1 Problem 1

- `Optimization` problem:
  find the minimum integer b such that there exists $B \subset V$ of size $|B| = b$ satisfying: $\forall v \in V \setminus B \exists u \in B : (u, v) \in E$
- `Decision` problem:
  given an integer b, decide whether there exists $B \subset V$ of size $|B| = b$ satisfying: $\forall v \in V \setminus B \exists u \in B : (u, v) \in E$

1.1 A Polynomial Reduction from Optimization to Decision

`Optimization` $\leq_{\text{p}}$ `Decision` algorithm

1. Suppose that $G = (V, E)$ is given with n vertices and m edges
2. Iterate through possible b's from 1 to $|V| = n$ with an increment size of 1.
3. Use b as the given integer in the decision problem and query the decision black box. The first b, making the decision problem answer yes, is the minimum integer asked in the original(optimization) problem.
4. If there's no b that `Decision` returns 'yes' after the iteration in step 2 ends, there's no such a minimum integer.

1.2 Correctness and Time Complexity

1.2.1 Correctness

- Optimization $\Rightarrow$ Decision
  Trivially feed the minimum integer b found in the optimization problem to the decision problem as the given integer input b. The decision algorithm must return true as a concrete B has been found in the optimization problem. Otherwise it leads to a contradiction. Similarly, if there's no such min integer $b = |B|$ found in the optimization problem, the decision problem won't find any B for any given integer $b \in [1, |V|]$.
- Decision $\Rightarrow$ Optimization
  If the first b making the decide procedure return true is not the minimum, in other words, there's an integer i smaller than b given to the decision problem where there is a $|I| = i$ satisfying, it raises a contradiction with the iteration procedure as i should be the first integer found instead of b. Therefore, the first integer b in the iteration cycle fed to the decision procedure in which there is such a B satisfying, must be the minimum.

1.2.2 Running time of the Reduction

An instance of the optimization problem is converted to $n = |V|$ instances of the decision problem. Iterating from 1 to n is of $O(n)$, a linear time, of course, is of a polynomial time.

1.3 Decision problem is in $\mathcal{NP}$

- **certificate**: A subset B of V
- **certifier**: Feed G, b and B to the certifier.Then the certifier performs the following algorithm
  1. check size of B satisfying $|B| = b$
  2. iterate through all nodes $v \in V \setminus B$, check if there is a neighbor u of v that $u \in B$. Return 'no' there's no such a neighbor found.
  3. the iteration is done(not halted before all edges are scanned througd),return 'yes'
- **time**: Step 1 runs in $O(1)$. Suppose we implement the data structure as adjacent lists, the step 2 runs in $O(|V|^2)$ for the worst scenario. Step 3 is of $O(1)$. Therefore the certifier is a overall polynomial running time algorithm.
- **correctness**:
  - certificate $\Rightarrow$ certifier
    If the subset B doesn't have the properties that $|B| = b$ and $\forall v \in V \setminus B \exists u \in B : (u, v) \in E$, the certifier halts before the whole iteration has been visited.
  - certifier $\Rightarrow$ certificate
    If the certifier returns 'yes', the input $|B|$ has been verified that it has the 2 properties just mentioned. Otherwise, B loses one of the properties at least thus is not a valid certificate.

1.4 Decision problem is $\mathcal{NP}$-Complete

Since we have proved that `Decision` is in $\mathcal{NP}$, now we only need to find a $\mathcal{NP}$-complete algorithm `Y` such that `Y` $\leq_p$ `Decision` to prove that `Decision` is $\mathcal{NP}$-complete.
A good candidate for `Y` is `Vertex Cover`, which has been proven to be $\mathcal{NP}$-complete in the lecture. Now we only need to show that `Vertex Cover` $\leq_p$ `Decision`. Let's compare these 2 problems firstly.

- `Decision`:
  given an integer b(also the G), decide whether there exists $B \subset V$ of size $|B| = b$ satisfying: $\forall v \in V \setminus B \exists u \in B : (u, v) \in E$
- `Vertex Cover`:
  given G and an integer k, decide whether there exists $S \subset V$, where $|S| \leq k$ such that: $\forall e := (u, v) \in E : u \in S \vee v \in S$.

These two problems are intuitively to be different forms of the same problem except that `Vertex Cover` has an undirected G while `Desicion` uses a directed one, nevertheless they have a similar structure of the constraints. For now, let's simply convert an instance of `Vertex Cover` $VC(G, k)$ to an instance of `Decision` $D(G', b)$ by setting b to the sum of k and the number of the edges for the graph in `Vertex Covel` ($b = k + m(|E|)$) and convert G to a directed G'. Here are steps to convert G to G'.

