Chapter 4
Dynamic Programming

CS 473: Algorithms, Fall 2018
September 6, 2018

4.1 Optimal Triangulations
4.1.0.1 The problem

(A) A convex polygon $P$ in the plane.
(B) Compute triangulation of $P$ of minimum total length.
(C) total length of diagonals of triangulation of $P$ +
(D) , plus length of perimeter of $P$ are minimized.
(E) What is the input?
(F) What is the output?

4.1.0.2 A polygon and possible triangulations

4.1.0.3 Definitions

Definition 4.1.1. (A) A set $S \subseteq \mathbb{R}^d$ is convex if for any to $x, y \in S$, the segment $xy$ is contained in $S$.
(B) convex polygon is a closed cycle of segments, with no vertex pointing inward. Formally, it is a simple closed polygonal curve which encloses a convex set.
(C) diagonal is a line segment connecting two vertices of a polygon which are not adjacent.
(D) triangulation is a partition of a convex polygon into (interior) disjoint triangles using diagonals.
4.1.0.4 How to triangulate?

Observation 4.1.2. Any triangulation of a convex polygon with \( n \) vertices is made out of exactly \( n - 2 \) triangles.

4.1.0.5 Idea: Divide and conquer by guessing a triangle

4.1.0.6 Details...

(A) Purpose: find triangulation of \( P \) with min total length.

(B) \( \Delta \) = min total length of diagonals of triangulation.

(C) Q: How to describe subproblems succinctly?

4.1.0.7 Idea: Divide and conquer by guessing a triangle

4.1.1 Idea: Divide and conquer by guessing a triangle

4.1.1.1 But which triangle?

4.1.1.1 But which triangle?
4.1.1.2 Guess triangle on the edge 1 − 2

To this end, we assume that the polygon is specified as list of vertices 1…n in a clockwise ordering. Namely, the input is a list of the vertices of the polygon, for every vertex, the two coordinates are specified. The key observation, is that in any triangulation of \( P \), there exist a triangle that uses the edge between vertex 1 and \( n \) (red edge in figure on the left).

In particular, removing the triangle using the edge 1 − \( n \) leaves us with two polygons which their vertices are consecutive along the original polygon.

4.1.1.3 Recursive solution

(A) \( M[i,j] \): price of triangulating a subpolygon from vertex \( i \) to vertex \( j \).
(B) Diagonals: twice. Perimeter: once.
(C) Get recurrence:

\[
M[i,j] = \begin{cases} 
0 & j \leq i \text{ or } j = i + 1 \\
\min_{i<k<j} \left( \Delta(i,j,k) + M[i,k] + M[k,j] \right) & \text{Otherwise}
\end{cases}
\]

Where \( d(i,j) = \sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2} \) and \( \Delta(i,j,k) = d(i,j) + d(j,k) + d(i,k) \),

(D) \( p_i = (x[i], y[i]) \)
(E) What is \( M[1,n] \)?

4.1.1.4 Running time...

Using dynamic programming (or just memoization), we get an algorithm that computes optimal triangulation in \( O(n^3) \) time using \( O(n^2) \) space.

4.1.1.5 Tidbit...

Given a convex polygon with \( n + 2 \) vertices, the number of different triangulations it has, is the Catalan number \( C_n = \frac{1}{n+1} \binom{2n}{n} \). Euler proved this in the 18th century.

4.2 Total Recall on DAGs

4.2.0.1 DAGs

Definition 4.2.1. A DAG is a directed acyclic graph. That is a directed graph with no cycles.

4.2.1 Topological Ordering/Sorting

4.2.1.1 You should already know that...

Definition 4.2.2. A topological ordering/topological sorting of \( G = (V,E) \) is an ordering \( \prec \) on \( V \) such that if \( (u,v) \in E \) then \( u \prec v \).
Lemma 4.2.3. A directed graph $G$ can be topologically ordered iff it is a DAG.

4.2.2 More things you should already know

4.2.2.1 You should already know that...

Lemma 4.2.4. A DAG $G = (V, E)$ with $n$ vertices and $m$ edges can be topologically sorted in $O(n + m)$ time.

Lemma 4.2.5. The strong connected components of a directed graph, can be computed in $O(n + m)$ time.

4.3 Maximum Weighted Independent Set in Trees

4.3.0.1 Maximum Weight Independent Set Problem

Problem 4.3.1 (Max Weight Independent Set).

Input: Graph $G = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$.

Goal: Find maximum weight independent set in $G$.

Maximum weight independent set in above graph: $\{B, D\}$.

