- Vertex cover of G is a set of vertices s.t. every edge in G is incident to at least one of these vertices. Given an undirected graph G = (V, E) and k, determine whether G has a vertex cover containing <= k vertices. Vertex-cover problem belongs to NP, since we can guess a cover of size <= k and check it easily in polynomial time. To prove that the vertex-cover problem is NP-complete we have to reduce an NP-complete problem to it (i.e. Clique Problem). We have to transform an arbitrary instance of the clique problem into an instance of the vertex-cover problem

- Let G = (V, E) and k represent an arbitrary instance of the clique problem. Let G' = (V, E') be the complement graph of G (i.e. G' has the same vertices and two vertices are connected in G' iff they are not connected in G). We claim that the clique problem is reduced to the vertex cover problem represented by the graph G' and n – k. Suppose that C = (U, F) is a clique in G. The set of vertices V – U covers all the edges in G', because in G' there are no edges connecting vertices in U (they are all in G). Thus, V – U is a vertex cover in G'. Therefore, if G has a clique of size k, then G' has a vertex cover of size n – k. Conversely, let D be a vertex cover in G'. Then, D covers all edges in G', so in G' there could be no edges connecting vertices in V – D. Thus, V – D generates a clique in G. Therefore, if there is a vertex cover of size k in G', then there is a click of size n – k in G. Can be performed in polynomial time since requires only the construction of G' from G.

------------------------------------------------------------------------

- Dominating set D is a set of vertices in G s.t. every vertex of G is either in D or is adjacent to at least one vertex from D. Given an undirected graph G = (V, E) and k, determine whether G has a dominating set containing <= k vertices. We reduce the vertex-cover problem to the dominating-set problem.

- Given an arbitrary instance (G, k) of the vertex-cover problem, our goal is to construct a new graph G' that has a dominating set of a certain size iff G has a vertex cover of size <= k. We start with G, and add |E| new vertices and 2|E| new edges to it. For each edge (v, w) of G, we add a new vertex vw and two new edges (v, vw) and (w, vw). In other words, we transform every edge into a triangle. Denote the new graph G'. It is easy to construct G' in polynomial time. We claim that G' has a dominating set of size m iff G has a vertex cover of size m. Let D be a dominating set of G'. If D contains any of the new vertices vw, then it can be replaced by either v or w and the set will still be a dominating set (both v and w cover all the vertices that vw covers). So, we can assume that D contains only vertices from G. But, since D dominates all the new vertices, it must contain at least one vertex from each original edge; hence, it is also a vertex cover for G. Conversely, if G is a vertex cover for G, then each edge is covered by C, so all new vertices are dominated. Old vertices are also dominated since all edges are covered.

------------------------------------------------------------------------

- 3SAT problem is an instance of the SAT problem in which in the Boolean expression, each clause contains exactly three variables. Given a Boolean expression in CNF s.t. each clause contains exactly three variables, determine whether it is satisfiable. We reduce the SAT problem to the 3SAT problem.

- First, 3SAT clearly belongs to NP as we can guess a truth assignment and verify that it satisfies the expression in polynomial time. Let E be an arbitrary instance of SAT. Replace each clause of E with several clauses, each of which has exactly 3 variables. Let C = (x1 + x2 + … + xk) be an arbitrary clause of E s.t. k >= 4. We write each variable in its "positive" form (i.e. don't use xi') for convenience of notation. We now show how to replace C with several clauses, each with 3 variables. The idea is to introduce new variables y1, y2, …,yk-3 that transform the clause into a 3SAT formulation without affecting its satisfiability. We use new variables for each clause. C is transformed into C' s.t. C' = (x1 + x2 + y1) . (x3 + y1' + y2) . (x4 + y2' + y3) … (xk -1 + xk + yk-3'). We claim that C' is satisfiable iff C is satisfiable. If C is satisfiable, then one of the xi's must be set to 1. In that case, we can set the values of the yi's in C' s.t. all clauses in C' are satisfiable as well. In general, if xi = 1, then we set y1, y2, …, yi-2 to be 1, and the rest to be 0, which satisfies C'. Conversely, if C' is satisfiable, then we claim that at least one of the xi's must be 1. Indeed , if all xi's are 0, then the expression becomes (y1) . (y1' + y2) . (y2' + y3) … (yk-3'), which is unsatisfiable. Using this reduction, we can replace any clause that has more than 3 variables with several clauses, each with exactly 3 variables. Thus, we have reduced a general instance of SAT into an instance of 3SAT s.t. one instance is satisfiable iff the other is. The reduction can clearly be done in polynomial time.