1. copy all nodes from G to G'
2. for each edge (u, v) in G, add a dummy node w on this edge and add a pair of directed edge for u and v. In other words, $\forall (u, v) \in G(E)$, we add a new node w to $G'(V')$ and edges $(u, w), (w, u), (w, v), (v, w)$ to $G'(E')$

After the reconstruction, $G'(V', E')$ has $|V| + |E|$ vertices and $4|E|$ directed edges. Now we attempt to prove that the solutions to two instances are bijective.

- 'yes' for $VC(G, k) \Rightarrow$ 'yes' for $D(G', k + |E|)$
  'yes' for $VC(G, k)$
  $\Rightarrow \exists S \subset V : |S| \leq k \wedge \forall (u, v) \in E : u \in S \vee v \in S$
  Because the way we constructed G' we also have,
  $\forall u' = u, v' = v \in V' : \exists w \in V' : (w, u') \wedge (w, v') \in E'$.
  Therefore, $\forall v' = v \in V'$ where $v \in S : \exists w \in V' : (w, v') \in E' \wedge (w, u' (\neq v')) \in E'$.
  Let's say we put all dummy nodes w's in G' and $v' = v \in S$ into a set W, we have:
  $\forall v' \in V' \setminus W : \exists w \in W : (w, v') \in E'$ and $|W| \leq k + |E|$
  Now we move some nodes from $V' \setminus W$ to W to make a new set B so that $|B| = k + |E|$ while we still keep the property
  $\forall v' \in V' \setminus B : \exists w \in B : (w, v') \in E'$
  $\Rightarrow$ 'yes' for $D(G', k + |E|)$
- 'yes' for $D(G', k + |E|) \Rightarrow$ 'yes' for $VC(G, k)$
  'yes' for $D(G', k + |E|)$
  $\Rightarrow$ There exists a node set B of size $k + |E|$ and $\forall v' \in V' \setminus B : \exists w \in B : (w, v') \in E'$
  Because of the way G' constructed, we can always swap dummy nodes ($w \in V' \wedge w \notin V$) in $V' \setminus B$ to non dummy nodes($v \in V \wedge v \notin V'$) in B while B still has the properties above.
  $\Rightarrow$ There exists a node set B of size $k + |E|$ and $\forall v' \in V' \setminus B : \exists w \in B : (w, v') \in E'$ and B contains all dummy nodes.
  Now we neglect all dummy nodes from B as B' where
  "there exists a node set B' of size k and $\forall v' \in V \setminus B' : \exists u' \in B' : (u', v') \in E$" still holds because B' is a subset of B and all nodes in B' are also in V.
  $\Rightarrow B'$ is a vertex cover for G and $|B'| = k$
  $\Rightarrow$ 'yes for $VC(G, k)$

The equivalent mapping of 'yes's have been proved. The running time is of a polynomial function as converting the graph G to G' costs $O(|E| + |V|)$(linear time for generating $|V| + |E|$ nodes and $4|E|$ edges) and assigning b costs $O(1)$. Therefore, `Vertex Cover` $\leq_p$ `Decision`. With `Decision` $\in \mathcal{NP}$ and `Vertex Cover` $\in \mathcal{NP}$-complete have been proven, we conclude that `Decision` is in $\mathcal{NP}$-complete.

1.5 Optimization problem is $\mathcal{NP}$-Hard

As we have shown that the corresponding decision problem of it is in $\mathcal{NP}$-Complete, it is $\mathcal{NP}$-Hard. It is straightforward to prove, assuming that $\mathcal{P} \neq \mathcal{NP}$. Suppose the

optimization problem is not $\mathcal{NP}$-Hard, in other words, we can solve `Optimization` by an algorithm of a polynomial running time. Then by a converse way of the reduction in section 1.1,
`|Decision|`$\leq_p$ `Optimization`:
simply call the optimization black box and compare the min size $b_{\min}$ returned with the given input b in the decision problem.
If $b < b_{\min}$, simply return 'no'. Otherwise, return 'yes'.
, we have constructed a way of solving the decision problem in a polynomial time, as we use a single call of the `Optimization`(hypothetically supposed to be a polynomial algorithm) and compare. We also need to prove the correctness of the reduction before inferring further:

- `Decision` $\Rightarrow$ `Optimization`
  If `Decision` of b returns 'yes', $b >= b_{\min}$ of `Optimization`. Otherwise($b < b_{\min}$), we have found a valid subset B of size b smaller than $b_{\min}$, which means $b_{\min} = b$ and causes a contradiction. Thus, the 'yes' instances are consistent.

