Algorithm Design 2 Lecture Notes - Dynamic Programming

Ms. Aditi Panda

Department of Computer Science and Engineering,
Institute of Technical Education & Research,
Siksha 'O' Anusandhan University, Bhubaneswar

October 25, 2021

Contents

1 Dynamic Programming

- The technique was developed by Richard Bellman in the 1950s for optimization problems.
- Dynamic programming (DP) efficiently implements a recursive algorithm by storing partial results.
- DP algorithm solves each subproblem just once and then remembers its answer, thereby avoiding re-computation of the answer for similar subproblem every time.

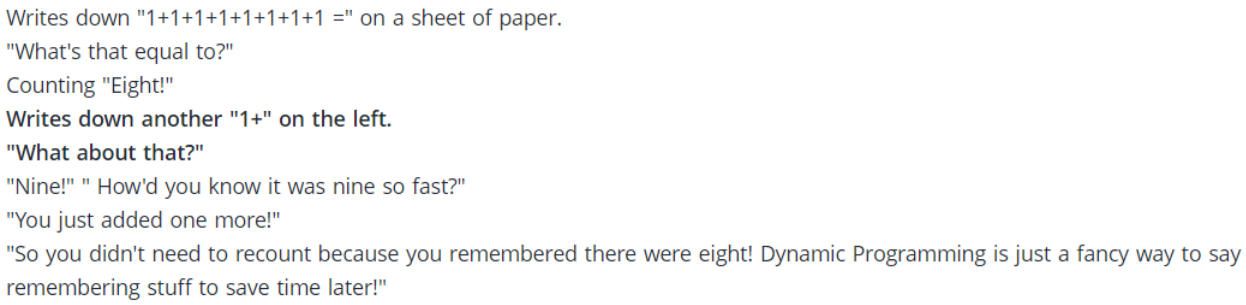

Figure 1: Dynamic Programming Scenario 1

1.1 Dynamic Programming and Recursion

- Dynamic programming is basically, recursion plus using common sense.
- What it means is that recursion allows you to express the value of a function in terms of other values of that function.
- Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster.
- This is what we call **Memoization** - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

- The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

2 Differences between dynamic programming and other approaches

- Greedy algorithms that make the best local decision at each step are typically efficient but usually do not guarantee global optimality.
- Exhaustive search algorithms that try all possibilities and select the best always produce the optimum result, but usually at a prohibitive cost in terms of time complexity.
- Dynamic programming combines the best of both worlds. It gives us a way to design custom algorithms that systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).
- Divide-and-conquer algorithms partition the problem into disjoint subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
- In contrast, dynamic programming applies when the subproblems overlap—that is, when subproblems share **subsubproblems**. In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common **subsubproblems**.
- The main difference between divide and conquer and dynamic programming is that the divide and conquer combines the solutions of the sub-problems to obtain the solution of the main problem while dynamic programming uses the result of the sub-problems to find the optimum solution of the main problem.
    - Start with a recursive algorithm or definition. Only once we have a correct recursive algorithm do we worry about speeding it up by using a results matrix.