------------------------------------------------------------------------

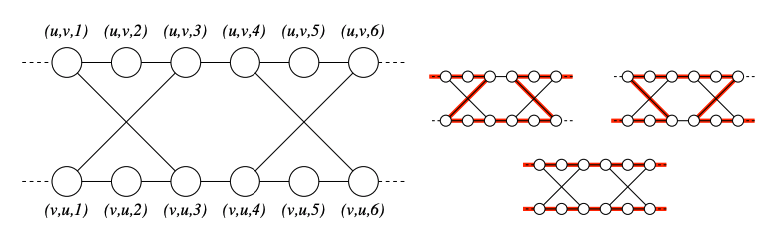
- Clique C in G is a subgraph of G s.t. all vertices in C are connected to all other vertices in C (i.e. C is a complete subgraph). Given an undirected graph G = (V, E) and k, determine whether G contains a clique of size >= k. We reduce SAT to the clique problem. Let E be an arbitrary Boolean expr in CNF, E = E1 . E2 … Em. Consider the clause Ei = (x + y + z + w). We associate a "column" of four vertices with the variables in E' even if they also appear in other clauses. That is, the graph G will have a vertex for each appearance of each variable. The question is how to connect these vertices s.t. G contains a clique of size >= k iff E is satisfiable. We choose k to be equal to the number of clauses m. The edges of G are as follows. Vertices from the same columns (i.e. vertices associated with variables of the same clause) are not connected. Vertices from different columns are almost always connected unless they correspond to the same variable appearing in complementary form. That is, the only time we don't connect 2 vertices from different clauses is when one is x and the other is x'. Ex: E = (x + y + z') . (x' + y' + z) . (y + z'). G can be constructed in polynomial time. We claim that G has a clique of size >= m iff E is satisfiable. The construction guarantees that the maximal clique size doesn't exceed m independent of E. Assume E is satisfiable. Then, there exists a truth assignment s.t. each clause contains at least one variable whose value is 1. We choose the vertex corresponding to this variable for the clique. The result is indeed a clique, since the only time 2 vertices from different columns aren't connected is when they are the complement of each other, which can't happen in a consistent truth assigment. Conversely, assume that G contains a clique of size >= m. The clique must contain exactly one vertex from each column. We assign the corresponding variables a value of 1. Since all the vertices in the clique are connected to one another, and we made sure that x and x' are never connected, the truth assignment is consistent.

