SMP and Gale Shapley

- instability: say (m, w') is instable if
  - m prefers w' more than his current w, and
  - w' prefers m more than her current partner m'
- note: the set S returned by GS is unique, even if there's more than one perfect pairing
- runtime: $\Theta(n^2)$

```
Gale-Shapley {
    // initially all m in M and w in W is free
    while (there is a man m who's free
             && has not proposed to every woman)
        {
        choose such a man m;
        let w be the highest ranked woman on m's
            pref list which m hasn't proposed to;
        if (w is free)
            {m, w} are now engaged
        else  # w is currently engaged to m'
            if (w prefers m* to m)
                do nothing;
            else
                (m, w) gets engaged;
                m* is free;
        }
        Return S (set of all engaged pairs
    }
```

Runtime Analysis

- big O, Ω, Θ definition

$$f = O(g): \ \exists c, \exists n_0 \ \text{ s.t } n \geq n_0 \rightarrow f(n) \leq cg(n)$$
$$f = \Omega(g): \ \exists c, \exists n_0 \ \text{ s. t } n \geq n_0 \rightarrow f(n) \geq cg(n)$$
$$\rightarrow \text{ or } g \in O(f)$$
$$f = \Theta(g): \ f = O(g) \text{ and } f = \Omega(g)$$

- small o, ω

$$f = o(g): \ \forall c, \exists n_0 \ \text{ s.t } n \geq n_0 \rightarrow f(n) < cg(n)$$
$$f = \omega(g) \ \forall c, \exists n_0 \ \text{ s.t } n \geq n_0 \rightarrow f(n) > cg(n)$$

- small $o, \omega + \Theta$ limit definition

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L \begin{cases} L = 0 \rightarrow f \in o(g) \\ L = c \rightarrow f \in \Theta(g) \\ L = \infty \rightarrow f \in w(g) \end{cases}$$

- big O, Ω limit definition

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = L \begin{cases} L \neq \infty \rightarrow f \in O(g) \\ L = \infty \rightarrow f \in \Omega(g) \end{cases}$$

- these are a list of common runtime (fastest $\rightarrow$ slowest in terms of runtime)

Constant	$\mathcal{O}(1)$	
Log	$\mathcal{O}(\log n)$	
PolyLog	$\mathcal{O}((\log n)^k)$	$1 < k$
SubLinear	$\mathcal{O}(n^c)$	$0 < c < 1$
Linear	$\mathcal{O}(n)$	
LogLinear	$\mathcal{O}(n \log n)$	
SubQuadratic	$\mathcal{O}(n^d)$	$1 < d < 2$
Quadratic	$\mathcal{O}(n^2)$	
LogQuadratic	$\mathcal{O}(n^2 \log n)$	
Cubic	$\mathcal{O}(n^3)$	
Polynomial	$\mathcal{O}(n^a)$	
	$\mathcal{O}(n^b)$	$3 < a < b$
Exponential	$\mathcal{O}(\alpha^n)$	
	$\mathcal{O}(\beta^n)$	$1 < \alpha < \beta$
Factorial	$\mathcal{O}(n!)$	
Power	$\mathcal{O}(n^n)$	

- important math/log rules

$$c^{\log_c a} = a \qquad \log_a(x) > \log_b(x) \text{ if } a < b$$
$$\log_c(a \cdot b) = \log_c a + \log_c b \qquad \log_c(b^k) = k \log_c b$$
$$\log_c(a/b) = \log_c a - \log_c b \qquad \log_a b = \log_c a / \log_c b$$
$$C(n,r) = \binom{n}{r} = \frac{n!}{r!(n-r)!} \qquad P(n,r) = \frac{n!}{(n-r)!}$$

