Question 1

(a) $x = X_2 * 2^{2n/3} + X_1 * 2^{n/3} + X_0$

(b)
```
Multiply(x, y) {
```
let $x = x_2 * 2^{2n/3} + x_1 * 2^{n/3} + x_0$
let $y = y_2 * 2^{2n/3} + y_1 * 2^{n/3} + y_0$

```
let z = [0..6] #Array of size 6
```
$z[0] = Multiply(x_2, y_2)$
$z[1] = Multiply(x_1 + x_2, y_1 + y_2)$
$z[2] = Multiply(x_1, y_1)$
$z[3] = Multiply(x_0 + x_2, y_0 + y_2)$
$z[4] = Multiply(x_0, y_0)$
$z[5] = Multiply(x_0 + x_1, y_0 + y_1)$

```
let product =
```
$z[0] * 2^{4n/3} +$
$(z[1] - z[0] - z[2]) * 2^n +$
$(z[3] - z[4] - z[0] + z[2]) * 2^{2n/3} +$
$(z[5] - z[4] - z[2]) * 2^{n/3} +$
$z[4]$

```
return product
}
```

Justification for my algorithm: The return value, with the elements of the array z replaced, is mathematically equal to:

$x_2y_2 * 2^{4n/3} +$
$[(x_1 + x_2)(y_1 + y_2) - x_2y_2 - x_1y_1] * 2^n +$
$[(x_0 + x_2)(y_0 + y_2) - x_0y_0 - x_2y_2 + x_1y_1] * 2^{2n/3} +$
$[(x_0 + x_1)(y_0 + y_1) - x_0y_0 - x_1y_1] * 2^{n/3} +$
x_0y_0

So our goal is to prove that xy is equal to the return value. We begin by multiplying xy out directly, then we rewrite the resulting equation by combining like-terms, and finally we reduce the number of unique subproblems.

Multiply:
let $x = x_2 * 2^{2n/3} + x_1 * 2^{n/3} + x_0$
let $y = y_2 * 2^{2n/3} + y_1 * 2^{n/3} + y_0$

$$xy = x_2y_2 * 2^{4n/3} + x_2y_1 * 2^n + x_2y_0 * 2^{2n/3} + x_1y_2 * 2^n + x_1y_1 * 2^{2n/3} + x_1y_0 * 2^{n/3} + x_0y_2 * 2^{2n/3} + x_0y_1 * 2^{n/3} + x_0y_0$$

Combine:

$$\begin{aligned} xy = {} & x_2y_2 * 2^{4n/3} \ + \\ & (x_2y_1 + x_1y_2) * 2^n \ + \\ & (x_2y_0 + x_0y_2 + x_1y_1) * 2^{2n/3} \ + \\ & (x_1y_0 + x_0y_1) * 2^{n/3} \ + \\ & x_0y_0 \end{aligned}$$

To reduce the number of unique subproblems, we make the following observations:

$$(x_1 + x_2)(y_1 + y_2) = x_1y_1 + x_2y_1 + x_1y_2 + x_2y_2$$
$$\therefore (x_1 + x_2)(y_1 + y_2) - x_2y_2 - x_1y_1 = x_2y_1 + x_1y_2$$

$$(x_0 + x_2)(y_0 + y_2) = x_0y_0 + x_2y_0 + x_0y_2 + x_2y_2$$
$$\therefore (x_0 + x_2)(y_0 + y_2) - x_0y_0 - x_2y_2 = x_2y_0 + x_0y_2$$

$$(x_0 + y_1)(x_0 + y_1) = x_0y_0 + x_1y_0 + x_0y_1 + x_1y_1$$
$$\therefore (x_0 + x_1)(y_0 + y_1) - x_0y_0 - x_1y_1 = x_1y_0 + x_0y_1$$

Finally, in our equation for xy, we replace $x_2y_1 + x_1y_2$, $x_2y_0 + x_0y_2$, and $x_1y_0 + x_0y_1$ with their respective equivalent expressions.

$$\begin{aligned} xy = {} & x_2y_2 * 2^{4n/3} \ + \\ & [(x_1 + x_2)(y_1 + y_2) - x_2y_2 - x_1y_1] * 2^n \ + \\ & [(x_0 + x_2)(y_0 + y_2) - x_0y_0 - x_2y_2 + x_1y_1] * 2^{2n/3} \ + \\ & [(x_0 + x_1)(y_0 + y_1) - x_0y_0 - x_1y_1] * 2^{n/3} \ + \\ & x_0y_0 \end{aligned}$$

Thus, we see that xy is indeed equivalent to the return value of our algorithm, therefore proving correctness.