------------------------------------------------------------------------

- Given an undirected graph G = (V, E), determine whether G can be 3-colored. We reduce 3SAT to the 3-coloring problem. Let E be an arbitrary instance of 3SAT. We have to construct a graph G s.t. E is satisfiable iff G can be 3-colored. First, build the main triangle M; and since M is a triangle, it requires at least 3 colors. We label M with the "colors" T, F, and A. These colors are used only for the proof; they are not part of the graph. We will later associate these colors with the assignment of truth-values to the variables of E. For each variable x, we build another triangle Mx whose vertices are x, x', and A, where A is the same vertex in M. So, if there are k variables, we will have k + 1 triangles, all sharing one common vertex A. The idea is that, if x is colored with the color T, then x' must be colored with F (since they are both connected to A). We now impose the condition that at least one variable in each clause has value 1. Assume that the clause is (x + y + z). We introduce six new vertices and connect them to the existing vertices (there is only 1 vertex in the whole graph labeled T, and one vertex for each x, y, or z). Call the 3 new vertices connected to T and x, y, or z the outer vertices, and the 3 new vertices in the triangle the inner vertices. We claim that this construct guarantees that, if no more than 3 colors are used, then at least one of x, y, or z must be colored T. None of them can be colored A, since they are all connected to A. If all are colored F, then the 3 new vertices connected to them must be colored A, but then the inner triangle cannot be colored with 3 colors. The complete graph corresponding to the expression (x' + y + z') . (x' + y' + z). If E is satisfiable then there is a satisfiable truth assignment. We color the vertices associated with the variables according to this truth assignment (T if x = 1, and F otherwise). M is colored with T, F, and A as indicated. Each clause must have at least one variable whose value is 1. Hence, we can color the corresponding outer vertex with F, the rest of the outer vertices with A, and the inner triangle accordingly. Thus, G can be colored with 3 colors. Conversely, if G can be colored with 3 colors, we name the colors according to the coloring of M (which must be colored with 3 colors). Because of the triangles, the colors of the variables correspond to a consistent truth assignment. The construct guarantees that at least variable in each clause is colored with T. Finally, G can clearly be constructed in polynomial time.

------------------------------------------------------------------------

- Hamiltonian cycle in a graph is a simple cycle that contains each vertex exactly once. Determine whether a given graph contains a Hamiltonian cycle. We reduce vertex-cover to Hamiltonian cycle. Given a graph G and k, we create another graph G' s.t. G' has a Hamiltonian cycle iff G has a vertex cover of size k. Our transformation will consist of putting together several gadgets. For each edge (u, v) in G, we have an edge gadget in G' consisting of 12 vertices and 14 edges. An edge gadget for (u, v) is the only possible Hamiltonian paths through it. The four corner vertices (u, v, 1), (u, v, 6), (v, u, 1), and (v, u, 6) each have an edge leaving the gadget. A Hamiltonian cycle can only pass through an edge gadget in one of the 3 ways shown in the figure. These paths through the edge gadget will correspond to one or both of the vertices u and v being in the vertex cover. G' also contains k cover vertices, simply numbered 1 to k. For each vertex u in G, we string together all the edge gadgets for edges (u, v) into a single vertex chain and then connect the ends of the chain to all the cover vertices. Specifically, suppose u has d neighbors v1, v2, …, vd. Then G' has the following edges: d – 1 edges between (u, vi, 6) and (u, vi+1, 1) (for all i between 1 and d – 1); k edges between the cover vertices and (u, v1, 1); and k edges between the cover vertices and (u, vd, 6). It's not hard to prove that if {v1, v2, …, vk} is a vertex cover of G, then G' has a Hamiltonian Cycle. To get this cycle, we start at cover vertex 1, traverse through the vertex chain for v1, then visit cover vertex 2, then traverse the vertex chain for v2, and so forth, until returning to cover vertex 1. Conversely, one can prove that any Hamiltonian cycle in G' alternates between cover vertices and vertex chains, and that the vertex chains correspond to the k vertices in a vertex cover of G. Thus, G has a vertex cover of size k iff G' has a Hamiltonian Cycle.



------------------------------------------------------------------------

- Traveling salesman. Let G = (V, E) be a weighted complete graph. A traveling salesman tour is a Hamiltonian cycle. Determine, given G and W, whether there exists a traveling-salesman tour s.t. the total length of its edges is <= W. Reduction from Hamiltonian cycle. For a given weighted graph G' = (V', E'), with non-negative weights, and k', the Traveling Salesman problem is to find whether G' contains a simple cycle of length <= k that passes through all the vertices [length of a cycle is the sum of weights of all the edges in the cycle]. To reduce the HC problem to the TSP for a given graph G, complete the graph G, by adding edges between all pairs of vertices that were not connected in G. Let the new graph be G' = (V', E') where V' = V and E' = {(u, v)} for any u, v element of V'. For edges in G' that were also present in G, assign weight 0. For other edges assign weight 1. Construction of the complementary graph can be done in polynomial time. The graph G has a HC iff there exists a cycle in G' passing through all vertices exactly once, and that has a length <= 0 (i.e. has a solution for the instance of the TSP where k = 0). If so, then the cycle contains only edges original present in G (the new edges in G' have weight 1 and hence can't be part of a cycle of length <= 0). Hence there exists a HC in G. If there exists a HC in G, it forms a cycle in G' with length = 0, since the weights of all the edges is 0. Hence there exists a solution for the TSP in G' with length <= 0.