Graphs

- **simple definitions**
  - path: sequence of verticies $\{v_1, v_2, \ldots, v_k\}$ s.t there exists an edge between consecutive vertices
  - simple path: a path that doesn't pass through any vertex more than once
  - cycle: path w/ common beginning and ending (ex. $\{v_1, v_2, \ldots, v_k\}$, $v_1 = v_k$)
- **types of graphs:**
  - simple: no self-edge and multi-edged vertices
  - cyclic: graph got at least 1 cycle
  - connected: every pair of distinct vertices has an edge b/t them
  - complete: every vertex has an edge to every other vertex in the graph
  - tree: undirected. connected, acyclic graph
- **math for graphs**: let graph $G = (V, E)$, $|V| = n$ and $|E| = m$
  - simple graphs:

$$\text{sum of degrees in G: } \sum_{v \in V} deg(v) = 2m$$
$$\text{\# of edges in complete graph: } m = \frac{n}{2}(n-1)$$

  - connected simple graph: $O(n) \subset O(m) \subset O(n^2)$
    $\longrightarrow m \in O(n) \Longrightarrow$ graph is sparse
    $\longrightarrow m \in O(n^2) \Longrightarrow$ graph is dense
  - min # of edges in connected graph: $m = n - 1$
  - max # of edges in connected graph
    $\longrightarrow$ simple: $n(n-1)/2$ (complete graph)
    $\longrightarrow$ not simple: DNE
- topological ordering: all vertices line up in a way that all edges point forward (top. ordering might not be unique)
- **some propositions:**
  - any pair of v's in a tree, there's a path that connects them
  - any graph w/ n vertices and n edges has a cycle
  - let G be an undirected graph w/ n vertices, if any of the following 2 is True, all 3 is True
    1. G is connected
    2. G is acyclic
    3. G has $n-1$ edges
- basic graph function runtime

	Adjacency Matrix	Adjacency List
Insert Vertex	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$
Remove Vertex	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$
Insert Edge	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Remove Edge	$\mathcal{O}(1)$	$\mathcal{O}(\deg(v))$
Vertices Adjacent?	$\mathcal{O}(1)$	$\mathcal{O}(\deg(v))$
Incident Edges	$\mathcal{O}(n)$	$\mathcal{O}(\deg(v))$

Search Algorithm

- **runtime for graph searching algorithms**

	Run Time	Data Structure
BFS	$\mathcal{O}(n+m)$	Queue
DFS	$\mathcal{O}(n+m)$	Stack or Recursion
Dijkstra's	$\mathcal{O}(m \log n)$	Priority Queue
Prim's	$\mathcal{O}(m \log n)$	Priority Queue
Kruskal's	$\mathcal{O}(m \log n)$	Disjoint Sets

- Prim's and Krushkal's are used to find MST in a weighted graph (can handle negative edges)
- Dijkstra's used to find shortest path between two nodes in a weighted graph (cannot handle negative edges)
- **Note**: heights of BFS & DFS nodes depends on start node and order nodes are checked

- BFS
  - uses a queue → explores the graph in layers
  - used to find the shortest path from s to all $v \in V$

```
BFS(s) { //s is the start node
    queue.enqueue(s);
    mark s as visited;
    while (!queue.empty)
        u = queue.dequeue;
        for each edge(u,v) {
            if (v is not visited)
                mark v as visited;
                q.enqueue(v);
                //could add p[v] = u here to
                //keep track of parents
        }
}
```

- DFS
  - uses a stack/recursion
  - explores the point further from the graph first
  - does not find the shortest path
  - can identify cycles

```
DFS(s) {
    for each i in [1...n]
        explored[i] = false;
    DFSHelper(s)
}

DFSHelper(u) {
    explored[u] = true;
    for each edge (u,v) {
        if (v is not visited)
            p[v] = u;
            DFSHelper(v)
    }
}
```

Divide and Conquer

- Summations and Geometric Series

$$\sum_{i=1}^{n} ar^i = a\left(\frac{1-r^n}{1-r}\right) = a \cdot \frac{r^{n+1}-1}{r-1} \qquad \sum_{i=0}^{\infty} ar^i = \frac{a}{1-r} \quad \text{if } |r| < 1$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=1}^{n} i^2 = \frac{n(n+1)(n+2)}{6}$$