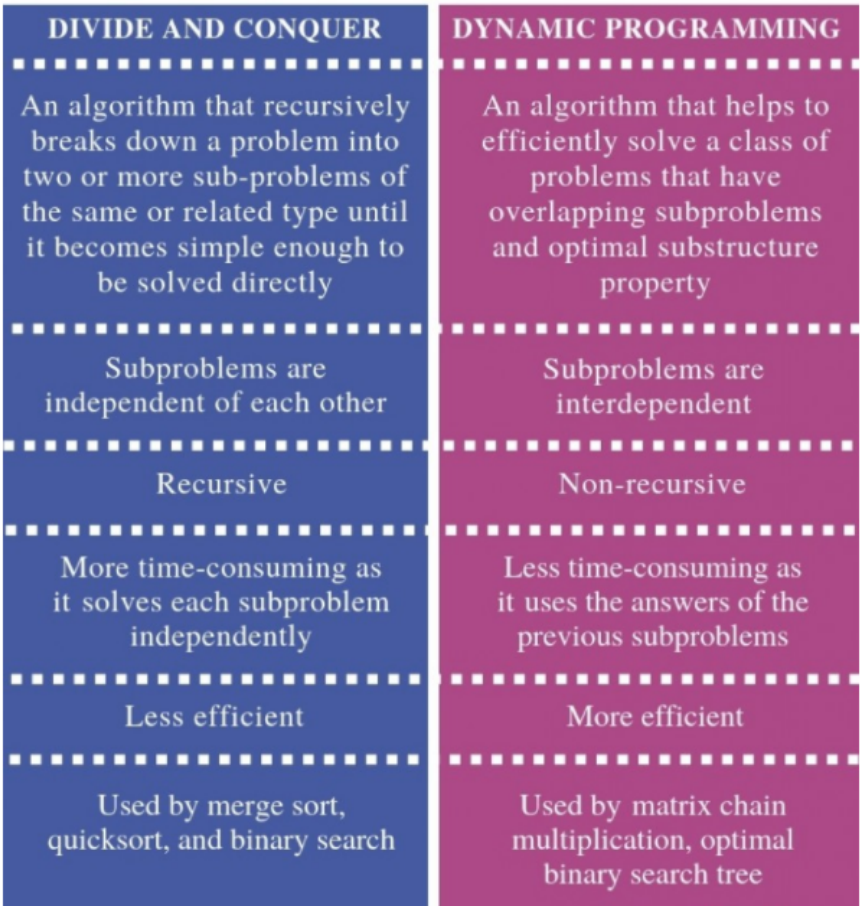

Figure 2: Divide and Conquer vs Dynamic Programming

3 Ordered data

- Dynamic programming is generally the right method for optimization problems on combinatorial objects that have an inherent left to right order among components.
- Left-to-right objects includes: character strings, rooted trees, polygons, and integer sequences.

4 How to use DP?

A sequence of four steps is used:

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from computed information

5 Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2} \tag{1}$$

5.1 Fibonacci Numbers by Recursion

```
long fib_r(int n)
{
        if (n == 0) return(0);
        if (n == 1) return(1);

        return(fib_r(n-1) + fib_r(n-2));
}
```

Figure 3: Recursive Code for Generating Fibonacci numbers

How much time does this algorithm take to compute F(n)?
Fibonacci numbers are related to the golden ratio ϕ and to its conjugate $\hat{\phi}$, which are the two roots of the equation:

$$x^2 = x + 1 \tag{2}$$

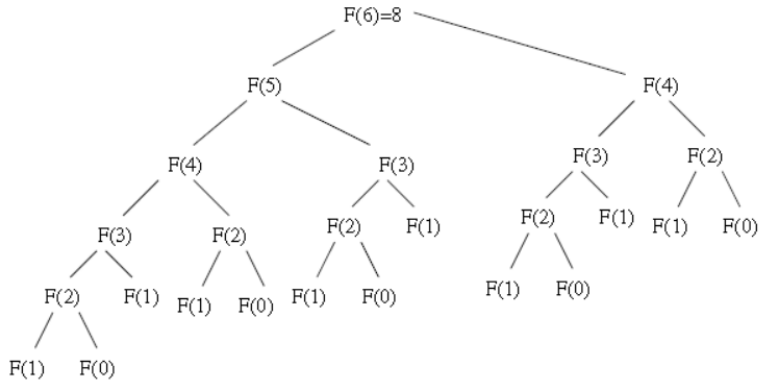

Figure 4: The computation tree for computing Fibonacci numbers recursively

and are given by the following formulas:

$$\phi = \frac{1 + \sqrt{(5)}}{2} = 1.61803, ... \tag{3}$$

and

$$\hat{\phi} = \frac{1 - \sqrt{(5)}}{2} = -0.61803, ... \tag{4}$$

When we take any two successive (one after the other) Fibonacci Numbers, their ratio is very close to the Golden Ratio (Fig. 7).