------------------------------------------------------------------------

- Hamiltonian Path problem can be reduced from vertex-cover. This can be done by showing that HP and HC are polynomial-time reducible to each other, and that HC can be reduced from vertex-cover, which implies the original problem. Polynomial reduction from HP to HC. Given an instance of HP problem (G, s, t), we create a new graph G': add a new vertex v and two edges {t, v} and {v, s} to G. It's easy to see that there is a HP from s to t in G iff there is an HC in G'. Polynomial reduction from HC to HP. Given an instance of HC G, we create an instance of HP G': let G' be a copy of G, and add 2 new vertices s and t. For very node v in G that has an edges {s, v}, add an edge {v, t} in G'. Again, it can easily be seen that there is a HC in G iff there is a HP from s to t in G'.

------------------------------------------------------------------------

- Independent Set is a set of vertices no two of which are connected. Determine, given G and k, whether G contains an IS with >= k vertices. Reduction from vertex-cover. If G = (V, E), then S is an IS and V – S is a vertex-cover. Suppose S is an IS, and let e = (u, v) be some edge. Only one of u, v can be in S. Hence, at least one of u, v is in V – S. So, V – S is a vertex cover. Suppose V – S is a vertex cover, and let u, v be elements of S. There can't be an edge between u and v (otherwise, that edge wouldn't be covered in V – S). So, S is an IS. Hence, G contains an IS of size k and G contains a vertex-cover of size V – k. To show the other way around (i.e. polynomial-reduction from IS to vertex-cover), change any instance of IS into an instance of vertex-cover. Given an instance of IS (G, k), we ask our vertex-cover black box if there is a vertex cover V – S of size <= V – k. By our previous theorem, S is an IS iff V – S is a vertex cover. If the vertex-cover black box says yes, then S must be an IS of size >= k. If it says no, then there is no vertex cover V – S of size <= V – k, hence there is no IS of size >= k.

------------------------------------------------------------------------

- Set-Cover: Given a set U of elements, a collection S1, S2, …, Sn of subsets of U and K, determine if there exist a collection of at most k of these sets whose union is equal to U. Reduction from vertex-cover to set-cover. Given instance of VC G = (V, E) and j, construct instance of SC problem with U = E and n subsets of u: label the vertices of G from 1 to n, and let Si be the set of edges that incident to vertex i. Note that Si is a subset of U. Finally, let k = j. This construction can be done in polynomial time. Run black box for SC problem and return the same result that it gives. Need to show that the original VC instance is a yes iff the SC instance is also a yes. Suppose G has a VC of size at most j. Let S be such a set of nodes. By our construction, S corresponds to a collection C of subsets of U. Since k = j, C clearly has at most k subsets. We claim that the sets listed in C cover U. To see this, consider any element of U. Such an element is an edge e in G, and since S is a vertex cover for G, at least one of e's endpoints is in S. Therefore, C contains at least one of the sets associated with the endpoints of e, and by definition, these both contain e. Now suppose there is a set cover C of size k. Since each set in C is naturally associated with a vertex in G, let S be the set of these vertices. |S| = |C| and thus S contains at most j nodes. Furthermore, consider any edge e. Since e is in the ground set U and C is a set cover, C must contain at least one set that includes e. But by our construction, the only sets that include e correspond to nodes that are endpoints of e. Thus, S must contain at least one of the endpoints of e.