4.3.0.2 Maximum Weight Independent Set in a Tree

Problem 4.3.2 (Max W. Independent Set in Tree).

Input: Tree $T = (V, E)$ and weights $w(v) \geq 0$ for each $v \in V$.

Goal: Find maximum weight independent set in $T$.

Maximum weight independent set in above tree: ??
4.3.0.3 Towards a Recursive Solution

(A) For an arbitrary graph $G$:

(A) Number vertices as $v_1, v_2, \ldots, v_n$

(B) Find recursively optimum solutions without $v_n$ (recurse on $G - v_n$) and with $v_n$ (recurse on $G - v_n - N(v_n)$ & include $v_n$).

(C) Saw that if graph $G$ is arbitrary there was no good ordering that resulted in a small number of subproblems.

(B) What about a tree?

(C) Natural candidate for $v_n$ is root $r$ of $T$?

4.3.1 Towards a Recursive Solution

4.3.1.1 Maximum Weight Independent Set in a Tree

(A) Natural candidate for $v_n$ is root $r$ of $T$?

(B) Let $O$ be an optimum solution to the whole problem.

\textbf{Case} $r \notin O$: $O$ contains optimum solution for each subtree hanging from a child of $r$.

\textbf{Case} $r \in O$: None of children of $r$ are in $O$.

$O \setminus \{r\}$ contains an optimum solution for each subtree hanging at a grandchild of $r$.

(C) Subproblems? Subtrees of $T$ hanging at nodes in $T$.

4.3.1.2 A Recursive Solution

(A) $T(u)$: subtree of $T$ hanging at node $u$.

(B) $OPT(u)$: max weighted independent set value in $T(u)$.

(C) $OPT(u) = \max \left\{ \sum_{v, \text{child of } u} OPT(v), w(u) + \sum_{v, \text{grandchild of } u} OPT(v) \right\}$

4.3.1.3 Iterative Algorithm

(A) Compute $OPT(u)$ bottom up.

(B) To evaluate $OPT(u)$ need to have computed values of all children and grandchildren of $u$

(C) What is an ordering of nodes of a tree $T$ to achieve above?

(D) Post-order traversal of a tree.
4.3.1.4 Iterative Algorithm

\[
\text{MIS-Tree}(T): \\
\text{Let } v_1, v_2, \ldots, v_n \text{ be a post-order traversal of nodes of } T \\
\text{for } i = 1 \text{ to } n \text{ do} \\
M[v_i] = \max \left( \sum_{v_j \text{ child of } v_i} M[v_j], w(v_i) + \sum_{v_j \text{ grandchild of } v_i} M[v_j] \right) \\
\text{return } M[v_n] \quad \text{// Note: } v_n \text{ is the root of } T
\]

Space: \(O(n)\) to store the value at each node of \(T\)

Running time:
(A) Naive bound: \(O(n^2)\) since each \(M[v_i]\) evaluation may take \(O(n)\) time and there are \(n\) evaluations.
(B) Better bound: \(O(n)\). A value \(M[v_j]\) is accessed only by its parent and grand parent.

4.3.1.5 Example

\[
\begin{array}{ccccccccccc}
  & & & r & & & & & & 10 \\
  & & & & & 5 & & & & \\
  & & & a & & & & & & \\
  & & & & & & 4 & & c & d & 4 & e & f & g & 11 \\
  & & & & & & & & & & & 2 & h & i & j & 8 \\
\end{array}
\]

4.3.1.6 Dominating set

Definition 4.3.3. \(G = (V, E)\). The set \(X \subseteq V\) is a dominating set, if any vertex \(v \in V\) is either in \(X\) or is adjacent to a vertex in \(X\).

Problem 4.3.4. Given weights on vertices, compute the minimum weight dominating set in \(G\).

Dominating Set is NP-Hard!

4.4 DAGs and Dynamic Programming

4.4.0.1 Recursion and DAGs

Observation 4.4.1. \(A\): recursive algorithm for problem \(\Pi\).

For each instance \(I\) of \(\Pi\) there is an associated DAG \(G(I)\).
(A) Create directed graph \(G(I)\) as follows...
(B) For each sub-problem in the execution of \(A\) on \(I\) create a node.
(C) If sub-problem \(v\) depends on or recursively calls sub-problem \(u\) add directed edge \((u, v)\) to graph.
(D) \(G(I)\) is a DAG. Why? If \(G(I)\) has a cycle then \(A\) will not terminate on \(I\).
4.4.1 Iterative Algorithm for...

