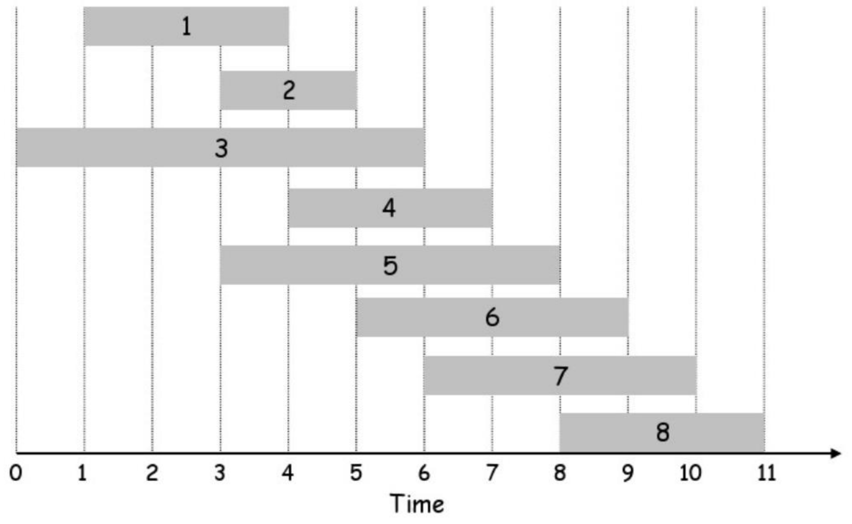
Dynamic programming 2

Outline

- Searching
  - Weighted Interval Scheduling/Maximum Profit in Job Scheduling
- Interval
  - Longest Palindromic Subsequence
  - Minimum Score Triangulation of Polygon
- State and Bitmask
  - Best Time to Buy and Sell Stock
  - Travelling Salesman Problem
  - Minimum XOR Sum of Two Arrays
  - Numbers With Repeated Digits

Weighted Interval-Scheduling (WIS)

- Problem:
  - Job j starts at s[j], finishes at f[j], and has weight/value v[j].
  - Two jobs **compatible** if they **don't overlap**.
  - Goal: find the **maximum weight** subset of mutually compatible jobs.
  - p[j]: largest index i < j such that job i is compatible with job j.
- Example:
  - Similar to knapsack?

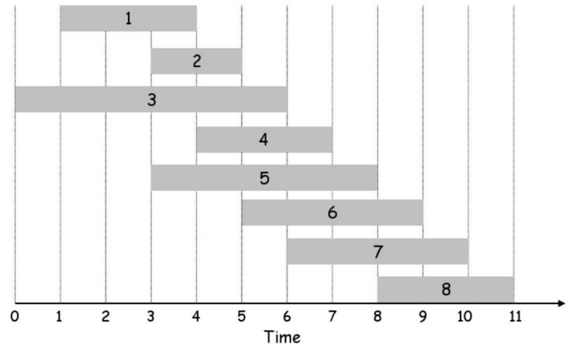


j	v[j]	p[j]
1	3	0
2	1	0
3	6	0
4	5	1
5	1	0
6	2	2
7	4	3
8	2	5

WIS - Optimal substructure

- Subproblems: prefix, dp[j]: maximum weight sum of mutually compatible jobs in first j jobs ([0,j]), jobs are sorted by f[j] (finishing time)
- Relate
  - Choose: v[j] + dp[p[j]]
    - Binary search to get p[j], the previous possible job
  - Not choose: 0 + dp[j – 1]
  - dp[j] = max(v[j] + dp[p[j]], dp[j – 1])
  - Example
    - dp[1] = max(3 + dp[0], dp[1 - 1]) = 3
    - dp[2] = max(1 + dp[0], dp[2 - 1]) = 3
    - dp[3] = max(6 + dp[0], dp[3 - 1]) = 6
    - dp[4] = max(5 + dp[1], dp[4 - 1]) = 8
    - …
    - dp[7] = max(4 + dp[3], dp[7 - 1]) = 10
    - dp[8] = max(2 + dp[5], dp[8 - 1]) = 10
- Topological order: increasing j
- Base case: dp[0] = 0
- Time analysis: sorting: O(nlogn), n times binary search: O(nlogn), thus total: O(nlogn)