So we have

$$\frac{F_n}{F_{n-1}} = 1.6 \tag{5}$$

Specifically, we have:

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}} \tag{6}$$

Since $|\hat{\phi}| < 1$,

$$\frac{|\hat{\phi}^i|}{\sqrt{5}} < \frac{1}{\sqrt{5}} \tag{7}$$

which implies that

$$F_i = \lfloor \frac{\phi^i}{\sqrt{5}} + \frac{1}{2} \rfloor \tag{8}$$

A	B	B / A
2	3	1.5
3	5	1.666666666...
5	8	1.6
8	13	1.625
13	21	1.615384615...
...	...	...
144	233	1.618055556...
233	377	1.618025751...
...	...	...

Figure 5: Relationship between Fibonacci numbers and Golden ratio

which is to say that the i_th Fibonacci number F_i is $\frac{\phi^i}{\sqrt{5}}$ rounded to the nearest integer. Thus, Fibonacci numbers grow exponentially. Since, $\mathbf{F_n > 1.6^n}$, we must have at least 1.6^n leaves or procedure calls! This humble little program takes exponential time to run!

5.2 Fibonacci Numbers by Caching

The key to avoiding recomputation is to explicitly check for the value before trying to compute it:

```
#define MAXN    45        /* largest interesting n */
#define UNKNOWN   -1      /* contents denote an empty cell */
long f[MAXN+1];           /* array for caching computed fib values */
```

Figure 6: Caching Fibonacci Values

It computes F(n) in linear time (in other words, O(n) time) because the recursive function fib c(k) is called exactly twice for each value $0 \leq k \leq n$.

```
long fib_c(int n)
{
        if (f[n] == UNKNOWN)
                f[n] = fib_c(n-1) + fib_c(n-2);

        return(f[n]);
}


long fib_c_driver(int n)
{
        int i;                  /* counter */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = UNKNOWN;

        return(fib_c(n));
}
```

Figure 7: Code to cache Fibonacci Values

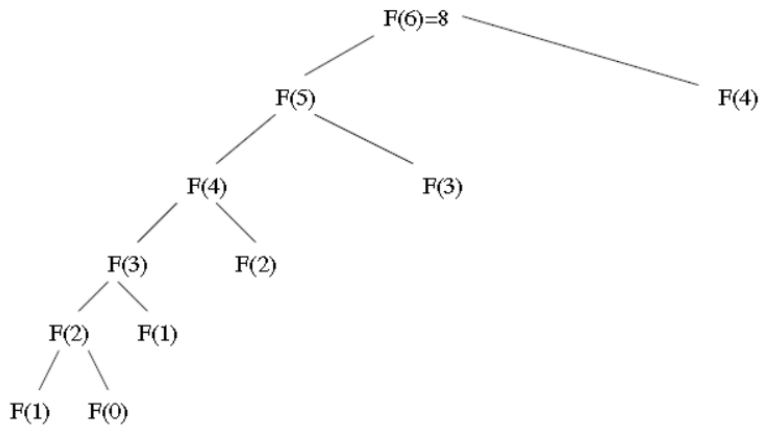

Figure 8: The computation tree for computing Fibonacci numbers using caching

5.3 Fibonacci Numbers by Dynamic Programming

We can calculate F_n in linear time more easily by explicitly specifying the order of evaluation of the recurrence relation. Each of the n values is computed as the simple sum of two integers in total O(n) time and space.

```
long fib_dp(int n)
{
        int i;              /* counter */
        long f[MAXN+1];     /* array to cache computed fib values */

        f[0] = 0;
        f[1] = 1;
        for (i=2; i<=n; i++)  f[i] = f[i-1]+f[i-2];

        return(f[n]);
}
```