4.4.1.1 Dynamic Programming and DAGs

Observation 4.4.2. An iterative algorithm $B$ obtained from a recursive algorithm $A$ for a problem $\Pi$ does the following:

For each instance $I$ of $\Pi$, it computes a topological sort of $G(I)$ and evaluates sub-problems according to the topological ordering.

(A) Sometimes the DAG $G(I)$ can be obtained directly without thinking about the recursive algorithm $A$
(B) In some cases (not all) the computation of an optimal solution reduces to a shortest/longest path in DAG $G(I)$
(C) Topological sort based shortest/longest path computation is dynamic programming!

4.4.2 A quick reminder...

4.4.2.1 A Recursive Algorithm for weighted interval scheduling

Let $O_i$ be value of an optimal schedule for the first $i$ jobs.

\[
\begin{align*}
\text{Schedule}(n): & \\
& \text{if } n = 0 \text{ then return } 0 \\
& \text{if } n = 1 \text{ then return } w(v_1) \\
& O_{p(n)} \leftarrow \text{Schedule}(p(n)) \\
& O_{n-1} \leftarrow \text{Schedule}(n-1) \\
& \text{if } (O_{p(n)} + w(v_n) < O_{n-1}) \text{ then} \\
& \quad O_n = O_{n-1} \\
& \text{else} \\
& \quad O_n = O_{p(n)} + w(v_n) \\
& \text{return } O_n
\end{align*}
\]

4.4.3 Weighted Interval Scheduling via...

4.4.3.1 Longest Path in a DAG

Given intervals, create a DAG as follows:

(A) Create one node for each interval, plus a dummy sink node $0$ for interval $0$, plus a dummy source node $s$.
(B) For each interval $i$ add edge $(i, p(i))$ of the length/weight of $v_i$.
(C) Add an edge from $s$ to $n$ of length $0$.
(D) For each interval $i$ add edge $(i, i - 1)$ of length $0$. 
4.4.3.2 Example

\[
\begin{align*}
p(5) &= 2, \\
p(4) &= 1, \\
p(3) &= 1, \\
p(2) &= 0, \\
p(1) &= 0
\end{align*}
\]

\[
\begin{array}{cccccc}
30 & 1 & 70 & 3 & 80 & 4 \\
20 & 2 & 10 & 5
\end{array}
\]

4.4.3.3 Relating Optimum Solution

(A) Given interval problem instance \( I \) let \( G(I) \) denote the DAG constructed as described.

(B) Claim 4.4.3. Optimum solution to weighted interval scheduling instance \( I \) is given by longest path from \( s \) to \( 0 \) in \( G(I) \).

(C) Assuming claim is true,

\( (A) \) If \( I \) has \( n \) intervals, DAG \( G(I) \) has \( n+2 \) nodes and \( O(n) \) edges. Creating \( G(I) \) takes \( O(n \log n) \) time: to find \( p(i) \) for each \( i \). How?

\( (B) \) Longest path can be computed in \( O(n) \) time — recall \( O(m+n) \) algorithm for shortest/longest paths in DAGs.

4.4.3.4 DAG for Longest Increasing Sequence

Given sequence \( a_1, a_2, \ldots, a_n \) create DAG as follows:

(A) add sentinel \( a_0 \) to sequence where \( a_0 \) is less than smallest element in sequence

(B) for each \( i \) there is a node \( v_i \)

(C) if \( i < j \) and \( a_i < a_j \) add an edge \((v_i, v_j)\)

(D) find longest path from \( v_0 \)
4.5 Edit Distance and Sequence Alignment

4.5.0.1 Spell Checking Problem

(A) Given a string “exponen” that is not in the dictionary, how should a spell checker suggest a nearby string?

(B) What does nearness mean?

(C) **Question:** Given two strings \(x_1x_2\ldots x_n\) and \(y_1y_2\ldots y_m\) what is a distance between them?

(D) **Edit Distance:** minimum number of “edits” to transform \(x\) into \(y\).

4.5.0.2 Edit Distance

**Definition 4.5.1.** Edit distance between two words \(X\) and \(Y\) is the number of letter insertions, letter deletions and letter substitutions required to obtain \(Y\) from \(X\).

**Example 4.5.2.** The edit distance between FOOD and MONEY is at most 4:

\[
\text{FOOD} \rightarrow \text{MOOD} \rightarrow \text{MONOD} \rightarrow \text{MONED} \rightarrow \text{MONEY}
\]

4.5.0.3 Edit Distance: Alternate View

Alignment Place words one on top of the other, with gaps in the first word indicating insertions, and gaps in the second word indicating deletions.

\[
\begin{array}{c}
F \quad O \quad O \quad D \\
M \quad O \quad N \quad E \quad Y
\end{array}
\]

Formally, an alignment is a set \(M\) of pairs \((i, j)\) such that each index appears at most once, and there is no “crossing”: \(i < i'\) and \(i\) is matched to \(j\) implies \(i'\) is matched to \(j' > j\). In the above example, this is \(M = \{(1, 1), (2, 2), (3, 3), (4, 5)\}\). Cost of an alignment is the number of mismatched columns plus number of unmatched indices in both strings.