0	1	2	3	4	5	6	7	8
0	3	3	6	8	8	8	10	10



j	v[j]	p[j]
1	3	0
2	1	0
3	6	0
4	5	1
5	1	0
6	2	2
7	4	3
8	2	5

WIS/Maximum Profit in Job Scheduling

```
int jobScheduling(vector<int>& startTime, vector<int>& endTime, vector<int>& profit) {
        // Compare based on endTime
        auto compareFinishTime = [](const vector<int>& job1, const vector<int>& job2) {return job1[1] < job2[1];};
        int n = startTime.size();
        vector<vector<int>> jobs(n);
        for (int i = 0; i < n; i++) jobs[i] = {startTime[i], endTime[i], profit[i]};
        sort(jobs.begin(), jobs.end(), compareFinishTime); // sort by finish time
        vector<int> dp(n + 1);
        for (int i = 1; i <= n; i++) {
            // to find by current job’s start time, dummy job
            vector<int> toFind = {0, jobs[i - 1][0], 0};
            // binary search to find the closest previous compatible job
            int p = upper_bound(jobs.begin(), jobs.begin() + i - 1, toFind, compareFinishTime) - jobs.begin();
            dp[i] = max(jobs[i - 1][2] + dp[p], dp[i - 1]);
        }
        return dp[n];
    }
```

https://leetcode.com/problems/maximum-profit-in-job-scheduling/

Interval

- Subproblems:
  - Subproblems are typically represented as **intervals** rather than prefixes or suffixes.
  - Solving and combining smaller interval-based subproblems progressively.
  - Usually in this format: dp[i,j]: the value in the index range [i,j].
- Relate:
  - Choose/Not Choose: gradually reducing the problem from **both ends**
    - Longest Palindromic Subsequence
  - Enumerate positions to divide: **subdivide** into smaller subproblems
    - Minimum Score Triangulation of Polygon
- Topological order: smaller/shorter subproblems/intervals to bigger/longer ones, need to consider the iteration order carefully

Longest Palindromic Subsequence

- Problem
  - Given a string s, find the longest palindromic subsequence's length in s.
  - Input: s = "bbbab"
  - Output: 4
  - Explanation: One possible longest palindromic subsequence is "bbbb“
  - Observation: For a given subsequence, if it is a **palindromic** subsequence and its length is greater than 2, then removing its first and last characters will still result in a palindromic subsequence.
  - Another approach: a palindrome reads the same forwards and backwards, thus, find the longest common subsequence (LCS) between the original string and its reversed version.

https://leetcode.com/problems/longest-palindromic-subsequence/

Longest Palindromic Subsequence

- Subproblems:
  - Interval i to j: the substring of s with index range [i,j]
  - dp[i][j]: the length of the longest palindromic subsequence **within** the interval i to j.
- Relate:
  - Choose/not choose, similar to LCS:
    - Choose one of s[i] and s[j]: max(dp[i+1][j], dp[i][j-1])
    - Choose s[i] and s[j]: dp[i+1][j-1] + 2
    - Not choose s[i] and s[j]: dp[i+1][j-1]
  - s[i] == s[j]
    - Find the longest palindromic subsequence in the range [i+1, j-1] and then add s[i] and s[j] to the beginning and end of it
    - dp[i][j] = dp[i+1][j-1] + 2
  - s[i] != s[j]
    - s[i] and s[j] cannot both be the endpoints of the same palindromic subsequence
    - dp[i][j] = max(dp[i+1][j], dp[i][j-1])
- Topological order: shorter to longer, decreasing i, increasing j
- Base cases:
  - Any subsequence of length 1 is a palindromic subsequence, dp[i][i]=1 or dp[i][j]=1 for i==j
  - No character: dp[i+1][i]=0, e.g., “bb”: dp[i][i+1]-> dp[i+1][i], or dp[i][j]=0 for i>j
- Original problem: dp[0][n–1]
- Time analysis: O(n*n)

Longest Palindromic Subsequence

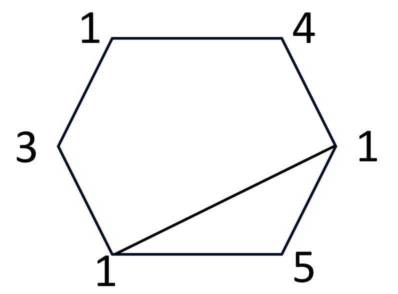
```
int longestPalindromeSubseq(string s) {
    int n = s.length();
    vector<vector<int>> dp(n, vector<int>(n));
    for (int i = n - 1; i >= 0; i--) {
        dp[i][i] = 1;
        for (int j = i + 1; j < n; j++) {
            if(s[i] == s[j])  dp[i][j] = dp[i + 1][j - 1] + 2;
            else dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
        }
    }
    return dp[0][n - 1];
}
```