Figure 9: Code to compute Fibonacci numbers by dynamic programming

More careful study shows that we do not need to store all the intermediate values for the entire period of execution. Because the recurrence depends on two arguments, we only need to retain the last two values we have seen:

```
long fib_ultimate(int n)
{
        int i;                  /* counter */
        long back2=0, back1=1;  /* last two values of f[n] */
        long next;              /* placeholder for sum */

        if (n == 0) return (0);

        for (i=2; i<n; i++) {
                next = back1+back2;
                back2 = back1;
                back1 = next;
        }
        return(back1+back2);
}
```

Figure 10: Code to further reduce storage demands to constant space for computing Fibonacci numbers by dynamic programming

6 Binomial Coefficients

Another illustration of how to eliminate recursion by specifying the order of evaluation. You can compute them straight from factorials. However, this method has a serious drawback.

Intermediate calculations can easily cause arithmetic overflow, even when the final coefficient fits comfortably within an integer. Next best method: using pascal's triangle.

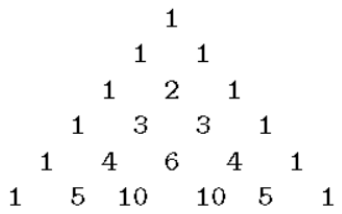

Figure 11: Pascal's Triangle

Each number is the sum of the two numbers directly above it. The recurrence relation implicit in this is that

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Figure 12: Recurrence of Binomial Coefficients

The best way to evaluate such a recurrence is to build a table of possible values up to the size that you are interested in.

m / n	0	1	2	3	4	5
0	A					
1	B	G				
2	C	1	H			
3	D	2	3	I		
4	E	4	5	6	J	
5	F	7	8	9	10	K

m / n	0	1	2	3	4	5
0	1					
1	1	1				
2	1	1	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	10	10	**5**	1

Figure 13: Evaluation order for binomial coefficient at M[5, 4] (l). Initialization conditions A-K, recurrence evaluations 1-10. Matrix contents after evaluation (r)

```
long binomial_coefficient(n,m)
int n,m;                          /* computer n choose m */
{
        int i,j;                  /* counters */
        long bc[MAXN][MAXN];      /* table of binomial coefficients */

        for (i=0; i<=n; i++) bc[i][0] = 1;

        for (j=0; j<=n; j++) bc[j][j] = 1;

        for (i=1; i<=n; i++)
                for (j=1; j<i; j++)
                        bc[i][j] = bc[i-1][j-1] + bc[i-1][j];

        return( bc[n][m] );
}
```

Figure 14: Code for Binomial Coefficients

7 Parsing Context-Free Grammars

sentence ::= noun–phrase
verb–phrase
noun–phrase ::= article noun
verb–phrase ::= verb noun–phrase
article ::= *the, a*
noun ::= *cat, milk*
verb ::= *drank*

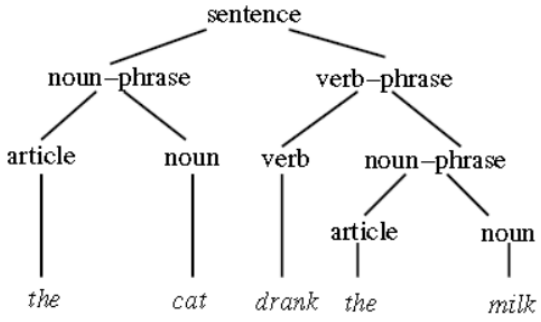

Figure 15: A context-free grammar (l) with an associated parse tree (r)