4.5.0.4 Edit Distance Problem

Problem Given two words, find the edit distance between them, i.e., an alignment of smallest cost.

4.5.0.5 Applications

(A) Spell-checkers and Dictionaries

(B) Unix **diff**

(C) DNA sequence alignment ... but, we need a new metric

4.5.0.6 Similarity Metric

**Definition 4.5.3.** For two strings \(X\) and \(Y\), the cost of alignment \(M\) is

(A) **[Gap penalty]** For each gap in the alignment, we incur a cost \(\delta\).

(B) **[Mismatch cost]** For each pair \(p\) and \(q\) that have been matched in \(M\), we incur cost \(\alpha_{pq}\); typically \(\alpha_{pp} = 0\).

Edit distance is special case when \(\delta = \alpha_{pq} = 1\).
4.5.0.7 An Example

Example 4.5.4.

\[
\begin{array}{cccccccc}
\text{o} & \text{c} & \text{u} & \text{r} & \text{r} & \text{a} & \text{n} & \text{c} & \text{e} \\
\text{o} & \text{c} & \text{c} & \text{u} & \text{r} & \text{r} & \text{e} & \text{n} & \text{c} & \text{e}
\end{array}
\]

Cost = \delta + \alpha_{ae}

Alternative:

\[
\begin{array}{cccccccc}
\text{o} & \text{c} & \text{u} & \text{r} & \text{r} & \text{a} & \text{n} & \text{c} & \text{e} \\
\text{o} & \text{c} & \text{c} & \text{u} & \text{r} & \text{r} & \text{e} & \text{n} & \text{c} & \text{e}
\end{array}
\]

Cost = 3\delta

Or a really stupid solution (delete string, insert other string):

\[
\begin{array}{cccccccc}
\text{o} & \text{c} & \text{u} & \text{r} & \text{r} & \text{a} & \text{n} & \text{c} & \text{e} \\
\text{o} & \text{c} & \text{u} & \text{r} & \text{r} & \text{e} & \text{n} & \text{c} & \text{e}
\end{array}
\]

Cost = 19\delta.

4.5.0.8 Sequence Alignment

Problem 4.5.5 (Sequence Alignment).

Input: Given two words X and Y, and gap penalty \(\delta\) and mismatch costs \(\alpha_{pq}\).

Goal: Find alignment of minimum cost.

4.5.1 Edit distance

4.5.1.1 Basic observation

Let \(X = \alpha x\) and \(Y = \beta y\)

\(\alpha, \beta\): strings.

\(x\) and \(y\) single characters.

Think about optimal edit distance between \(X\) and \(Y\) as alignment, and consider last column of alignment of the two strings:

\[
\begin{array}{|c|c|}
\hline
\alpha & x \\
\beta & y \\
\hline
\end{array}
\]

or

\[
\begin{array}{|c|c|}
\hline
\alpha & x \\
\beta y & \\
\hline
\end{array}
\]

or

\[
\begin{array}{|c|c|}
\hline
\alpha x & \\
\beta & y \\
\hline
\end{array}
\]

Observation 4.5.6. Prefixes must have optimal alignment!

4.5.1.2 Problem Structure

Observation 4.5.7. Let \(X = x_1x_2\cdots x_m\) and \(Y = y_1y_2\cdots y_n\). If \((m, n)\) are not matched then either the \(m\)th position of \(X\) remains unmatched or the \(n\)th position of \(Y\) remains unmatched.

(A) Case \(x_m\) and \(y_n\) are matched.

   (A) Pay mismatch cost \(\alpha_{x_my_n}\) plus cost of aligning strings \(x_1\cdots x_{m-1}\) and \(y_1\cdots y_{n-1}\)

(B) Case \(x_m\) is unmatched.

   (A) Pay gap penalty plus cost of aligning \(x_1\cdots x_{m-1}\) and \(y_1\cdots y_n\)

(C) Case \(y_n\) is unmatched.