- **The Master Theorem**

$$T(n) = \begin{cases} c \ \ (\text{has to be constant}) & \text{if } n < n_0 \\ aT\left(\frac{n}{b}\right) + cn^k & \text{if } n \geq n_0 \end{cases}$$

$$\text{if } a > b^k : T(n) \in \Theta(n^{\log_b a})$$

$$\text{if } a = b^k : T(n) \in \Theta(n^k \log n)$$

$$\text{if } a < b^k : T(n) \in \Theta(n^k)$$

→ Master theorem does not always give same bound as tree

- let tree be $T(n) = aT(n/b) + cn$, then $height(tree) = \log_b n$
- some divide and conquer algo w/ their runtime
  ⟶ quickSort: $\Theta(n \log n)$
  ⟶ quickSelect (find k-th largest element in array): $\Theta(n)$
- find work per level: get work at level one in form of $cn^k \cdot (a/b)$ and the work per level is $cn^k(a/b)^i$
- if you can't use Master theorem (unequal split) - 2 ways
  - **massage into Master Theorem**:

$$T(n) = T\left(\frac{3n}{4}\right) + T\left(\frac{n}{8}\right) + cn$$

$$\text{define } L(n) = T\left(\frac{n}{8}\right) + T\left(\frac{n}{8}\right) + cn \leq T(n)$$

$$= 2T\left(\frac{n}{8}\right) + cn$$

$$U(n) = 2T\left(\frac{3n}{4}\right) + cn \geq T(n)$$

$$\text{via Master, } L(n) = \Theta(n) \therefore T(n) = \Omega(n)$$

$$U(n) = \Theta(n) \therefore T(n) = O(n)$$

$$\therefore T(n) = \theta(n)$$

  - **take sum to infinity**: the lower bound is the work done at level 0, and upper bound is sum of work done per level take to infinity
    - usually give tighter bound than the Master Theorem
    - ex. $T(n) = 2T(n/3) + T(n/4) + cn^2$ for $n > 3$, constant otherwise
      Work done at level 0: one root node of size n with time cn^2
      Work done at level 1:

$$c(n/3)^2 + c(n/3)^2 + c(n/4)^2 = cn^2(41/144)$$

      Work per level: $cn^2 \cdot (41/144)^i$
      Lower bound: work done at level 0 so lower bound is $\Omega(cn^2)$
      Upper bound:

$$\sum_{i=1}^{\infty} cn^2\left(\frac{41}{144}\right)^i = \frac{cn^2}{1-\frac{41}{144}} = cn^2\left(\frac{103}{144}\right) = O(n^2)$$

Dynamic Programming

The following example will use the Fibonnaci sequence

$$F(n) = \begin{cases} 1 & \text{if } n \leq 2 \\ F(n-1) + F(n-2) & \text{if } n > 2 \end{cases}$$

1. Recursion
   - is pretty shit in terms of run time

```
Fib(n) {
    if (n <= 2)
        return 1;
    else {
        return F(n-2) + F(n-1)
    }
}
```

2. Top-down Memoization
   - store past calls/results to memory
   - needs helper → in helper check if it's been computed yet, usually means checking if the entry is 0 or −1

```
FibMem(n) {
    soln = new int[i...n]; // make soln array
    //fill array with 0
    return FibMemHelper(n, soln)
}

FibMemHelper(n, soln) {
    if (n <= 2)
        return 1;
    if (soln[n] != 0) // means it's been found
        return soln[n];
    else
        return FibMemHelper(n-1) + FibMemHelper(n-2)
}
```