- Compilers identify whether the given program is legal in the programming language, and reward you with syntax errors if not.
- This requires a precise description of the language syntax typically given by a context-free grammar.
- Each rule or production of the grammar defines an interpretation for the named symbol on the left side of the rule as a sequence of symbols on the right side of the rule.
- The right side can be a combination of nonterminals (themselves defined by rules) or terminal symbols defined simply as strings, such as *the*, *a*, *cat*, *milk*, and *drank*.
- Parsing a given text string S according to a given context-free grammar G is the algorithmic problem of constructing a parse tree of rule substitutions defining S as a single nonterminal symbol of G.
- We assume that the text string S has length n while the grammar G itself is of constant size. This is fair, since the grammar defining a particular programming language (say C or Java) is of fixed length regardless of the size of the program we are trying to compile.
- We assume that the definitions of each rule are in Chomsky normal form. This means that the right sides of every nontrivial rule consists

$$M[i,j,X] = \bigvee_{(X \to YZ) \in G} \left(\bigvee_{i=k}^{j} M[i,k,Y] \cdot M[k+1,j,Z] \right)$$

Figure 16: DP solution for Parsing using CFG

of (a) exactly two nonterminals, i.e. $X->YZ$, or (b) exactly one terminal symbol, $X->\alpha$.

- So how can we efficiently parse a string S using a context-free grammar where each interesting rule consists of two nonterminals?
- The key observation is that the rule applied at the root of the parse tree (say $X->YZ$) splits S at some position i such that the left part of the string (S[1, i]) must be generated by nonterminal Y , and the right part (S[i + 1, n]) generated by Z.
- A dynamic programming solution: Define M[i, j,X] to be a boolean function that is true iff substring S[i, j] is generated by nonterminal X. This is true if there exists a production $X->YZ$ and breaking point k between i and j such that the left part generates Y and the right part Z.

- The size of our state-space is $O(n^2)$, as there are $n(n+1)/2$ substrings defined by (i, j) pairs. Multiplying this by the number of nonterminals (say v) has no impact on the big-Oh, because the grammar was defined to be of constant size.
- Evaluating the value M[i, j,X] requires testing all intermediate values k, so it takes $O(n)$ in the worst case to evaluate each of the $O(n^2)$ cells. This yields an $O(n^3)$ or cubic-time algorithm for parsing.

8 The Floyd-Warshall Algorithm

- All-pairs shortest-paths problem on a directed graph G = (V, E)
- Assumption : Negative-weight edges may be present, but we assume that there are no negative-weight cycles.

- The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $p =< v_1, v_2, ..., v_l >$ is any vertex of p other than v_1 or v_l , that is, any vertex in the set $v_2, v_3, ..., v_{l-1}$

- The vertices of G are $V = 1, 2, ..., n$, let us consider a subset $1, 2, ..., k$ of vertices for some k.

- For any pair of vertices $i, j \subset V$, consider all paths from i to j whose intermediate vertices are all drawn from $1, 2, ..., k$, and let p be a minimum-weight (simple) path from among them.

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set $1, 2, ..., k-1$. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set $1, 2, ..., k-1$ is also a shortest path from i to j with all intermediate vertices in the set $1, 2, ..., k$.

- If k is an intermediate vertex of path p, then we decompose p into $i \xrightarrow{\text{p1}} k \xrightarrow{\text{p2}} j$.

- Because vertex k is not an intermediate vertex of path $p1$, all intermediate vertices of $p1$ are in the set $1, 2, ..., k-1$.

8.1 A recursive solution to the all-pairs shortest-paths problem

- Let $d_{i,j}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set $1, 2, ..., k$.

- When $k = 0$, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all.

- Such a path has at most one edge, and hence $d_{i,j}^{(0)} = w_{i,j}$.

Because for any path, all intermediate vertices are in the set $1, 2, ..., n$, the matrix $D^{(n)} = (d_{i,j}^{(n)})$ gives the final answer: $d_{i,j}^{(n)} = \delta_{i,j}$ for all $i, j \subset V$.