- Given a bipartite graph G = (V, E, U) in which we want to find a max-cardinality matching, we add two new vertices s and t, connect s to all vertices in V, and connect all vertices in U to t. We also direct all edges in E from V to U. We now assign capacities of 1 to all the edges, and we have a valid network-flow problem on the modified graph G'. Let M be a matching in G. There is a natural correspondence between M and a flow in G'. We assign a flow of 1 to all the edges in M and to all the edges connecting s or t to the matched vertices in M. All the other edges are assigned flow of 0. The total flow is thus equal to the number of edges in the matching. It turns out that M is a max matching iff the corresponding flow is a max flow in G'. One side is clear: if flow is max and it corresponds to a matching, then we cannot have a larger matching, since it would correspond to a larger flow.

------------------------------------------------------------------------

Baseball Elimination

- Our input consists of 2 arrays W[1..n] and G[1..n, 1..n], where W[i] is the number of games team i has already won, and G[i, j] is the number of upcoming games between teams i and j. We want to determine whether team n can end the season with the most wins. We want to assign a winner to each game, so that team n comes in first place. Let R[i] = the sum of G[i, j] denote the number of remaining games for team i. We assume team n wins all R[n] of its remaining games. Then team n can come in first place iff every other team i wins at most W[n] + R[n] – W[i] of its remaining R[i] remaining games. Since we want to assign winning teams to games, we start by building a bipartite graph whose nodes represent the games and the teams. We have (n 2) game nodes gi,j, one for each pair 1 <= i < j < n, and n – 1 team nodes ti, one for each 1 <= i < n. For each pair i, j, we add edges gi,j -> ti and gi,j -> tj with infinite capacity. We add a source vertex s and edges s -> gi,j with capacity G[i, j] for each pair i,j. Finally, we add target node t and edges ti->t with capacity W[n] – W[i] + R[n] for each team i. Theorem: Team n can end the season in first place iff there is a feasible flow in the graph that saturates every edge leaving s. Proof: Suppose it is possible for team n to end season in first place. Then every team i < n wins at most W[n] + R[n] – W[i] of the remaining games. For each game between team i and j that team i wins, add one unit of flow along the path s->gi,j->ti->t. Because there are exactly G[i, j] games between team i and j, every edge leaving s is saturated. Because each team i wins at most W[n] + R[n] – W[i] games, the resulting flow is feasible. Conversely, let f be a feasible flow that saturates every edge out of s. Suppose team i wins exactly f(gi,j->ti) games against team j, for all i and j. Then teams i and j play f(gi,j->ti) + f(gi,j->tj) = f(s->gi,j) = G[i, j] games, so every upcoming game is played. Moreover, each team i wins a total of the summation of f(gi,j->ti) = f(ti->t) <= W[n] + R[n] – W[i] upcoming games, and therefore at most W[n] + R[n] games overall. Thus, if team n wins all their upcoming games, they end season in first place. So, to decide whether our favorite team can win, construct flow network, compute max flow, and report whether max flow saturates edges leaving s.

------------------------------------------------------------------------

Project Selection

- Suppose we are given a set of n projects that we could possibly perform; for simplicity, we identify each project by an integer between 1 and n. Some projects cannot be started until certain other projects are completed. This set of dependencies is described by a directed acyclic graph, where an edge i->j indicates that project i depends on project j. Finally, each project i has an associated profit pi which is given to us if the project is completed; however, some projects have negative profits, which we interpret as positive costs. We can choose to finish any subset X of the projects that includes all its dependents; that is, for every project x ∈ X, every project that x depends on is also in X. Our goal is to find a valid subset of the projects whose total profit is as large as possible. In particular, if all of the jobs have negative profit, the correct answer is to do nothing. At a high level, our task to partition the projects into two subsets S and T, the jobs we Select and the jobs we Turn down. So intuitively, we’d like to model our problem as a minimum cut problem in a certain graph. But in which graph? How do we enforce prerequisites? We want to maximize profit, but we only know how to find minimum cuts. And how do we convert negative profits into positive capacities? We define a new graph G by adding a source vertex s and a target vertex t to the dependency graph, with an edge s->j for every profitable job (with pj > 0), and an edge i->t for every costly job (with pi < 0). Intuitively, we can think of s as a new job (“To the bank!”) with profit/cost 0 that we must perform last. We assign edge capacities as follows: c(s->j) = pj for every profitable job j; c(i->t) = −pi for every costly job i; c(i->j) = ∞ for every dependency edge i->j. All edge-capacities are positive, so this is a legal input to the maximum cut problem. Now consider an (s, t)-cut (S, T) in G. If the capacity |S, T| is finite, then for every dependency edge i->j, projects i and j are on the same side of the cut, which implies that S is a valid solution. Moreover, we claim that selecting the jobs in S earns us a total profit of C − |S, T|, where C is the sum of all the positive profits. This claim immediately implies that we can maximize our total profit by computing a minimum cut in G.