3. Dynamic Programming (Bottom-Up)
   - calculate all necessary entries first
   - use for loop → usually code inside for loop is direct translation of recurrence relation

```
FibDP(n) {
    arr[] = new int[1...n];
    arr[1] = 1; // base cases
    arr[2] = 1;
    for (int i = 3; i <= n; i++){
        arr[i] = arr[i-1] + arr[i-2];
    }
    return arr[n];
}
```

- making change problem: give the minimum number of change for a dollar amount - imagine all possible coins are \$0.25, \$0.10, and \$0.01.

```
MakeChange(n) {
    soln[] = new int[n+1];
    //fill it with -1
    return MCH(n)
}

MCH(i, soln) {
    if (i < 0): return infinity
        // to prevent out of bounds index
    else if (i == 0): return 0; // base case
    else
        if (soln[i] == -1)
            soln[i] = min(MCH(i-1), MCH(i-5), MCH(i-25));
        return soln[i];
}
_____________________________________________________
DP-Change(n) { //asume n > 0
    soln[] = new int[n+1]; //allow for 1-indexing
    for (i = 1; i <= n; i++)
        soln[i] = min(helper(i-1), helper(i-5),
                      helper(i-25));
    return soln[i];
}

helper(i) {
    // just need this to make sure index is in bounds
    if (i < 0): return infinity;
    else if (i == 0): return 0;
    else: return soln[i];
}
```

- **runtime**
  - recursion: see how many recursive call is called per iteration (call a), and how deep the recursive call (call b) - get a^b, multiply that with time for the base case
    - ex. Fibonacci: 2 recursive call, each getting called n times $r \to 2^n$ factor and base case is $O(1)$ - so total runtime is $O(2^n)$
  - Memoization & DP: think about how much work you're doing without recursive call and then think about how many (non-repetitive) recursive call you're making, then calculate the total
    - thinking about it in terms of for loops and DP makes a bit more sense
    - ex. Fibonacci: Work without recursive call is $\Theta(1)$. Number of unique recursive call is $\Theta(n)$. So total is $\Theta(n)$

- tips - when identifying sub problems
  - if problem involve discrete quantities then sub problem would be about smaller quantities $\to$ going backwards - likely memoization
  - if problem is going forward (computing the next value or seeing if going forward in 1 direction gives the max/min - i.e midterm scheduling q) $\to$ likely DP

NP Complete

- decision problem: problem that could be posed as yes no question - need to convert optimization problems into yes/no problems for NP proof
  $\longrightarrow$ "maximizing ..." = "... with size of at least k"
  $\longrightarrow$ "minimizing ..." = "... with size of at most k"
- let **P** denotes set of decision problems where there's a known polynomial time algorithm
- efficient certifiers: an algo B is an efficient certifier for problem X if it can take a possible result and verify if it satisfy X in poly time
- let **NP** denote set of problems for which we do not know if there's a poly-time algorithm. An algorithm X is in NP if there's an efficient certifier for it
- **NP-Complete** are questions in NP but not P. So we don't know if there's an efficient algo. A decision problem X is in **NP-Complete** if:
  - $X \in NP$, and
  - $Y \leq_p X$, $\forall Y \in NP$ (generalized to any (one) $Y \in NP$)
  - $Y \leq_p X$ means that there's a poly-time reduction from Y to X, or "X is at least as hard as Y"
- it's obvious that **P** $\subseteq$ **NP**
- if $Z \leq_p Y$ and $Y \leq_p X$, then $Z \leq_p X$
- if $Y \leq_p X$ and $X \in P$, then $Y \in P$
  $\longrightarrow$ if $Y \leq_p X$ and $Y \notin P$, then $X \notin P$ (contrapsitive)
- **NP-Hard**: like NP-Complete but you drop the first requirement, you don't know it's in NP, just that it can be reduced from a problem in NP
- steps to NP-Complete proof
  1. Show that there exist an efficient certifier for X (if answer is yes, then there's a proof of this fact that can be verified in P)
  2. Pick a known NP-Complete problem Y and specify how to reduce Y to X
  3. Prove that reduction is correct
     $\longrightarrow$ meaning (yes instance in Y $\Longleftrightarrow$ yes instance in X)
- **aside - recall Bipartite Graph:** A graph is Bipartite if you can partition V into V_1 and V_2 such that there are no adjacent edges in V_1 and no adjacent edges in V_2 (likewise, if vertices are adjacent, they're in different sets).
  - graph is bipartite if it's 2-colorable and has no odd cycles