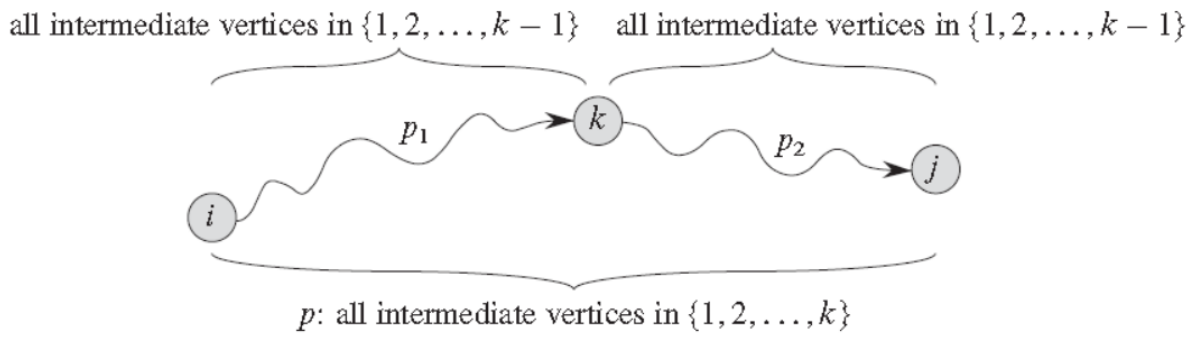

Figure 17: Path p is a shortest path from vertex i to vertex j, and k is the highest-numbered intermediate vertex of p. Path $p1$, the portion of path p from vertex i to vertex k, has all intermediate vertices in the set $1, 2, ..., k-1$. The path $p2$ from vertex k to vertex j also has all intermediate vertices in the set $1, 2, ..., k-1$.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0\,, \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1\,. \end{cases}$$

Figure 18: Recursive formulation of shortest path using Floyd-Warshall Algorithm

8.2 Computing the shortest-path weights bottom up

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4       let D^(k) = (d_ij^(k)) be a new n × n matrix
5       for i = 1 to n
6            for j = 1 to n
7                 d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

Figure 19: Pseudo-code of Floyd-Warshall Algorithm, running time: $\theta(n^3)$

8.3 Constructing a shortest path - Predecessor matrix

When $k = 0$, a shortest path from i to j has no intermediate vertices at all.

$$\pi_{ij}^{(0)} = \begin{cases} \text{NIL} & \text{if } i = j \text{ or } w_{ij} = \infty \,, \\ i & \text{if } i \neq j \text{ and } w_{ij} < \infty \,. \end{cases}$$

Figure 20: Predecessor matrix computation for $k = 0$

For $k \geq 1$, if we take the path $i- > k- > j$, where $k! = j$, then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set $1, 2, ..., k-1$. Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set $1, 2, ..., k-1$. For $k \geq 1$,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \,, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \,. \end{cases}$$

Figure 21: Predecessor matrix computation for $k \geq 1$

How to include this computation in code?

```
MOD-FLOYD-WARSHALL(W)
    n = W.rows
    D(0) = W
    let π(0) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            if i != j and D[i, j](0) < ∞
                π[i, j](0) = i
    for k = 1 to n
        let D(k) be a new n × n matrix
        let π(k) be a new n × n matrix
        for i = 1 to n
            for j = 1 to n
                if d[i, j](k - 1) ≤ d[i, k](k - 1) + d[k, j](k - 1)
                    d[i, j](k) = d[i, j](k - 1)
                    π[i, j](k) = π[i, j](k - 1)
                else
                    d[i, j](k) = d[i, k](k - 1) + d[k, j](k - 1)
                    π[i, j](k) = π[k, j](k - 1)
```

Figure 22: Computing Predecessor matrix in Floyd-Warshall Algorithm along with $D^{(n)}$ matrix

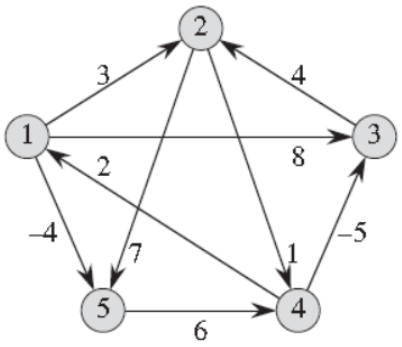

Figure 23: Example Graph

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 24: Example Execution of Floyd-Warshall algorithm

9 The Partition Problem

10 Minimum Weight Triangulation

- A triangulation of a polygon P = v_1, . . . , v_n, v_1 is a set of nonintersecting diagonals that partitions the polygon into triangles.
- The weight of a triangulation is the sum of the lengths of its diagonals.
- Any given polygon may have many different triangulations.

