1. Let the result of k_{th} iteration be S_k. Assume that there's an optimal solution containing S_k. For any hw out of Γ(S), the addition of it will make exceed the time limit. When we go through the $k+1_{th}$ iteration, the biggest element in Γ(S) is chosen. Let it be x. If there was no optimal solution that contains x, the the optimal solution would contain other elements smaller than x. As all elements are power of 2. If the sum of the elements chosen after k_{th} iteration in the optimal solution is bigger then x, the we can exchange part of the elements, whose sum is equal to x, with x, which is a contradiction. If the sum is smaller than x, then it's not optimal because choosing x makes a bigger S. That is to say, S_{k+1} is a subset of an optimal solution. Also, $S_0 = \emptyset$ is a subset of any optimal solution, so it is proved that we can get an S with the max reward through the algorithm.
2. 1. Assume that the algorithm gives a result smaller than half of the optimal, then the remaining time is longer than any of the hw that have been chosen. As the hw are sorted in a descent order, all the remaining hw taking time less than or equivalent to the smallest chosen hw, meaning that we can add at least one hw to the result, so there's a contradiction and it is proved that the algorithm gives a result more than or equal to 0.5 of the optimal.
   2. Time limit 2n, hw time cost n+1,n,n. Then the algorithm gives n+1, and the optimal is 2n. $lim \frac{n+1}{2n} = 0.5$
   3. Assume that the algorithm gives a result S smaller than 2/3 of the optimal, then the remaining time is longer than 1/3 of the total time and at least one subset smaller than 2/3 of optimal has been chosen. So there're remaining subsets of size 1or 2 are smaller than 2/3 of optimal. If they are all of size 1, then there will be hw that are smaller than 1/3 of the optimal because they are not chosen in former selections. If there's size 2 subset, then at least one hw of such a subset will be smaller than 1/3 of optimal. That is to say, we can always add one hw to S, thus S is not the end of the algorithm. So S always gives a result larger than 2/3 of the optimal.
3. 1. If two maximal sets A,B , |A|<|B|,then there exists x∈B/A and *A* ∪ {x} ∈ I. So A⊆*A* ∪ {x} and A is not a maximal set. So it is proved that maximal sets are of the same size.
   2. For any F ⊆ E (F does not contain a cycle), for any A⊆F, A does not contain a cycle, so A∈S. It satisfies hereditary property.

      For any A, B ∈ S with |A| < |B|, there's a vertex not connected by A, so there exists some x ∈B \ A such that it connects A to the unconnected vertex and creates no cycle, in other words, A ∪ {x} ∈ S. It satisfies exchange property.

      The maximal sets are MSTs.
   3. ∅ ∪ {x} = {x} ∈ I

      If there's no A∈ I with |A|>1, then {x} is the maximal independent set with maximum weight.

      Else, because of the exchange property, S can always grow until it's size of a maximal independent value. So there's maximal independent set containing x.

      Assume that there's no maximum weight S' containing x. Take one maximum weight maximal set A. Every time we delete one element(except x) that is in S'/A from S' to make an S'' , and according to exchange property there must be an element y∈ A/S' such that S'∪{y} ∈I. We can do this iteratively until we get an S' that has only one element x different than A, and S' is a maximal set with a bigger weight than A, which means that A is not a maximum weight maximal set.

So there must be maximum weight maximal set S' containing x.

4. If for the k_{th} iteration chosen set S_k, it is not a maximal set ,and there's a maximum weight maximal set S, consisted of elements satisfying S ∪ {x} ∈ I only, satisfying that $S_k \subseteq S$. Let the $k+1_{th}$ chosen element be x, if x is not in S, we can get a maximum weight maximal set by finding a maximal set containing x and change it in the way elaborated in last problem into a set that is just a maximum maximal set S' satisfying $S_{k+1} \subseteq S$. So we can for the last step, we will get exactly a maximum weight maximal set.

5. Let I be the set of valid S.

   Algorithm:

   1. S← ∅
   2. Sort *U* into decreasing order by weight *w*
   3. Recursion: For x ∈ U in decreasing order of w, if S∪ {x} ∈ I, S ← *S* ∪ {x}. End if |S|=n.
   4. Return S as result

   Proof:

   For M(U,I) at least the one vector sets are valid, so I is not empty. For any valid S, its subset is also linearly independent, so it satisfies hereditary property. For A,B ∈ I with |A|<|B|, there's at least one vector in B\A such that A combined with the vector is still linearly independent, otherwise B won't be linearly independent, so exchange property is satisfied. So M(U,I) is a matroid, and we can use Algorithm 1 to find S with maximum weight.

   Time complexity:

   The sorting takes $O(n) = nlogn$. The recursion takes O(n)=n, as it goes through U for once.

   So totally it's $O(n) = nlogn$.

4. Define a potential functionΦ, as the number of 1's in the binary representation of the integer. Initially, when the integer is zero, Φ = 0.

   When we perform an ADD operation, we need to consider two cases:

   1. If the least significant bit of the integer is 0, then we can simply flip it to 1, and the operation is done. This requires only one bit operation, and the potential function increases by 1. Therefore, the amortized cost of this operation is 1 + 1 = 2.
   2. If the least significant bit is already 1, we need to find the first 0 bit starting from the least significant bit and flip it to 1. This requires flipping all the 1 bits before it to 0, which can take several bit operations. However, when we flip the 0 bit to 1, the potential function increases by 1, canceling out the cost of flipping all the 1 bits. Therefore, the amortized cost of this operation is the actual cost plus the change in potential function, which is the same as the cost of flipping the 0 bit, which is k+1, where k is the number of 1 bits before the first 0 bit.

   Now, let's consider the worst-case scenario, where we perform n ADD operations, and the integer is of the form $2^m - 1$(i.e., all bits are 1). In this case, each ADD operation requires flipping all the bits to 0 and then flipping the least significant bit to 1, which takes 2m bit operations. However, the potential function also increases by 1 each time, so after n operations, the potential function is n, and the total cost is at most 2mn + n. Therefore, the amortized cost of each operation is (2mn + n)/n = 2m + 1, which is O(1) since m is a constant for fixed-size integers.

In conclusion, we have shown that the amortized cost of ADD is O(1) using the potential method of amortized analysis, with the potential function Φ = number of 1 bits in the binary representation of the integer.