Important Problems

The following pairs of NP-complete problems are of type {**packing**, **covering**, **partitioning problems**, **sequencing**, **numerical**}

- **Independent Set**: For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is independent if no vertices in S are joined by any edge. Given a graph G and $k \in \mathbb{N}$ does G contain an independent set of size k or larger
- **Set Packing**: Given an n-element set U, a collection of subsets $\{S_1, S_2, \ldots, S_m\} \subset U$ and $k \in \mathbb{N}$, does there exists a collection of at least k of those sets with property that no two of them intersect
- **Vertex Cover**: For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a vertex cover if every edge $e \in E$ has at least one endpoints in S. Given a graph G and $k \in \mathbb{N}$, does G contain a vertex cover of size k or smaller
- **Set Cover**: Given an n-element set U, a collection of subsets $\{S_1, S_2, \ldots, S_m\} \subset U$ & a number k, is there a collection of at most k of those sets whose union is equal to U.
- **3D Matching**: Given 3 disjoint sets, X, Y, Z, is there a set $T \subseteq X \times Y \times Z$ such that each elements of $U \cup Y \cup Z$ is contained exactly once in these triples
  - ex. $X = \{\text{instructors}\}, Y = \{\text{courses}\}, Z = \{\text{time slots}\}$
  - is a special case of Set Cover, we're looking to cover the ground set $U = X \cup Y \cup Z$ using at most n sets from $X \times Y \times Z$
  - is a special case of Set Packing, since we're looking for n disjoint subsets of ground set $U = X \cup Y \cup Z$
  - Bipartite Matching (aka 2D matching): Given 2 Bipartite sets U and V find the maximum matching
    - ex. $U = \{readers\}, V = \{books\}$ and (u, v) is book v person u willing to read $\to$ solve in $O(mn)$ time
- **Graph Coloring**: A graph $G = (V, E)$ is said to be k-colorable if the endpoints of any edges (u, v) can be coloured using diff colors when there's k available colours. Given a graph G and k, does G have k-coloring?
  - proof of NP-completeness is reduced from 3-SAT $\to$ that's why 2-colorable $\in P$
- **Hamiltonian Cycle**: A simple cycle is a cycle in a graph with no repeated vertices (a cycle is permutation $\{v_1, v_2, \ldots, v_n\}$ with a pair $v_j = v_k$ but $j \neq k$. Given an undirected graph $G = (V, E)$, can you a simple cycle that visits every node $v \in V$
- **Traveling Salesman**: A tour is a path that starts at city C_1 and visits every city exactly once and ends at C_1 again. Given a set of $\{C_1, C_2, \ldots, C_n\}$, with list of costs where c_{ij} of traveling from C_i to C_j and a number k, is there a tour with costs at most k
- **Subset Sum**: Given a set of natural number $V = \{v_1, v_2 \ldots, v_n\}$ and a number k. Is there a subset $U \subseteq V$ such that sum of U equals k?
- **Set Partition**: Given a set of n integers $V = \{v_1, v_2, \ldots, v_n\}$, can elements of V be partitioned into two sets U and $(U - V)$ such that $\sum_{u \in U} u_i = \sum_{u \in (V-U)} u_i$?