   (A) Pay gap penalty plus cost of aligning \(x_1\cdots x_m\) and \(y_1\cdots y_{n-1}\)
4.5.1.3 Subproblems and Recurrence

Optimal Costs Let \( \text{Opt}(i, j) \) be optimal cost of aligning \( x_1 \cdots x_i \) and \( y_1 \cdots y_j \). Then

\[
\text{Opt}(i, j) = \min \begin{cases} 
\alpha_{x_i, y_j} + \text{Opt}(i-1, j-1), \\
\delta + \text{Opt}(i-1, j), \\
\delta + \text{Opt}(i, j-1)
\end{cases}
\]

Base Cases: \( \text{Opt}(i, 0) = \delta \cdot i \) and \( \text{Opt}(0, j) = \delta \cdot j \)

4.5.1.4 Dynamic Programming Solution

\[
\begin{align*}
\text{for all } i & \text{ do } M[i, 0] = i\delta \\
\text{for all } j & \text{ do } M[0, j] = j\delta \\
\text{for } i = 1 \text{ to } m & \text{ do } \\
& \text{ for } j = 1 \text{ to } n \text{ do } \\
M[i, j] & = \min \begin{cases} 
\alpha_{x_i, y_j} + M[i-1, j-1], \\
\delta + M[i-1, j], \\
\delta + M[i, j-1]
\end{cases}
\end{align*}
\]

Analysis

(A) Running time is \( O(mn) \).
(B) Space used is \( O(mn) \).

4.5.1.5 Matrix and DAG of Computation

4.5.1.6 Sequence Alignment in Practice

(A) Typically the DNA sequences that are aligned are about \( 10^5 \) letters long!
(B) So about \( 10^{10} \) operations and \( 10^{10} \) bytes needed
(C) The killer is the 10GB storage
(D) Can we reduce space requirements?
4.5.1.7 Optimizing Space

(A) Recall
\[
M(i, j) = \min \begin{cases} 
\alpha_{x_i y_j} + M(i - 1, j - 1), \\
\delta + M(i - 1, j), \\
\delta + M(i, j - 1)
\end{cases}
\]

(B) Entries in \( j \)th column only depend on \((j - 1) \)st column and earlier entries in \( j \)th column
(C) Only store the current column and the previous column reusing space; \( N(i, 0) \) stores \( M(i, j - 1) \) and \( N(i, 1) \) stores \( M(i, j) \)

4.5.1.8 Computing in column order to save space

4.5.1.9 Space Efficient Algorithm

for all \( i \) do \( N[i, 0] = i \delta \)
for \( j = 1 \) to \( n \) do
\( N[0, 1] = j \delta \) (\(* \) corresponds to \( M(0, j) \) \(* \))
for \( i = 1 \) to \( m \) do
\[
N[i, 1] = \min \begin{cases} 
\alpha_{x_i y_j} + N[i - 1, 0] \\
\delta + N[i - 1, 1] \\
\delta + N[i, 0]
\end{cases}
\]
for \( i = 1 \) to \( m \) do
Copy \( N[i, 0] = N[i, 1] \)

Analysis Running time is \( O(mn) \) and space used is \( O(2m) = O(m) \)

4.5.1.10 Analyzing Space Efficiency

(A) From the \( m \times n \) matrix \( M \) we can construct the actual alignment (exercise)
(B) Matrix \( N \) computes cost of optimal alignment but no way to construct the actual alignment
(C) Space efficient computation of alignment? More complicated algorithm — see textbook.

4.5.1.11 Takeaway Points

(A) Dynamic programming is based on finding a recursive way to solve the problem. Need a recursion that generates a small number of subproblems.
(B) Given a recursive algorithm there is a natural **DAG** associated with the subproblems that are generated for given instance; this is the dependency graph. An iterative algorithm simply evaluates the subproblems in some topological sort of this **DAG**.

(C) The space required to evaluate the answer can be reduced in some cases by a careful examination of that dependency **DAG** of the subproblems and keeping only a subset of the **DAG** at any time.

### 4.6 All Pairs Shortest Paths

#### 4.6.0.1 Shortest Path Problems

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Given nodes $s, t$ find shortest path from $s$ to $t$.

(B) Given node $s$ find shortest path from $s$ to all other nodes.

(C) Find shortest paths for all pairs of nodes.

#### 4.6.0.2 Single-Source Shortest Paths

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Given nodes $s, t$ find shortest path from $s$ to $t$.

(B) Given node $s$ find shortest path from $s$ to all other nodes.

**Dijkstra’s algorithm** for non-negative edge lengths. Running time: $O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

**Bellman-Ford algorithm** for arbitrary edge lengths. Running time: $O(nm)$.

#### 4.6.0.3 All-Pairs Shortest Paths

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

(A) Find shortest paths for all pairs of nodes.

Apply single-source algorithms $n$ times, once for each vertex.

