: One for each initial set $\{1,2,...,i\}$ of items and each possible value for the remaining available weight $w$. 
Subset Sums and Knapsacks: Adding a Variable

A dynamic programming approach to the subset sum problem:

A subproblem for each \(i=0,1,\ldots,n\) and each integer \(0 \leq w \leq W\).

\[ OPT(i, w) = \text{Value of the optimal solution using a subset of the items \{1,\ldots,i\} with maximum allowed weight } w \]

where the maximum is over subsets \(S \subseteq \{1,\ldots,i\}\) that satisfy \(\sum_{j \in S} w_j \leq w\)

\[ OPT(n, W) \text{ is the quantity we are looking for in the end.} \]

Let \(O\) denote an optimum solution for the original problem.

- If \(n \not\in O\), then \(OPT(n, W) = OPT(n-1, W)\) since we can simply ignore item \(n\)
- If \(n \in O\), then \(OPT(n, W) = w_n + OPT(n-1, W-w_n)\) since we now seek to use the remaining capacity of \(W-w_n\) in an optimal way across items 1,2,\ldots,n-1
- When \(W < w_n\), we must have \(OPT(n, W) = OPT(n-1, W)\)

In short, we have the following recursion:

- If \(w < w_i\), then \(OPT(i, w) = OPT(i-1, w)\)
- Else \(OPT(i, w) = \max (OPT(i-1, w), w_i + OPT(i-1, w-w_i))\)
Subset Sums and Knapsacks: Adding a Variable

A dynamic programming approach to the subset sum problem:

We want to design an algorithm that builds up a table of all \( \text{OPT}(i,w) \) values while computing each of them at most once.

\[ \text{Subset Sum}(n,W) \]

Array \( M[0...n, 0...W] \)

Initialize \( M[0,w]=0 \) for each \( w=0,1,...,W \)

For \( i=1,2,...,n \)

For \( w=0,...,W \)

Use the recurrence to compute \( M[i,w] \)

Endfor

Endfor

Return \( M[n,W] \)

As an example, consider an instance with weight limit \( W=6 \) and \( n=3 \) items of sizes \( w_1=w_2=2 \) and \( w_3=3 \). Draw the tables for \( M[i,w] \)

The optimal value should be \( \text{OPT}(3,6)=5 \)
Subset Sums and Knapsacks: Adding a Variable

- A dynamic programming approach to the subset sum problem:
  Running time: Calculating each entry of the table takes $O(1)$ time. Therefore, the algorithm runs in time $O(nW)$

Notice that the running time is not a polynomial function of $n$ but rather a polynomial function of $n$ and $W$ (the largest integer involved in defining the problem)

- We call such algorithms “pseudo-polynomial”
- Recall that subset sum problem is NP-complete anyway

- A dynamic programming approach to the knapsack problem:
  Recall that each item $i$ has a distinct value $v_i$ in addition to the nonnegative weight $w_i$.
  If $w < w_i$ then $\text{OPT}(i,w) = \text{OPT}(i-1,w)$
  Else $\text{OPT}(i,w) = \max(\text{OPT}(i-1,w), v_i + \text{OPT}(i-1,w-w_i))$

Hence, knapsack problem can also be solved in $O(nW)$ time.
Shortest Paths in a Graph

Recall Dijkstra’s shortest path (SP) algorithm

When there are negative edge costs, Dijkstra does not work!

Observe that adding a large constant to each edge gives inaccurate answers!

What can be the motivation to have negative edge costs?
Ex: Transactions in a financial setting

An algorithm more flexible and decentralized than Dijkstra is needed

Bellman-Ford algorithm, which is based on dynamic programming, serves this purpose
Shortest Paths in a Graph

A dynamic programming based approach:

First, we need to check that no negative cycles exist. Otherwise, nothing would work.

Remark: If a graph G has no negative cycles, then there is an SP from s to t that is simple (does not repeat nodes) and hence has at most n-1 edges, where n is the total number of vertices in G.

Proof: Since every cycle has nonnegative cost, the SP P from s to t with the fewest number of edges does not repeat any vertex v. Because if it did repeat a vertex v, we could remove the portion of P between consecutive visits to v, resulting in a path of no greater cost and fewer edges.