------------------------------------------------------------------------

MeetingRooms [O(NlogN)]

sort intervals based on start times; push first interval to min heap pq (sorted by end times)

for each i = 1 to n interval:

get interval on pq finishing earliest called interval

if curr starts at or after interval end time then update interval end time (interval.end = intervals[i].end)

else need new room, so push curr to pq

push interval (updated or not) back to pq

ret size of pq

Bipartite Graph [O(|V| + |E|)]

init arr colors

for each node (o to n) in g:

if g[node] is not empty and colors[node] = 0 then

colors[node] = 1

push node to queue q

while q is not empty:

get node u from q

for each edge (u, v) incident to u:

if color[u] = color[v] then

ret false

else if color[v] = 0 then

color[v] = (color[u] == 1 ? 2 : 1)

push v to q

ret true

------------------------------------------------------------------------

Dijkstra (shortest path) [O(M log N)]

init dist of all v to inf

create empty min heap pq

where item = (wt, vertex)

insert (0, src) to pq

while pq is not empty:

extract min dist vertex u

for each edge (u, v) incident to u:

if there is shorter path to v through u (i.e. if dist[u] + wt(u, v) < dist[v])

update dist of v (i.e. dist[v] = dist[u] + wt(u, v)

insert v to pq

------------------------------------------------------------------------

Prim (MST) [start from s and greedily build by adding e of min cost] [O(M log N)]

init parent of all v to -1

init dist of all v to inf

create empty min heap pq

where item = (wt, vertex)

(don't want to process some v already in MST)

init all vertices as not part of MST

insert (0, src) to pq

while pq is not empty:

extract min dist vertex u

for each edge (u, v) incident to u:

if inMST[v] = false and dist[v] > wt(u, v)

dist[v] = wt(u, v)

insert v into pq

parent[v] = u

print MST using parent

------------------------------------------------------------------------

Kruskal (MST) [insert edge by min cost] [O(M log M)]

sort edges by incr. cost

mstWt = 0

create DisjointSet (i.e. parent[v] = v, rank[v] = 0)

for each (u, v) from edges:

if u and v aren't in same set then

print edge(u, v)

mstWt += wt of (u, v)

union(u, v)

------------------------------------------------------------------------

Counting Inversions [Mergesort] [O(NlogN)]

Merge-n-cnt(A, B) [O(N)]

Maintain curr ptr into A, B

Maintain cnt for # inversions

while both lists not empty:

Let ai and bj be elements pointed to

Append smaller of the 2 to list C

If bj is smaller elem then

incr. cnt by # elems remaining in A

Advance curr ptr in lsit where elem removed

Once one list is empty, append rem. of the other list

Return cnt and merged list C

Sort-n-cnt (L) [O(NlogN)]

if list has 1 elem then no inversions

else div list into 2 halves A, B:

(ra, A) = srt-n-cnt(A)

(rb, B) = srt-n-cnt(B)

(rl, L) = merge-n-cnt(A, B)

ret r = ra + rb + rl and sorted list L

------------------------------------------------------------------------

Majority Elem [O(NlogN)]

if n = 1, ret a[0]

k = floor (n/2)