(A) Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.

(B) Arbitrary edge lengths: $O(n^2 m)$.

$\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?
4.6.0.4 Shortest Paths and Recursion

(A) Compute the shortest path distance from $s$ to $t$ recursively?
(B) What are the smaller sub-problems?

**Lemma 4.6.1.** Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

$(A)$ $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Sub-problem idea: paths of fewer hops/edges

4.6.0.5 Hop-based Recur’: Single-Source Shortest Paths

(A) Single-source problem: fix source $s$.
(B) $OPT(v, k)$: shortest path dist. from $s$ to $v$ using at most $k$ edges.
(C) Note: $dist(s, v) = OPT(v, n - 1)$.
(D) Recursion for $OPT(v, k)$:

$$OPT(v, k) = \min \left\{ \min_{u \in V}(OPT(u, k - 1) + c(u, v)), \right. \left. OPT(v, k - 1) \right\}$$

(E) Base case: $OPT(v, 1) = c(s, v)$ if $(s, v) \in E$ otherwise $\infty$
(F) Leads to Bellman-Ford algorithm — see text book.
(G) $OPT(v, k)$ values are also of independent interest: shortest paths with at most $k$ hops.

4.6.0.6 All-Pairs: Recursion on index of intermediate nodes

(A) Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$
(B) $dist(i, j, k)$: shortest path distance between $v_i$ and $v_j$ among all paths in which the largest index of an intermediate node is at most $k$
4.6.0.7 All-Pairs: Recursion on index of intermediate nodes

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
\]

\[
\text{dist}(i, j, k - 1) = \min \begin{cases} \\
\text{dist}(i, j, k - 1) \\
\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1) \\
\end{cases}
\]

Base case: \( \text{dist}(i, j, 0) = c(i, j) \) if \((i, j) \in E\), otherwise \(\infty\)

**Correctness:** If \(i \rightarrow j\) shortest path goes through \(k\) then \(k\) occurs only once on the path — otherwise there is a negative length cycle.
4.6.1 Floyd-Warshall Algorithm

4.6.1.1 for All-Pairs Shortest Paths

Check if $G$ has a negative cycle // Bellman-Ford: $O(mn)$ time
if $\exists$ negative cycle then return "$\text{Negative cycle}\"
for $i = 1$ to $n$ do
  for $j = 1$ to $n$ do
    $\text{dist}(i,j,0) = c(i,j)$ (* $c(i,j) = \infty$ if $(i,j) \notin E$, 0 if $i = j$ *)
for $k = 1$ to $n$ do
  for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
      $\text{dist}(i,j,k) = \min\left\{ \text{dist}(i,j,k-1), \text{dist}(i,k,k-1) + \text{dist}(k,j,k-1) \right\}$

Correctness: Recursion works under the assumption that all shortest paths are defined (no negative length cycle).

Running Time: $\Theta(n^3)$, Space: $\Theta(n^3)$.

4.6.2 Floyd-Warshall Algorithm

4.6.2.1 for All-Pairs Shortest Paths

Do we need a separate algorithm to check if there is negative cycle?

Correctness: exercise

4.6.2.2 Floyd-Warshall Algorithm: Finding the Paths

(A) Question: Can we find the paths in addition to the distances?
(B) Create a $n \times n$ array Next that stores the next vertex on shortest path for each pair of vertices
(C) With array Next, for any pair of given vertices $i, j$ can compute a shortest path in $O(n)$ time.
4.6.3 Floyd-Warshall Algorithm

4.6.3.1 Finding the Paths

```
for i = 1 to n do
    for j = 1 to n do
        d(i, j, 0) = c(i, j)  (* c(i, j) = ∞ if (i, j) not edge, 0 if i = j *)
        Next(i, j) = -1
    for k = 1 to n do
        for i = 1 to n do
            for j = 1 to n do
                if (d(i, j, k - 1) > d(i, k, k - 1) + d(k, j, k - 1)) then
                    d(i, j, k) = d(i, k, k - 1) + d(k, j, k - 1)
                    Next(i, j) = k
```

Exercise: Given Next array and any two vertices i, j describe an \( O(n) \) algorithm to find a i-j shortest path.