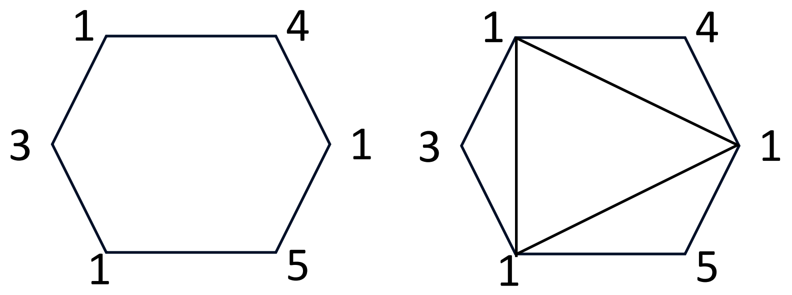
```
int longestPalindromeSubseq(string s) {
    int n = s.length();
    vector<vector<int>> memo(n, vector<int>(n, -1));
    return dfs(0, n - 1, memo, s);
}
int dfs (int i, int j, vector<vector<int>>& memo, string& s) {
    if (i > j) return 0; // empty string
    if (i == j) return 1; // only one char
    int& result = memo[i][j]; // reference, for updating memo
    if (result != -1) return result; // reuse
    if (s[i] == s[j]) result = dfs(i + 1, j - 1, memo, s) + 2; // choose both
    else result = max(dfs(i + 1, j, memo, s), dfs(i, j - 1, memo, s)); // max of choosing one

    return result;
}
```

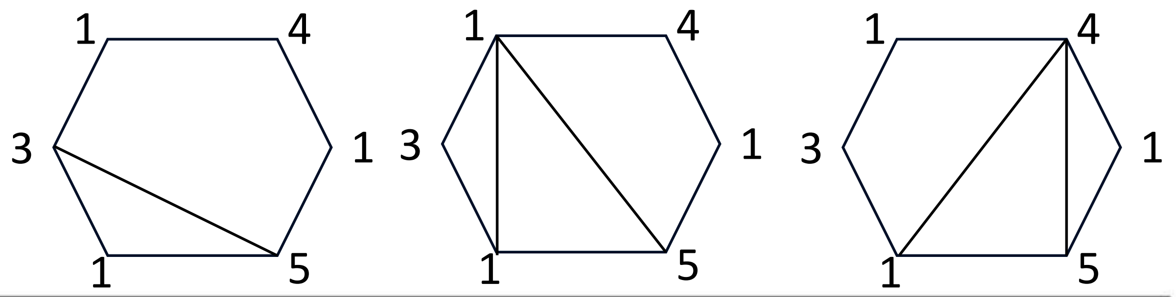
Minimum Score Triangulation of Polygon

https://leetcode.com/problems/minimum-score-triangulation-of-polygon/

- Problem:
  - You have a convex n-sided polygon where each vertex has an integer value. You are given an integer array values where values[i] is the value of the ith vertex in clockwise order.
  - Polygon triangulation is a process where you divide a polygon into a set of triangles and the vertices of each triangle must also be vertices of the original polygon. Note that no other shapes other than triangles are allowed in the division. This process will result in n - 2 triangles.
  - You will triangulate the polygon. For each triangle, the weight of that triangle is the product of the values at its vertices. The total score of the triangulation is the sum of these weights over all n - 2 triangles.
  - Return the minimum possible score that you can achieve with some triangulation of the polygon.
  - Input: values = [3,1,4,1,5,1]
  - Output: 1*3*1+1*4*1+1*5*1+1*1*1 = 13
  - Idea: there must be one triangle contains edge 1-5, enumerate the other vertex (of this triangle)