- **special, 3-SAT**: all clauses are of length 3 with n literals, of the form below. Given a SAT instance, could you create a truth assignment $T = \{t_1, t_2, \dots, t_n\}$, $t_i \in \{0,1\}$ that satisfies the instance
  $\longrightarrow$ ex. $(x_1 \vee \overline{x}_2 \vee x_3) \cap (x_4 \vee x_5 \vee x_6)$
  $\longrightarrow$ 2-SAT $\in P$
- **special, clique**: For a graph $G = (V, E)$, a subset of vertices $S \subseteq V$ is a clique if every pair of vertices in V is joined by an edge (so find a complete subgraph basically). Given a graph G and k, does G contain a clique of size at least k?
- **note**: 3-SAT $\leq$ Independent Set $\leq$ Vertex Cover $\leq$ Set Cover

Example Reduction

Independent Set $<_p$ Vertex Cover

1. Vertex Cover $\in$ NP

Given a solution set S, for every vertex in S, delete all adjacent edges from the graph's edge set E. At the end, if E is empty, it was a vertex cover. If we use adjacency lists, runtime is $O(m) \subseteq O(n^2)$ - so it's in poly-time, thus vertex cover $\in$ NP.

2. Reduction:

From diagram below, we can see that independent set and vertex cover problems are complements of each other. So for a graph $G = (V, E)$ with n vertices, and independent set S of size k produces and vertex cover of $(V - S)$ of size $n - k$. So no changes necessary for the graph itself, just pass $k' = n - k$ into `VertexCover(G, k)`

3. Proposition: Set S is an independent set iff its complement $V - S$ is a vertex cover

$\Longrightarrow$: Proceed by contradiction and suppose S is an independent set, yet $V - S$ is not a vertex cover. That means there's an edge $e = (u, v) \in E$ such that neither endpoints are in $(V - S)$ - so $u, v \notin (V - S)$. But then that means that $u, v \in S$, but then S is not an independent set. ■

$\Longleftarrow$: Proceed by contradiction and suppose $(V - S)$ is a vertex cover, but S is not an independent set. So there must be an edge $e = (u, v) \in E$ such that both $u, v \in S$. But that means that $u, v \notin V - S$, so e is an edge with no end point in $(V - S)$ and so $(V - S)$ is not a vertex cover. ■

Hamiltonian Path $\leq$ The Traveling Salesman

1. TSP $\in$ NP

Given a possible solution set of vertices S, check that S is a tour by removing every $v \in S$ from V (if $v \in V$, if $v \notin V$ that means that we've traveled to that city twice, reject) and that $(v_i, v_{i+1}) \in E$. We can also keep track of total cost of every $(v_i, v_{i+1}) \in S$ At the end, V should be empty and the sum of cost should be k. We could check all this in $O(n)$ so TSP $\in$ NP.

2. The Reduction

For an instance $G = (V, E)$ with $|V| = n$ of `HamCycle`, we'll make a new instance I_G. In I_G, we'll turn every node $i \in V$ to corresponding cities C_i in `TSP`. We then can create edges between all cities (make a complete graph, not a requirement in TSP but for our case we want this) make set the weights of $(C_i, C_j) = 1$ if $(i, j) \in E$, otherwise, the cost is 2. We also set the new $k' = n$ (set it to n because we want to reach every node) and feed that into TSP. Creating a new list of cities can be done in $O(n)$ time and if we're using adjacency lists for the edges, we can set all edges to 2 in $O(n^2)$ and change all edges that exist in E to 1 in also $O(n^2)$ times. So total time of reduction is $O(n^2)$

3. Proposition: Show that your reduction is correct, that is G is a Yes-instance of HamCycle iff I_G is a Yes-instance of TSP