4.6.3.2 Summary of results on shortest paths

<table>
<thead>
<tr>
<th>Single vertex</th>
<th>( n ) * Dijkstra</th>
<th>( O(n \log n + m) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>No negative edges</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edges cost might be negative</td>
<td>Bellman Ford</td>
<td>( O(nm) )</td>
</tr>
<tr>
<td>But no negative cycles</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All Pairs Shortest Paths</th>
<th>( n ) * Bellman Ford</th>
<th>( O(n^2m) = O(n^4) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>No negative edges</td>
<td></td>
<td>( O(n^2 \log n + nm) )</td>
</tr>
<tr>
<td>No negative cycles</td>
<td>Floyd-Warshall</td>
<td>( O(n^3) )</td>
</tr>
</tbody>
</table>

4.7 Knapsack

4.7.0.1 Knapsack Problem

Problem 4.7.1 (Knapsack).

Input: Given a Knapsack of capacity \( W \) lbs. and \( n \) objects with \( i \)th object having weight \( w_i \) and value \( v_i \); assume \( W, w_i, v_i \) are all positive integers.

Goal: Fill the Knapsack without exceeding weight limit while maximizing value.

(A) Basic problem that arises in many applications as a sub-problem.

4.7.0.2 Knapsack Example

<table>
<thead>
<tr>
<th>Item</th>
<th>I₁</th>
<th>I₂</th>
<th>I₃</th>
<th>I₄</th>
<th>I₅</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>1</td>
<td>6</td>
<td>18</td>
<td>22</td>
<td>28</td>
</tr>
<tr>
<td>Weight</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

Example 4.7.2.

If \( W = 11 \), the best is \( \{I₃, I₄\} \) giving value 40.

Special Case When \( v_i = w_i \), the Knapsack problem is called the Subset Sum Problem.
4.7.0.3 Greedy Approach

(A) Pick objects with greatest value

(A) Let $W = 2$, $w_1 = w_2 = 1$, $w_3 = 2$, $v_1 = v_2 = 2$ and $v_3 = 3$; greedy strategy will pick \{3\}, but the optimal is \{1, 2\}

(B) Pick objects with smallest weight

(A) Let $W = 2$, $w_1 = 1$, $w_2 = 2$, $v_1 = 1$ and $v_2 = 3$; greedy strategy will pick \{1\}, but the optimal is \{2\}

(C) Pick objects with largest $v_i/w_i$ ratio

(A) Let $W = 4$, $w_1 = w_2 = 2$, $w_3 = 3$, $v_1 = v_2 = 3$ and $v_3 = 5$; greedy strategy will pick \{3\}, but the optimal is \{1, 2\}

(B) Can show that a slight modification always gives half the optimum profit: pick the better of the output of this algorithm and the largest value item. Also, the algorithms gives better approximations when all item weights are small when compared to $W$.

4.7.0.4 Towards a Recursive Solution

First guess: $\text{Opt}(i)$ is the optimum solution value for items $1, \ldots, i$.

Observation 4.7.3. Consider an optimal solution $\mathcal{O}$ for $1, \ldots, i$

Case item $i \notin \mathcal{O}$ $\mathcal{O}$ is an optimal solution to items $1$ to $i - 1$

Case item $i \in \mathcal{O}$ Then $\mathcal{O} - \{i\}$ is an optimum solution for items $1$ to $n - 1$ in knapsack of capacity $W - w_i$.

Subproblems depend also on remaining capacity. Cannot write subproblem only in terms of $\text{Opt}(1), \ldots, \text{Opt}(i - 1)$.

$\text{Opt}(i, w)$: optimum profit for items $1$ to $i$ in knapsack of size $w$

Goal: compute $\text{Opt}(n, W)$

4.7.0.5 Dynamic Programming Solution

Definition 4.7.4. Let $\text{Opt}(i, w)$ be the optimal way of picking items from $1$ to $i$, with total weight not exceeding $w$.

$$\text{Opt}(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ \text{Opt}(i - 1, w) & \text{if } w_i > w \\ \max \left\{ \begin{array}{l} \text{Opt}(i - 1, w) \\ \text{Opt}(i - 1, w - w_i) + v_i \end{array} \right. & \text{otherwise} \end{cases}$$

4.7.0.6 An Iterative Algorithm

```plaintext
for $w = 0$ to $W$ do
    $M[0, w] = 0$
for $i = 1$ to $n$ do
    for $w = 1$ to $W$ do
        if ($w_i > w$) then
            $M[i, w] = M[i - 1, w]$
        else
            $M[i, w] = \max(M[i - 1, w], M[i - 1, w - w_i] + v_i)$
```
Running Time

(A) Time taken is $O(nW)$
(B) Input has size $O(n + \log W + \sum_{i=1}^{n}(\log v_i + \log w_i))$; so running time not polynomial but “pseudo-polynomial”!