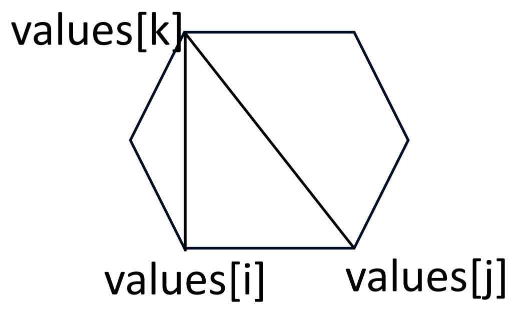


Minimum Score Triangulation of Polygon

- Subproblems:
  - Interval i to j: the subpolygon formed with the polygon edges clockwise from vertex i to vertex j plus the edge j-i
  - dp[i][j]: the minimum score obtained from the interval i to j
- Relate:
  - Minimum score through enumerating k
    - minimum score from i to k: dp[i][k]
    - minimum score from k to j: dp[i][k]
    - score of i-k-j

```
  for (int k = i + 1; k < j; k++)
      dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j] + values[i] * values[j] * values[k]);
```

- Topological order:
  - i<k, dp[i][j] needs dp[k][j] (conceptually, dp[i+x][j]), thus, decreasing i
  - j>k, dp[i][j] needs dp[i][k] (conceptually, dp[i][j-x]), thus, increasing j
- Base cases: dp[i][j] = INT_MAX (dp[i][i+1] = 0)
- Original problem: dp[0][n–1]
- Time analysis: O(n*n*n)



Minimum Score Triangulation of Polygon

```
int minScoreTriangulation(vector<int>& values) {
    int n = values.size();
    vector<vector<int>> memo(n, vector<int>(n, -1)); // -1: not computed
    return dfs(0, n - 1, memo, values);
}
int dfs(int i, int j, vector<vector<int>>& memo, vector<int>& values) {
    if (i + 1 == j)  return 0; // 2 points, cannot form a tri
    int &result = memo[i][j]; // reference, for updating memo
    if (result != -1) return result; // reuse
    result = INT_MAX;
    for (int k = i + 1; k < j; k++) { // enumerate vertex k
        result = min(result, dfs(i, k, memo, values) + dfs(k, j, memo,
values) + values[i] * values[j] * values[k]);
    }
    return result;
}
```

```
int minScoreTriangulation(vector<int>& values) {
    int n = values.size();
    vector<vector<int>> dp(n, vector<int>(n));
    for (int i = n - 3; i >= 0; i--) {
        for (int j = i + 2; j < n; j++) {
            dp[i][j] = INT_MAX;
            for (int k = i + 1; k < j; k++) {
                dp[i][j] = min(dp[i][j], dp[i][k]
+ dp[k][j] + values[i] * values[j] * values[k]);
            }
        }
    }
    return dp[0][n - 1];
}
```

Summary

- Searching
  - The current state may transit to one of the previous state, searching/sorting maybe needed.
- Interval
  - Subproblems are typically represented as **intervals**
  - Choose/Not Choose: gradually reducing the problem from **both ends**
  - Enumerate positions to divide: **subdivide** into smaller subproblems
  - Exercises:
    - https://leetcode.com/problems/minimum-cost-to-cut-a-stick/
    - https://leetcode.com/problems/minimum-cost-to-merge-stones/
    - https://leetcode.com/problems/burst-balloons/

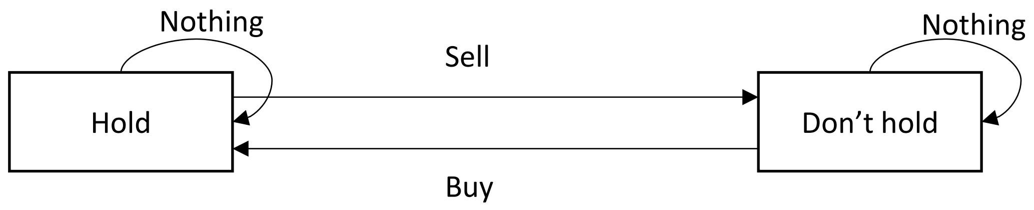
Best Time to Buy and Sell Stock II

- Problem:
  - You are given an integer array prices where prices[i] is the price of a given stock on the ith day.
  - On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.
  - Find and return the maximum profit you can achieve.
  - Input: prices = [7,1,5,3,6,4]
  - Output: 7
  - Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.
  - Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
  - Total profit is 4 + 3 = 7.