- `Optimization` $\Rightarrow$ `Decision`
  If $b >= b_{\min}$, we know that there's a valid subset $B_{\min}$ of size $b_{\min}$. Based on $B_{\min}$, we can always construct a subset B, where $|B| \geq |B_{\min}|$, by removing some of the nodes from $V \setminus B_{\min}$ and add these nodes to $B_{\min}$ to form a new subset B. As all nodes v in $V \setminus B_{\min}$ has a property that $\exists u \in B : (u, v) \in E$, the nodes in a subset $(V \setminus B)$ of $V \setminus B_{\min}$ still keep this property. Therefore such a B of size b exists, in other words, `Decision` given b returns 'yes'.

Therefore, `Decision` is in $\mathcal{P}$, as we have polynomially reduced it to the oracle `Optimization` and the oracle is supposed to run in a polynomial time, which raises a contradiction of the conclusion in section 1.4(`Decision` $\in \mathcal{NP}$-complete $\Rightarrow$ we have found a way to solve a $\mathcal{NP}$-complete in a polynomial time. $\Rightarrow \mathcal{P} = \mathcal{NP}$). By the contradiction(Given that `Decision` $\leq_p$ `Optimization` and `Decision` is $\mathcal{NP}$-complete), we have proven that `Optimization` is not possible to finish in a polynomial time, i.e. $\mathcal{NP}$-hard. And of course, it is $\boxed{\text{not } \mathcal{NP}\text{-complete}}$ simply because it is not even a decision problem.

2 Problem 2

2.1 Size of the input in bits

- indices of planets(objects): n
  and for each object there are such attributes:
  - total distance to other planets within the same quadrant: $\delta_l \leq \Delta$
  - base building cost: $c_l \leq D$
  - quadrant index: k
- total cost capacity: D

Therefore, loosely upper bounded and totally, the input requires $n(\Delta + D + k)^n D$. In bits, it is bounded by $\boxed{\log n + n \log (\Delta + D + k) + \log D}$

2.2 Decision problem

`Search`:
Given n as well as all the Q_i's, δ_i's, c_l's, and D (where $c_l \leq D$ for every l), find one planet $l_i \in Q_i$ for each quadrant Q_i, such that $\sum_{i=1}^{k} c_{l_i} \leq D$ while minimising $\sum_{i=1}^{k} \delta_{l_i}$.

A decision version of the search problem is to check the existence of an instance satisfying the requirements in the search problem. Therefore,
`Decision`:

Given n as well as all the Q_i's, δ_l's, c_l's, D (where $c_l \leq D$ for every l) and a threshold integer Δ, does there exist one planet $l_i \in Q_i$ for each quadrant Q_i, such that $\sum_{i=1}^{k} c_{l_i} \leq D$ while $\sum_{i=1}^{k} \delta_{l_i} \leq \Delta$.

2.3 Decision in $\mathcal{NP}$

- **certificate**:
  k planets (l_i's, $i \in [1, k]$) with their corresponding attributes(Q_i, δ_l and c_l).
- **certifier**:
  The certifier is feed the inputs in `Decision`, together with the certificate. It performs such a procedure:
  1. check whether each planet in the certificate is in a unique disjoint quadrant
  2. check the sum of the base building costs of the planets in the certificate is not larger than D
  3. check the sum of the total space distances of the planets in the certificate is not larger than Δ

  If any validation step above is not fulfilled, return false. Otherwise return yes.
- **running time**:
  1. In step 1, suppose we implement a single linked list to store each planet's quadrant index(a hash table would be better but I am just lazy). For each planet in the certificate, if its quadrant index is not in the list, apppend the index to the end of the list. Otherwise it means there are two planets from the same quadrant. This runs in $O(k^2)$ as $O(k)$ is the worst case for iterating through the list for each planet.
  2. Step 2 and step 3 performs arithmetic operations for k planets and a comparison operation. Both are in $O(k)$

  In total, the certifier runs in $O(k^2)$ which means it is a polynomial time verifier.
- **correctness**: certificate $\Leftrightarrow$ certifier
  Both directions are trivial as the certifier makes sure all properties of the certificate fulfilled. The certifier returns true if and only if the certificate is a combination of valid planets.