L = MajorityElem(a[1..k]), R = MajorityElem(a[k+1..n])

if L = R then ret L

Lcnt = GetFreq(a[1..n], L), Rcnt = GetFreq(a[1..n], R)

if Lcnt > k + 1, then ret L

elif Rcnt > k + 1, then ret R

else ret none

Subset Sum (O/1 Knapsack)

bool helper(arr nums, int t, int index) DP

if (t < 0) ret false

if (t == 0) ret true

for (i = index to nums.size)

if (helper(nums, t – nums[i], i + 1)) ret true

ret false

bool canPartition(arr nums)

int sum = 0

for (int i : nums) sum += i

ret helper(nums, sum / 2, 0)

bool canPartition(arr nums) 2D DP

int sum = 0

for (int i : nums) sum += i

if (sum % 2 != 0) ret false

sum /= 2

vector<vector<bool>> dp(nums.size + 1, vector<bool>(sum + 1))

for (int i = 0 to dp.size) dp[i][0] = true

for (int i = 1 to dp.size)

for (int j = 1 to dp[0].size)

dp[i][j] = dp[i – 1][j]

if (j >= nums[i – 1]) dp[i][j] = (dp[i][j] || dp[i – 1][j – nums[i – 1]]

ret dp[dp.size – 1][dp[0].size – 1]

------------------------------------------------------------------------

bool helper(nums, visited, t, k, currSum, startIndex)

if (k == 0) ret true

if (currSum == t) ret helper(nums, visited, t, k – 1, 0, 0)

for (i = startIndex to nums.size)

if (visited[i] == 0)

visited[i] = 1

if (helper(nums, visited, t, k, currSum + nums[i], i + 1)) ret true

visited[i] = 0

ret false

PartitionKEqualSubsets(arr nums, int k)

int sum = 0

for (int i : nums) sum += i

if (k <= 0 || sum % k != 0) ret false

vector<int> visited(nums.size)

ret helper(nums, visited, sum / k, k, 0, 0)

------------------------------------------------------------------------

Coin Change (min coins to get amt) [amt = 11, [1, 2, 5], ret 3 => 5 + 5 + 1] [O(N\*wt)]

int coinChange(arr coins, int amt)

vector<int> dp(amt + 1, amt + 1)

dp[0] = 0

for (j = 0 to coins.size)

for (i = 1; i <= amt)

if (1 + dp[i – coins[j]] < dp[i]) dp[i] = 1 + dp[i – coins[j]]

ret (dp[dp.size – 1] > amt) ? -1 : dp[dp.size – 1]

------------------------------------------------------------------------

Coin Change 2 (num combos that make amt) [amt = 4, [1, 2], ret 3 => 1 + 1 + 1 + 1, 2 + 2, 2 + 1 + 1]

int change(int amt, arr coins)

vector<vector<int>> dp(coins.size + 1, vector<int>(amt + 1))

for (i = 0 to dp.size) dp[i][0] = 1

for (i = 1 to dp.size)

for (j = 1 to dp[0].size)

dp[i][j] = dp[i – 1][j] // using coins [0, i – 1] to make amt j

// using coin i to make amt j + using coins [0, i] to make amt j – coins[i]

if (j >= coins[i – 1]) dp[i][j] += dp[i][j – coins[i – 1]]

ret dp[coins.size][amt]

------------------------------------------------------------------------

CountSmallerAfterSelf

Node { left, right, val, sum, dup = 1

Node(v, s) : val(v), sum(s) {} }

countSmaller(arr nums)

vector<int> rs(nums.size)

Node root = null

for (i = nums.size – 1 to 0)

root = insert(nums[i], root, rs, i, 0)

ret rs

insert(num, node, rs, i, preSum)

if (node is null)

node = new Node(num, 0)

rs[i] = preSum

elif (node.val == num)

node.dup++

rs[i] = preSum + node.sum

elif (node.val > num)

node.sum++

node.left = insert(num, node.left, rs, i, preSum)

else

node.right = insert(num, node.right, i, preSum + node.dup + node.sum)

ret node

------------------------------------------------------------------------

TowersOfHanoi(disk, src, dst, aux)

if disk = 1, then move src to dst

else

hanoi(disk – 1, src, aux, dst)

move src to dst

Hanoi(disk – 1, aux, dst, src)