https://leetcode.com/problems/best-time-to-buy-and-sell-stock-ii/

Best Time to Buy and Sell Stock II

- Idea: what happens at the last day?
  - (prefix) The profit from day 0 to 5 = profit from day 0 to 4 + the profit on day 5
  - Two factors: days, hold/don’t hold the stock,
  - Consider what is the last transection
    - Buy: profit -= stock price
    - Sell: profit += stock price
    - Nothing: no change



- Subproblems: dp[i,j]: **after** day i the max profit when hold/don’t hold the stock (j=1/0). Note that, **after** day **i-1** is the **start** of day **i**
- Relate
  - Buy, don’t hold on day i-1, hold on day i: dp[i,1] = dp[i-1,0] - prices[i]
  - Sell, hold on day i-1, don’t hold on day i: dp[i,0] = dp[i-1,1] + prices[i]
  - Nothing: dp[i,0] = dp[i-1,0], dp[i,1] = dp[i-1,1]
  - dp[i,0] = max(dp[i-1,0], dp[i-1,1] + prices[i])
  - dp[i,1] = max(dp[i-1,1], dp[i-1,0] - prices[i])
- Topological order: increasing i
- Base cases: -1 is for illustrative purpose
  - dp[-1][0] = 0, at the **start** of day 0, don’t hold, profit 0
  - dp[-1][1] = INT_MIN, at the **start** of day 0, it is not possible to hold
- Original problem: max(dp[n-1,0], dp[n-1,1]) = dp[n-1][0], don’t hold/sell it definitely gives more profit
- Time analysis: O(n)

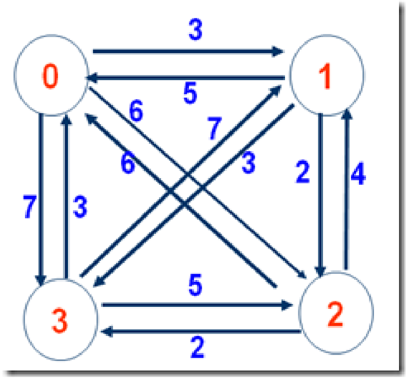
Best Time to Buy and Sell Stock II

```
int maxProfit(vector<int>& prices) {
    int n = prices.size();
    vector<array<int, 2>> memo(n, {-1, -1}); // -1: not computed
    return dfs(n - 1, false, memo, prices);
}
int dfs(int i, bool hold, vector<array<int, 2>>& memo,
vector<int>& prices) {
    if (i < 0) return hold ? INT_MIN : 0;
    int& result = memo[i][hold]; // refernce for updating memo
    if (result != -1) return result; // reuse
    if (hold) {
        result = max(dfs(i - 1, true, memo, prices), dfs(i - 1,
false, memo, prices) - prices[i]);
    } else {
        result = max(dfs(i - 1, false, memo, prices), dfs(i - 1,
true, memo, prices) + prices[i]);
    }
    return result;
};
```

```
int maxProfit(vector<int>& prices) {
    int n = prices.size();
    vector<array<int, 2>> dp(n + 1);
    dp[0][1] = INT_MIN;
    for (int i = 1; i <= n; i++) { // off the index by 1
        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i-
1]);
        dp[i][1] = max(dp[i-1][1], dp[i-1][0] - prices[i-
1]);
    }
    return dp[n][0];
}
```

Traveling Salesman Problem (TSP)

- Problem: Given A set of n cities and a distance matrix dist[i][j] representing the distance between city i and city j. Find the shortest path that starts from a given city, visits all cities once, and returns to the start city.
- Brute force approach by calculating all possible permutations of the cities: O(n!)
- Idea:
    - Factors to consider
        - Current city
        - Options of cities to travel
    - How to represent a (sub)set of cities? Using a bitmask,2^n