$\Longleftarrow$: Suppose that I_G is a Yes-instance of TSP, so there's a tour of the cities that cost at most $k = n$. Let $\{C_{i1}, C_{i2}, \dots, C_{in}\}$ be successive cities on this tour. Then the cost is 1 to get from any city to the next conseuctive city in the tour, and also the cost is 1 to get back from the last to the first (if any of the cost was 2, the total would be at least $n+1$). The reduction forces the fact that inter-city costs are 1 iff there's an edge in E between the corresponding nodes in G, so E must contain (i_n, i_1) as well as (i_j, i_{j+1}) for $1 \leq j \leq n - 1$. So if we take the corresponding nodes, the permutation $\{i_1, i_2, \dots, i_n\}$ is a Hamiltonian cycle of G and G is a Yes-instance

$\Longrightarrow$: Suppose G is a Yes-instance, with Hamiltonian cycle $i_1, i_2, \dots, i_n$. Then the reduction guarantees that Cities $C_{i1}, C_{i2}, \dots, C_{in}$ form a tour where the cost from one city to the next is 1, and the cost of getting back from the last city to the first is also 1. So, I_G has a tour of cost n and so is a Yes-instance of the TSP problem

Subset Sum $\leq$ Set Partition

1. Set Partition $\in$ NP

Given sets A and B, to check that they are partitions of U, we need to check $A \cup B = U$, $A \cap B = \varnothing$ and $\sum_{a_i \in A} a_i = \sum_{b_i \in B} b_i$. To check first requirement (the union), add A and B to a set and check if that's equal to $U \longrightarrow$ would take $O(n \log n)$ because need to sort the sets to compare. For the second, put elements of A into hashsets and check every element of B to see if it's already in there. Thirdly, we can compute the sum as we go along doing the last step $\longrightarrow$ both these things take $O(n)$. So checking takes $O(n \log n) \subset O(n^2)$. Thus Set Partition $\in$ NP.

2. The Reduction

The intution is that for a set V, $\sum_{v_i \in V} v_i = s \in \mathbb{Z}$. If we successfully partition V into U and $U - V$, then $\sum_{u_i \in U} u_i = \sum_{u_i \in (V-U)} u_i = s/2$. Then, if we wanted it look for a partition that sum to k, we need all elements in V to sum to $2k$ - can do this by adding a integer t to V with value $(2k - s)$.

So, given a set V, for a set $V' = V \cup \{2k - s\}$ and feed V' into `SubsetCover(V)`. This will return 2 subsets, both of which sum to k, pick the one without $2k - s$. The reduction takes polynomial time because you're just adding an element.

3. Proposition: Let $V = \{v_1, v_2, \dots, v_n\}$ and $k \in \mathbb{Z}$. **Prove that an instance is Yes instance in the Subset Sum problem iff it's a Yes instance in Set Partition**

$\Longrightarrow$: Suppose that I is a Yes-instance in `SubsetSum`. This means that there's a subset $W \subseteq V$ such that $\sum_{w_i \in W} w_i = k$. Let $\sum_{v_i \in V} v_i = s \in \mathbb{Z}$ and let $V' = V \cup \{2k - s\}$, we will feed V' into `SetPartition`. Consider sets W and $V' - W$, we need to show that these 2 sets are the partition of V' that will give us the right answer.
First, we have $W \cup (V' - W) = V'$ and $W \cap (V' - W) = \varnothing$ via definition of complement. Since $\sum_{v_i' \in V'} v_i' = s + (2k - s) = 2k$ and $\sum_{w_i \in W} w_i = k$, this must mean $\sum_{w_i \in (V'-W)} w_i$. Therefore, we have $\sum_{w_i \in W} w_i = \sum_{w_i \in (V'-W)} w_i$ as required. So we have a Yes instance in `SetPartition` as well. ■

$\Longleftarrow$: Suppose there's a partition of V', U and $V' - U$. Because of the nature of V' and set partition, we know $\sum_{u_i \in U} u_i = \sum_{u_i \in (V'-U)} u_i = k$. We also have the fact that they are disjoint, so $2k - s$ belongs to just one of the sets. WLOG, assume that set is $V' - U$, then U is the subset of V that we're looking for. And so we have a Yes instance in `SubsetSum` as well