------------------------------------------------------------------------

longestPalindromicSubstring(str s)

string rs = 0

vector<vector<bool>> dp(s.size, vector<bool>(s.size))

for (i = size – 1 to 0)

for (j = i to s.size)

dp[i][j] = (s[i] == s[j] && (j – i <= 2 || dp[i + 1][j – 1]))

if (dp[i][j] && (j – i + 1 > rs.size)) rs = s.substr(i, j – i + 1)

ret rs

------------------------------------------------------------------------

EditDistance(str w1, str w2)

vector<vector<int>> dp(w1.size + 1, vector<int>(w2.size + 1))

for (i = 0 to dp.size) dp[i][0] = i

for (j = 0 to dp[0].size) dp[0][j] = j

for (i = 1 to dp.size)

for(j = 1 to dp[0].size)

if (w1[i – 1] == w2[j – 1]) dp[i][j] = dp[i – 1][j – 1]

else dp[i][j] = min(dp[i – 1][j], min(dp[i][j – 1], dp[i – 1][j – 1])) + 1

ret dp[dp.size – 1][dp[0].size – 1]

------------------------------------------------------------------------

WordBreak

bool helper(str s, hashset<str> wordDict, i) Recursive

if (i == s.size) ret true

for (j = i + 1; j <= s.size)

if (s.substr(i, j – i) not in wordDict && helper(s, wordDict, j)) ret true

ret false

bool wordbreak(str s, arr wordDict)

hashset<string> set(wordDict.begin, wordDict.end)

ret helper(s, set, 0)

wordbreak(s, arr wordDict) DP

hashset<string> set(wordDict.begin, wordDict.end)

vector<int> dp(s.size + 1)

dp[0] = true

for (i = 1; i <= s.size)

for (j = i – 1 to 0)

if (s.substr(j, i – j) not in set && dp[j])

dp[i] = true; break

ret dp[dp.size – 1]

------------------------------------------------------------------------

RegexMatch(str s, str p)

vector<vector<bool>> dp(s.size + 1, vector<bool>(p.size + 1))

dp[0][0] = true

for (i = 1 to p.size)

if (p[i] == '\*' && dp[0][i – 1]) dp[0][i + 1] = true

for (i = 1 to dp.size)

for (j = 1 to dp[0].size)

if (p[j – 1] == '.' || p[j – 1] == s[i – 1]) dp[i][j] = dp[i – 1][j – 1]

elif (p[j – 1] == '\*')

dp[i][j] = dp[i][j – 2]

if (p[j – 2] == '.' || p[j – 2] == s[i – 1]) dp[i][j] |= dp[i – 1][j]

else dp[i][j] = false

ret dp[s.size][p.size]

------------------------------------------------------------------------

DecodeWays

int helper(str s, index) [Recursive]

if (index == s.size) ret 1

if (s[index] == '0') ret 0

int rs = helper(s, index + 1)

if (index < s.size – 1 && (s[index] == '1' || (s[index] == '2' && s[index + 1] < '7')))

rs += helper(s, index + 2)

ret rs

int decodeWays(str s) { ret helper(s, 0) }

int helper(str s, memo, index) Recursive + Memo [O(N)]

if (memo[index] != 0) ret memo[index]

if (index == s.size) ret 1

if (s[index] == '0') ret 0

int rs = helper(s, memo, index + 1)

if (index < s.size – 1 && (s[index] == '1' || (s[index] == '2' && s[index + 1] < '7')))

rs += helper(s, memo, index + 2)

ret memo[index] = rs

------------------------------------------------------------------------

TargetSum [num ways to assign symbols to sum of integers = S] [nums = [1, 1, 1, 1, 1], S = 3, ret 5]

helper(nums, rs, t, index)

if (index == nums.size)

if (t == 0) rs++

return

helper(nums, rs, t – nums[index], index + 1)

helper(nums, rs, t + nums[index], index + 1)

TargetSum(arr nums, S) { rs = 0; helper(nums, rs, S, 0); ret rs }