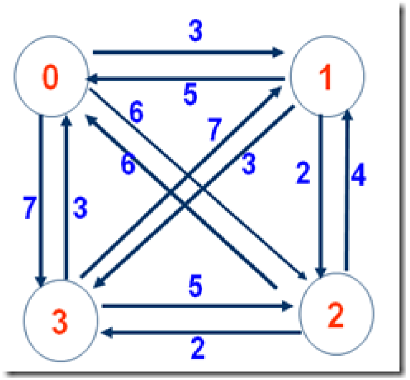


i\j	0	1	2	3
0	0	3	6	7
1	5	0	2	3
2	6	4	0	2
3	3	7	5	0

Using Bitmask for Set

- Add j (S U {j}) = set 1 at jth bit: S | 1<<j
- Remove j (S \ {j}) = set 0 at jth bit: S & ~(1<<j)
- Toggle at jth bit: S ^ (1<<j)
- Check if j is inside S = Check if jth bit is 1: if (S & (1<<j))
  or if ((S>>j) & 1)
- Set as empty: S = 0, Set as full: S = (1<<n)-1
- S's complement set: S = S ^ ((1<<n)-1)
- Union: S | T, intersection: S & T
- Enumerate all subsets, n bits:
  - `for (int s=0; s<(1<<n); s++) cout<<s<<endl;`
- Enumerate all subsets of X, including X:
  - `for (int s=X; s>=0; s=(s-1)&X) cout<<s<<endl;`

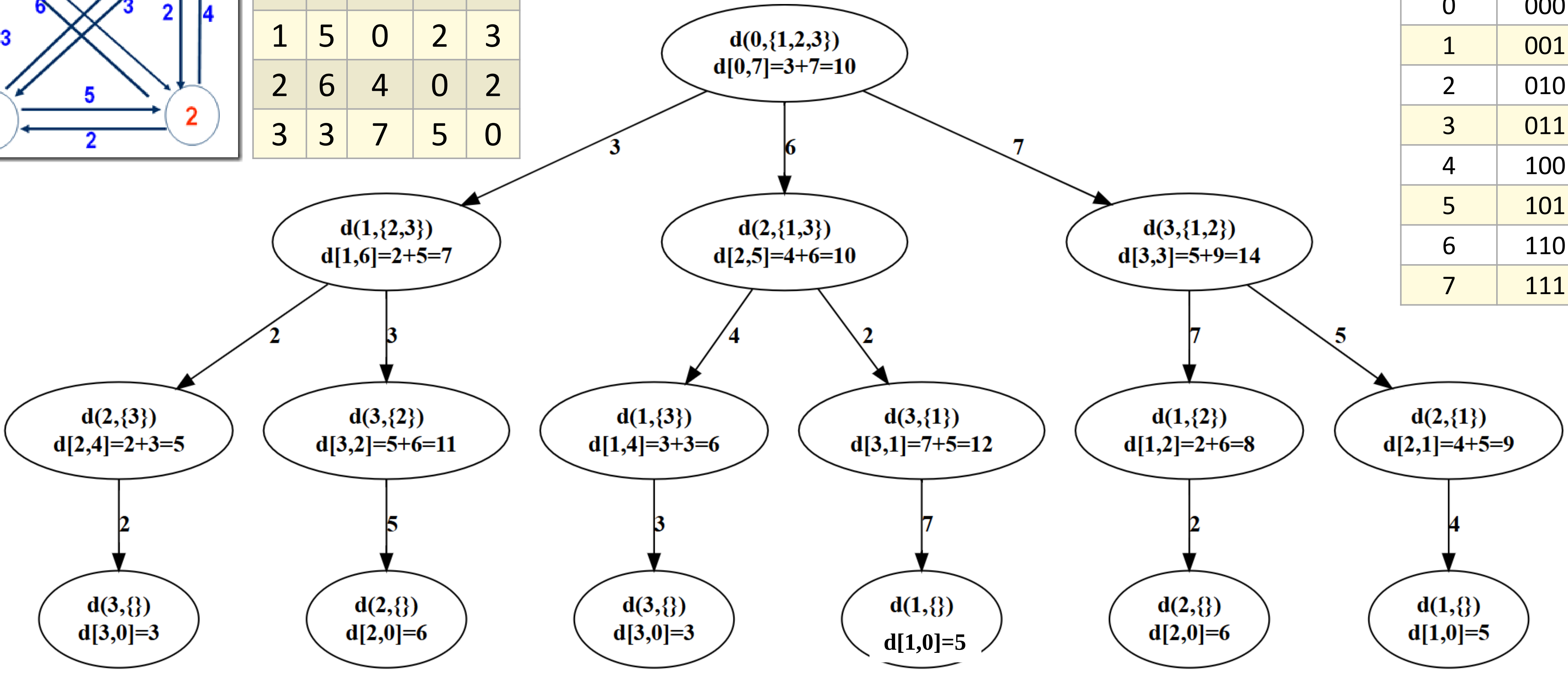
Traveling Salesman Problem (TSP)



i\j	0	1	2	3
0	0	3	6	7
1	5	0	2	3
2	6	4	0	2
3	3	7	5	0

ans = min(ans, dist[i][j] + d(j, mask & ~(1 << j)))

num	bit	set
0	000	{}
1	001	{1}
2	010	{2}
3	011	{1,2}
4	100	{3}
5	101	{1,3}
6	110	{2,3}
7	111	{1,2,3}