Let \( \text{OPT}(i,v) = \) Minimum cost of a \((v,t)\) path using at most \(i\) edges

Observe that our original problem is to compute \( \text{OPT}(n-1,s) \)
A dynamic programming based approach:

Let us fix a path \( P \) (as above) representing \( \text{OPT}(i,v) \)

- If the path uses at most \( i-1 \) edges, then \( \text{OPT}(i,v) = \text{OPT}(i-1,v) \)
- If the path uses \( i \) edges and the first edge is \((v,w)\), then \( \text{OPT}(i,v) = c_{vw} + \text{OPT}(i-1,w) \)

We can therefore have the following recursion formula:

If \( i > 0 \) then \( \text{OPT}(i,v) = \min(\text{OPT}(i-1,v), \min_{w \in V}(\text{OPT}(i-1,w) + c_{vw})) \)
Shortest Paths in a Graph

A dynamic programming based approach:

M: Optimum value for that particular subproblem

ShortestPath(G,s,t)

n=number of vertices in G
Array M[0...n-1, V]
Define M[0,t]=0 and M[0,v]=∞ for all other v∈V
For i=1...n-1
  For v∈V in any order
    Compute M[i,v] using the recurrence
  Endfor
Endfor
Return M[n-1,s]
Shortest Paths in a Graph

A dynamic programming based approach:

The table M has $n^2$ entries and each entry can take $O(n)$ time to compute since there are at most $n$ nodes we have to consider.

Therefore, this dynamic programming based method takes $O(n^3)$ time.

Entries corresponding to the values $M[i,v]$:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>∞</td>
<td>-3</td>
<td>-3</td>
<td>-4</td>
<td>-6</td>
<td>-6</td>
</tr>
<tr>
<td>b</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td>-2</td>
<td>-2</td>
<td>-2</td>
</tr>
<tr>
<td>c</td>
<td>∞</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>d</td>
<td>∞</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>e</td>
<td>∞</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Shortest Paths in a Graph

An improved running time analysis for the dynamic programming based approach:

Consider the computation of the array entry $M[i,v]$ according to the recurrence.

We assumed it could take $O(n)$ time to compute this minimum since there are $n$ possible nodes $w$.

However, we need only compute this minimum over all nodes $w$ for which $v$ has an edge to $w$. Let us use $n_v$ to denote this number. Then, it takes time $O(n_v)$ to compute the array entry $M[i,v]$.

We have to compute an entry for every node $v$ and every index $0 \leq i \leq n-1$. This gives a running time bound of $O(n \sum n_v)$.

Since we have a digraph here and each edge leaves exactly one of the nodes in $V$, each edge is counted exactly once by this expression. Hence, $\sum n_v = m$ where $m$ is the total number of edges. Therefore, the total running time bound is $O(mn)$.
Balanced Partition Problem

**Problem Formulation:**
Input: A set of integers each in the range 0...K.
Goal: Partition these integers into two subsets $S_1$ and $S_2$ s.t. You minimize $|S_1 - S_2|$

**A dynamic programming based approach:**

- Let $P(i,j) = 1$ if some subset of $\{A_1, \ldots, A_i\}$ has a sum of $j$, and 0 otherwise

- Here, $i$ varies from 1 to $n$ and $j$ varies from 0 to $nK$

- We then have $P(i,j) = 1$ if $P(i-1,j) = 1$ (The first $i-1$ sets already sum up to $j$) OR
  - if $P(i-1,j-A_i) = 1$ (The first $i-1$ sets sum up to $j-A_i$ and the next one already equals $A_i$. Therefore, the first $i$ sets sum up to $j$)

- We can write the recurrence relation as follows:
  $P(i,j) = \max\{P(i-1,j), P(i-1, j-A_i)\}$
Balanced Partition Problem

A dynamic programming based approach:

The table has $O(n^2k)$ entries and each takes $O(1)$ time to compute. Hence, the total running time is $O(n^2k)$

We still need to solve the original problem. Let $S = \left( \sum A_i \right) / 2$. If I can hit $S$ exactly, then my objective value is going to be zero.

My goal is to find $\min_{i \leq S} \{S-i : P(n,i) = 1\}$. Of all possible sums $i$ that I can get by dividing up these $n$ numbers, I want to pick the one that minimizes the distance from $S$.

Let sum($S_1$) = $i$ and sum($S_2$) = $2S-i$

Now if I look at my objective, I see that $|S_1 - S_2| = 2S - 2i$, which is equal to $2\min_{i \leq S} \{S-i : P(n,i) = 1\}$.

So, this minimization expression gives the optimal objective value to my problem.