2.4 Decision in $\mathcal{NP}$-complete

Since we have just proved that `Decision` is $\mathcal{NP}$, we need to reduce from a well known $\mathcal{NP}$-complete problem to `Decision` so that we prove `Decision` is $\mathcal{NP}$-hard as well. `Knapsack` problem is a good candidate as we have proven it $\mathcal{NP}$-hard and there's a capacity restriction similar to the one we have in our `Decision` problem while we need to achieve some optimality. To simply the process, we use the decision variant of `Knapsack` problem, denoted as `Knapsack Decision` which we assume as a known $\mathcal{NP}$-complete problem. Here is a reminder of what `Knapsack Decision` problem is.
`Knapsack Decision`:
Given a set of items $I = \{1, 2, ..., n\}$, each item i has a weight w_i and a value v_i, a knapsack capacity C and a target value T, does there exist a subset $S \subseteq I$ such that the total weight of items in S does not exceed the knapsack capacity ($\sum_{i \in S} w_i \leq C$) and the total value of items in S is at least the target value ($\sum_{i \in S} v_i \geq T$).

- `Knapsack Decision` $\leq_{\text{p}}$ `Decision` algorithm
  The basic idea is to think of base building costs as weights and total space distances as values. Two subtle differences: one is that `Desicion` requires the sum of total space distances(values) at most ($\geq$) Δ while `Knapsack Decision` filters the sum of values at least ($\geq$) T. This can be solved by nagating values. The other, is that, `Knapsack` needs a subset of a set of items(planets) while `Decision` splits the set(of planets) into k disjoint subsets and then picks one item from each subset to form the required target subset. This can be solved by spliting the set(of items) into a new set of exactly n(number of items in the set) sets where there's only one distinct item. However, `Decision` selects one item from each set. So we need to add a dummy item to each set so that we can mimic selecting a subset of the original set of items. Here are steps to convert an instance of `Knapsack Decision` to that of `Decision`.
  Given an instance of `Knapsack Decision`: a set of items $I = \{1, 2, ..., n\}$, each item i has a weight w_i and a value v_i, a knapsack capacity C and a target value T, we perform such operations:

  1. negate all values ($v'_i = -v_i, i \in I$)
  2. assign each item in I, i to a planet l_i and duplicating item's fields' information to planets' attributes: w_i to c_{l_i} and v'_i to δ_{l_i}
  3. assign $2n$ (n is the length of I, we need double it since we have n dummy planets) as input n for `Decision`, $\{l_i, l'_i\}$, where l'_i is a dummy planet with $\delta'_{l_i} = 0, c'_{l_i} = 0$, as Q_i, C as D, $-T$ as Δ.

  Now we have an instance input($2n$, Q_i's, δ_l's, c_l's, D and Δ) for our `Decision` problem, specifically where, $k = n = |I|, \forall Q_i : |Q_i| = 2, |\{Q_i\text{'s}\}| = 2n$.

- reduction algorithm running time
  - Step 1 and Step 2:Negation and assignment operations are in $O(1)$. There are n items to perform these operations upon. The overall running time is a linear time $O(n)$.

  - Step 3 costs $O(n)$ for creating dummy planets and aggregating them into Q_i. Other assignments cost $O(1)$.

  The overall running time is of $O(n)$, of a polynomial time.

- reduction algorithm correctness
  As I have mentioned the 2 conversion obstacles in the reduction algorithm design, I would briefly argue that the 'yes' instances for `Knapsack Decision` and `Decision` have a bijective mapping.

  - mapping between I and $\{\{l_i, l_i'\} = Q_i\}$ is bijective, where fields for $i \in I$ and fields for l_i are bijective as well. This means $\sum_{i \in S} w_i = \sum_{i=1}^{|S|} c_{l_i} = \sum_{i=1}^{|S|} c_{l_i} + \sum_{j=1}^{n-|S|} c_{l_j'}$ because $c_{l_j'} = 0$. Besides $D = C$, the 'yes' instances for two problems must be consistent as they check the same inequality. This also applies to the sum of values.
  - specially, $\sum_{i \in S} v_i \geq T \Leftrightarrow \sum_{i \in S} v_i' \leq -T \Leftrightarrow \sum_{i \in S} v_i' \leq \Delta$
    $\Leftrightarrow \sum_{i=1}^{|S|} \delta_{l_i} + \sum_{j=1}^{n-|S|} \delta_{l_j'} \leq \Delta$

  Therefore, true for `Knapsack Decision` $\Leftrightarrow$ true for `Decision`

We have found a vaild reduction algorithm in polynomial time from `Knapsack Decision` to `Decision`. With that `Decision` is $\mathcal{NP}$ and `Knapsack Decision` is $\mathcal{NP}$-complete, we conclude `Decision` is $\mathcal{NP}$-complete.