Traveling Salesman Problem (TSP)

- Subproblem:
  - d(i, mask): minimum cost to visit all cities in mask (a bitmask) and ended at city i.
  - mask is a bitmask where the jth bit is
    - 1 if the jth city is not visited and available
    - 0 if the jth city is visited and not available
- Relate:
  - For each j, compute the minimum cost (ans) of visiting j from i + visiting all cities in mask \ {j}:

```
    ans = min(ans, dist[i][j] + d(j, mask & ~(1 << j)))
  d(i, mask) = ans
```

- Topological order: enumerate all subsets
- Base Case: if mask is empty, dist[i][0]
- Time analysis: O(n*n*2^n), at most O(n*2^n) subproblems, and each one takes O(n) to solve

Traveling Salesman Problem (TSP)

```
const int MAXN = 20;
int dist[MAXN][MAXN], memo[MAXN][1 << MAXN], n;
void build() {
    fill(&memo[0][0], &memo[0][0] + MAXN * (1 << MAXN), -1);
}
int d(int i, int s) {
    if (s == 0) return dist[i][0]; // all visited
    if (memo[i][s] != -1) return memo[i][s]; // reuse

    int ans = INT_MAX;
    for (int j = n-1; j >=0; j--) {
        if ((s & (1 << j)) != 0) // if j is not visited
            ans = min(ans, dist[i][j] + d(j, s & ~(1 << j)));
    }
    return memo[i][s] = ans;
}
```

```
int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    while (cin >> n) {
        build();
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                cin >> dist[i][j];
        cout << d(0, (1 << n) - 1) << endl;
    }
    return 0;
}
```

Minimum XOR Sum of Two Arrays

- Problem:
  - You are given two integer arrays nums1 and nums2 of length n.
  - The XOR sum of the two integer arrays is (nums1[0] XOR nums2[0]) + (nums1[1] XOR nums2[1]) + ... + (nums1[n - 1] XOR nums2[n - 1]) (0-indexed).
  - For example, the XOR sum of [1,2,3] and [3,2,1] is equal to (1 XOR 3) + (2 XOR 2) + (3 XOR 1) = 2 + 0 + 2 = 4.
  - Rearrange the elements of nums2 such that the resulting XOR sum is minimized.
  - Return the XOR sum after the rearrangement
- Idea:
  - At each position, choose an element from the (set) nums2
  - Two factors to consider, index in nums1 and available elements from nums2, set ~ bitmask

https://leetcode.com/problems/minimum-xor-sum-of-two-arrays/

Minimum XOR Sum of Two Arrays

- Subproblems:
  - dp[i][mask]: the minimum XOR Sum when considering the first i elements in nums1, and the usage status of nums2, mask.
  - mask is a bitmask of length n:
    - 1 at j indicates that nums2[j] has been used
    - 0 at j indicates that nums2[j] has not been used
- Relate:
  - At i, the first i elements are considered, thus, the numbers of 1s in mask should be i otherwise skip
  - Enumerate nums1[i] matches with nums2[j], jth bit in mask should be 1
  - Transit from previous: i-1 of nums1 elements, previous state of nums2 (with 0 at j)
  - dp[i][mask]=min(dp[i][mask], dp[i–1][prev] + nums1[i] ^ nums2[j])
  - prev is set as 0 at jth bit in mask, prev = s & ~(1 << j)
- Topological order: shorter to longer, increasing i, increasing j
- Base cases: dp[0][0] = 0
- Original problem: dp[n][(1<<n)-1]
- Time analysis: O(n*n*2^n), O(n*2^n) subproblems, each transit cost O(n)