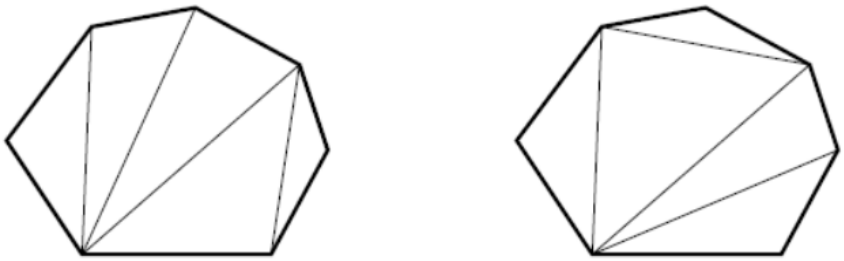

Figure 25: Example Triangulations of the Same polygon

- We seek to find its **minimum weight triangulation** for a given polygon p.
- To apply dynamic programming, we need a way to carve up the polygon into smaller pieces.
- Every edge of the input polygon must be involved in exactly one triangle.
- Turning this edge into a triangle means identifying the third vertex.
- Once we find the correct connecting vertex, the polygon will be partitioned into two smaller pieces, both of which need to be triangulated optimally.
- Let $T[i, j]$ be the cost of triangulating from vertex v_i to vertex v_j , ignoring the length of the chord d_{ij} from v_i to v_j .

- The basis condition applies when i and j are immediate neighbors, as $T[i, i+1] = 0$.

$$T[i,j] \quad = \quad \min_{k=i+1}^{j-1} (T[i,k] + T[k,j] + d_{ik} + d_{kj})$$

Figure 26: Recurrence of Min Weight Triangulations of polygons

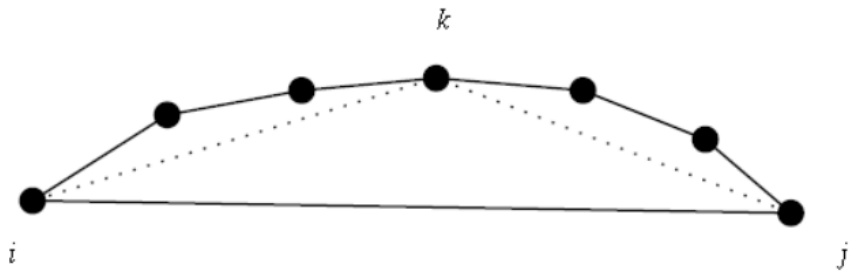

Figure 27: Selecting the vertex k to pair with an edge (i, j) of the polygon

Minimum-Weight-Triangulation(P)
    for $i = 1$ to $n-1$ do $T[i, i+1] = 0$
    for $gap = 2$ to $n-1$
        for $i = 1$ to $n - gap$ do
            $j = i + gap$
            $T[i,j] = \min_{k=i+1}^{j-1}(T[i,k] + T[k,j] + d_{P_i,P_k} + d_{P_k,P_j})$
    return $T[1,n]$

Figure 28: Code of Min Weight Triangulations of polygons

There are $\binom{n}{2}$ values of T, each of which takes $O(j-i)$ time if we evaluate the sections in order of increasing size. Since $j - i = O(n)$, complete evaluation takes $O(n^3)$ time and $O(n^2)$ space.

Figure 29: Time complexity of Min Weight Triangulations of polygons

11 Limitations of Dynamic Programming