2.5 Search in $\mathcal{NP}$-hard

Here we won't concretely prove that `Decision` $\leq_p$ `Search` like section 1.5 (But I will give an idea to do so. We can construct a `Search` instance base on a `Decision` instance by keeping all inputs except Δ. We then call `Search` oracle to get its output. Then we compare the sum of total space distances of the output planets with Δ). Instead, we apply a theorem from the lecture that `Decision` $\equiv_p$ `Optimization` $\equiv_p$ `Search` $\Rightarrow$ `Decision`$\leq_p$ `Search`. With `Decision` is $\mathcal{NP}$-complete proven in section 2.4, `Search` is $\mathcal{NP}$-hard.

2.6 Pseudo polynomial dynamic programmming

Because we have proved that `Knapsack Decision` $\leq_p$ `Decision`, it is reasonable to solve `Search` using a common technique Dynamic Programming for classical `Knapsack` problems. But in our problem, we need to select 1 base planet from each quadrant. Therefore, we borrow the DP from the lecture and modify it accordingly.

- Subproblem
  $OPT(i, C)$: min sum of the total space distances of planets selected in Q_1, ..., Q_i with the space dollar budget C.

- Recurrence Cases

  - if no planet selected in Q_i: $OPT(i, C) = OPT(i-1, C)$

  - if a planet l_{ij} is selected in Q_i: $OPT(i, C) = OPT(i-1, C - c_{l_{ij}}) + \delta_{l_{ij}}$

- Base cases
  $OPT(i, C) = 0$ when $i = 0$ which means no planet selected at all, sum of the total space distances is 0. The space dollars D is not spent at all.

- Summary

$$OPT(i,C) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,C) & \text{if } \forall l_i \in Q_i : c_{l_i} > C \\ \min\{OPT(i-1,C), \min\{OPT(i-1, C - c_{l_{ij}}) + \delta_{l_{ij}}\}\} & \text{otherwise} \end{cases}$$

Then, we can implement a concrete dynamique programming algorithm by using the recursions defined above.

1. Initialize the DP matrix OPT of size $(k+1) \times (D+1)$ with o's and a tracing matrix T with the same dimensions with placeholders(None objects)

2. Iterate through k quadrants. For each quadrant iterator Q_i:

   2.1. iterate through D budget integers. For each budget integer$C$ use the recursions:

   2.1.1. iterate through planets l_i's in the current quadrant Q_i.
   - If the planet building cost of a planet $c_{l_i} \leq C$, update $OPT[i][C]$ as $\min\{OPT[i-1][C], OPT[i-1][C - c_{l_{ij}}] + \delta_{l_{ij}}\}$. If the later one is used, store the planet at $T[i][C]$.
   - If all planets within Q_i have building cost larger than C(after iterating all l_i's), update $OPT[i][C]$ as $OPT[i-1][C]$

3. backtracing the T matrix from $i = k, C = D$ and halts when i or C is o to get the output for the search problem, decreasing C. If there's a planet object stored, it's a planet l built as a base and we check $T[i-1][C - c_l]$. If for i there's no solution stored for any C, it means there's no solution. We return k such stored planets otherwise there's no way to use D total dollars to build.

2.7 Correctness

The algorithm is built upon the recursive relations defined in "Summary" section in 2.6. Therefore, we analyze the correctness of the recursion instead. The base case is trivially correct as when $i = 0$ it means no planet selected at all, sum of the total space distances is 0. The recurrence is built upon its subproblem which is considering 1 less quadrant. Therefore, either there's a planet in the new added quadrant which costs less than the remaining budget C or all planets in the new quadrant need more budgets than C. Sufficiency is thus also satisfied. We always use min operations to filter the current achivable optimality, thus if there's a combination of building k planets with totol cost less than D, the sum of the distances is minimized.

2.8 Time Complexity

- Initialization DP matrix costs $O(1)$
- For updating DP matrix, loosely assume that each quadrant has n planets, iterated C integers, and iterated k quadrants. It costs $O(nkD) \in O(n^2D)$ as $k \leq n$.
- backtracing costs $O(k + D)$ as i and C doesn't increase(goes towards the $T[0][0]$ corner only).

The overall total running time is $\boxed{O(n^2D)}$.