Minimum XOR Sum of Two Arrays

```
int minimumXORSum(vector<int>& nums1, vector<int>& nums2) {
    int n = nums1.size(), mask = 1 << n;
    vector<vector<int>> dp(n + 10, vector<int>(mask, 0x3f3f3f3f));
    dp[0][0] = 0;
    auto get1Cnt = [&](int s, int n) { // count the number of 1s
        int ans = 0; for (int i = 0; i < n; i++) ans += (s >> i) & 1;return ans;};

    for (int i = 1; i <= n; i++) {
        for (int s = 0; s < mask; s++) {
            if (get1Cnt(s, n) != i) continue; // invalid # if 1s
            for (int j = 1; j <= n; j++) {
                if ((s >> (j-1)) & 1 == 0) continue; // j-1 not used
                dp[i][s] = min(dp[i][s], dp[i-1][s & ~(1<<(j-1))] + (nums1[i-1] ^ nums2[j-1]));
            }
        }
    }
    return dp[n][mask - 1];
}
```

Numbers With Repeated Digits

- Problem:
  - Given an integer n, return the number of positive integers in the range [1, n] that have at least one repeated digit.
  - Input: n = 20
  - Output: 1
  - Explanation: The only positive number (<= 20) with at least 1 repeated digit is 11.
- Idea
  - Enumerate all digits to meet constraints and validity requirements
  - Set to help with selecting digits, set ~ bitmask
  - If the problem is difficult (i.e., at least one), we can consider negate it (i.e., ~(at least one) == no)
    - Count of numbers without repeated digits
    - The answer is then equal to n minus the count of numbers without repeated digits

https://leetcode.com/problems/numbers-with-repeated-digits/

Numbers With Repeated Digits

- Subproblems/Relate
  - Convert n into a string s
  - dp(i,mask,isLimit,isNum): the number of valid ways to construct digits from the ith position onwards.
  - Consider these factors (parameters) to make the number valid:
    - **mask:** Indicates the set of digits that have already been used, the digit chosen for the ith position must not be in the mask.
    - **isLimit:** Indicates if current digit x is constrained by n, if true, x<= s[i], otherwise, x<= 9. If x==s[i], subsequent digits will still be constrained by n
      - E.g., if n=123, and the digit at i=0 is 1, then the digit at i=1 can be at most 2 (13.. > 123, invalid)
    - **isNum:** Indicates whether the preceding digits (before the i-th position) have been filled with digits. If false, skip or fill with >=1. If true, fill with >=0.
      - E.g., if n=123, skipping the digit at i=0 means constructing a number less than 100. If i=1 is not skipped, then the number being constructed will be between 10 and 99; if i=1 is skipped, the number will be less than 10
- Topological order: enumerate the digits
- Base case: s[0], mask is empty, isLimit by n, no preceding digit:dp(0, 0, true, false)
- Time Complexity: O(10m2^10), O(m2^10) subproblems (m is the length of s), transit cost: O(10)

Numbers With Repeated Digits

```
int dp (int i, int mask, bool is_limit, bool is_num, int memo[][1 << 10], string& s){
    if (i == s.length()) return is_num; // is_num is true then find one
    if (!is_limit && is_num && memo[i][mask] != -1)
        return memo[i][mask]; // reuse
    int result = 0;
    if (!is_num) // skip the current position
        result = dp(i + 1, mask, false, false, memo, s);
    int up = is_limit ? s[i] - '0' : 9; // if all number before are equal to n's, then <= s[i]
    for (int d = 1 - is_num; d <= up; ++d) // enum digit d
        if ((mask >> d & 1) == 0) // d is not in mask
            result += dp(i + 1, mask | (1 << d), is_limit && d == up, true, memo, s);
    if (!is_limit && is_num)
        memo[i][mask] = result;
    return result;
};
```

```
int numDupDigitsAtMostN(int n) {
    auto s = to_string(n);
    int m = s.length();
    int memo[m][1 << 10];
    memset(memo, -1, sizeof(memo)); // -1 not computted
    return n - dp(0, 0, true, false, memo, s);
}
```

Summary

- State machine point of view
  - Best Time to Buy and Sell Stock II
- State with a set as parameter: bitmask
  - Traveling Salesman Problem
  - Minimum XOR Sum of Two Arrays
- State of Digits: consider index, bitmask, constraints and validity
  - Numbers With Repeated Digits
- Exercises:
  - Best Time to Buy and Sell Stock problems in Leetcode: 1-4, cooldown, transection fee
  - https://leetcode.com/problems/shortest-path-visiting-all-nodes/
  - https://leetcode.com/problems/beautiful-arrangement/
  - https://leetcode.com/problems/number-of-digit-one/