4.7.0.7 Knapsack Algorithm and Polynomial time

(A) Input size for Knapsack: $O(n) + \log W + \sum_{i=1}^{n}(\log w_i + \log v_i)$.
(B) Running time of dynamic programming algorithm: $O(nW)$.
(C) Not a polynomial time algorithm.
(D) Example: $W = 2^n$ and $w_i, v_i \in [1..2^n]$. Input size is $O(n^2)$, running time is $O(n2^n)$ arithmetic/comparisons.
(E) Algorithm is called a pseudo-polynomial time algorithm because running time is polynomial if numbers in input are of size polynomial in the combinatorial size of problem.
(F) Knapsack is NP-Hard if numbers are not polynomial in $n$.

4.8 Traveling Salesman Problem

4.8.0.1 Traveling Salesman Problem

Problem 4.8.1 (TSP).

Input: A graph $G = (V, E)$ with non-negative edge costs/lengths. Cost $c(e)$ for each edge $e \in E$.

Goal: Find a tour of minimum cost that visits each node.

(A) No polynomial time algorithm known. Problem is NP-Hard.
4.8.0.2 Drawings using TSP
4.8.0.3 Example: optimal tour for cities of a country (which one?)

4.8.0.4 An Exponential Time Algorithm

(A) How many different tours are there? \( n! \)

(B) Stirling’s formula: \( n! \approx \sqrt{n}(n/e)^n \) which is \( \Theta(2^{cn\log n}) \) for some constant \( c > 1 \)

(C) Can we do better? Can we get a \( 2^{O(n)} \) time algorithm?

4.8.0.5 Towards a Recursive Solution

(A) Order vertices as \( v_1, v_2, \ldots, v_n \)

(B) \( OPT(S) \): optimum TSP tour for the vertices \( S \subseteq V \) in the graph restricted to \( S \). Want \( OPT(V) \).

Can we compute \( OPT(S) \) recursively?

(A) Say \( v \in S \). What are the two neighbors of \( v \) in optimum tour in \( S \)?

(B) If \( u, w \) are neighbors of \( v \) in an optimum tour of \( S \) then removing \( v \) gives an optimum path from \( u \) to \( w \) visiting all nodes in \( S - \{v\} \).

Path from \( u \) to \( w \) is not a recursive subproblem! Need to find a more general problem to allow recursion.

4.8.0.6 A More General Problem: TSP Path

(A) Problem 4.8.2 (TSP Path).

Input: A graph \( G = (V, E) \) with non-negative edge costs/lengths \( c(e) \) for edge \( e \) and two nodes \( s, t \).
Goal: Find a path from $s$ to $t$ of minimum cost that visits each node exactly once.

(B) Can solve TSP using above. Do you see how?
(C) Recursion for optimum TSP Path problem:
(A) $OPT(u, v, S)$: optimum TSP Path from $u$ to $v$ in the graph restricted to $S$ (here $u, v \in S$).

4.8.1 A More General Problem: TSP Path

4.8.1.1 Continued...

(A) What is the next node in the optimum path from $u$ to $v$?
(B) Suppose it is $w$. Then what is $OPT(u, v, S)$?
(C) $OPT(u, v, S) = c(u, w) + OPT(w, v, S - \{u\})$
(D) We do not know $w$! So try all possibilities for $w$.

4.8.1.2 A Recursive Solution

(A) $OPT(u, v, S) = \min_{w \in S, w \neq u, v} \left( c(u, w) + OPT(w, v, S - \{u\}) \right)$
(B) What are the subproblems for the original problem $OPT(s, t, V)$?
$OPT(u, v, S)$ for $u, v \in S, S \subseteq V$.
(C) How many subproblems?
   (A) number of distinct subsets $S$ of $V$ is at most $2^n$
   (B) number of pairs of nodes in a set $S$ is at most $n^2$
   (C) hence number of subproblems is $O(n^2 2^n)$
(D) Exercise: Show that one can compute TSP using above dynamic program in $O(n^3 2^n)$ time and $O(n^2 2^n)$ space.

Disadvantage of dynamic programming solution: memory!

4.8.1.3 Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

(A) How to come up with the recursion?
(B) How to recognize that dynamic programming may apply?

4.8.1.4 Some Tips

(A) Problems where there is a natural linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.
(B) Problems involving trees: recursion based on subtrees.
(C) More generally:
   (A) Problem admits a natural recursive divide and conquer
   (B) If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
   (C) If optimal solution depends on all pieces then can apply dynamic programming if interface/interaction between pieces is limited. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.
4.8.1.5 Examples

(A) Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?

(B) Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?

(C) Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?

(D) Independent Set in a graph: break graph into two graphs. What is the interaction? Very high!

(E) Knapsack: Split items into two sets of half each. What is the interaction?