The running time of our algorithm can be computed using the Master Theorem because our algorithm is split into a number of similar subproblems through the recursive application of the $Multiply()$ function.

First, note that we divide the algorithm into 6 subproblems of size $n/3$ each. The subproblem size is based on the fact that x_2, x_1, x_0 and y_2, y_1, y_0 are each a third of the bits in x and y respectively. Also we assume that bitshifting is constant and that bitshifting takes place whenever we multiply a value by 2^a, where a is some integer value. To account for addition, we assume that the rest of the algorithm runs in O(n) time.

Thus, we have $T(n) = 6T(n/3) + O(n)$.

Since $O(n^{\log_3 6}) > O(n)$, by Master Theorem, the time complexity is $O(n^{\log_3 6})$, which is strictly less than $O(n^2)$ since $\log_3 6 \approx 1.631 < 2$.

(c) Our algorithm is slower than the divide-and-conquer algorithm with a running time of $\Theta(n^{log_2 3}) \approx \Theta(n^{1.585})$ since $\log_3 6 \approx 1.631$. Our algorithm has a worse upper bound of $O(n^{1.632})$.

Question 2

(a) For the input, since input is a set of processes, then define Pro(n) to be a set of processes of n processes.
Pseudocode:

```
Minprocessor(Pro(n)):
    sort all processes by their start time in ascending order
    M = ∅
    i = 0
    assigned = 0
    for j in Pro(n):
        for k in M:
            if s_j is available in processor k:
                assign process j to processor k
                assigned = 1
                break loop
        if assigned == 0:
            i += 1
            assign process j to processor i
            add processor i to set M
        assigned = 0
    return length(M)
```

(b) Proof:
Base case: n = 0
Minprocessor(Pro(0)) = 0, which is the minimum number of processors.
n = 1
Minprocessor(Pro(1)) = 1, which is the minimum number of processors.
Induction Hypothesis: Assume $\forall k, 1 \leq k \leq n$, Minprocessor(Pro(k)) returns the minimal number of processors required Min(k).
Induction Step: we want to proof Minprocessor(Pro(n + 1)) also returns the minimal number of processors.

Case 1: the n+1 process can be added to a processor in set M
Then Min(n + 1) = Min(n)
From the algorithm, we get Minprocessor(Pro(n + 1)) = Minprocessor(Pro(n))
Since Minprocessor(Pro(n)) = Min(n) by induction hypothesis
Then Minprocessor(Pro(n + 1)) = Min(n + 1)
Then Minprocessor returns the minimal number of processors required when there are n + 1 processes

Case 2: the n+1 process can not be added to a processor in set M
Then Min(n + 1) = Min(n) + 1
From algorithm, Minprocessor(Pro(n + 1)) = Minprocessor(Pro(n)) + 1
Since Minprocessor(Pro(n)) = Min(n) by induction hypothesis
Then Minprocessor(Pro(n + 1)) = Min(n + 1)
Then Minprocessor returns the minimal number of processors required when there are n + 1 processes

Therefore, Minprocessor always return the minimal number of processors required

(c) This algorithm puts all processors required in a set. At the beginning it starts with an empty set and sorts all input processes by their start time in ascending order. Whenever there is a new process need to be assigned to a processor, the algorithm will first check the existing processors in the set. If the new process can fit into a processor in the set, then assign the process to this processor. If the process can not fit in any processor in the set, then add a new processor to the set and assign the process to this new processor. For the data structure, since it is a set of processors, then we can use list to store the processors. For the runtime, it should be $O(n^2)$. There are two loops in the algorithm, the outer loop will iterate n times, for the inner loop, in the worst case, the process can not fit into any processors every time, then the inner loop will go through all the processors in the set, therefore the total runtime for the loops is $1 + 2 + 3 + 4 + ... + n = \frac{n(n+1)}{2} = O(n^2)$. At beginning, we have sorted all processes, and the runtime of sort is $O(nlogn)$, since $O(nlogn) < O(n^2)$, the total runtime of the algorithm is $O(n^2)$.

Question 3

(a) ScheduleTwice(A = $[(s_1, f_1), (s_2, f_2), ...(s_n, f_n)]$)

```
    sortByEndTime(A)
    arrayToLinkedList(A)
    A_1 = GreedySchedule(A)
    A_2 = GreedySchedule(A)
    return A_1, A_1
```

GreedySchedule(A)

```
    S = new set with first item in A
    lastAdded = first item in A
    for item in A
        if item doesn't conflict with lastAdded
            add item to S
            lastAdded = item
            delete item from A
    return S
```

(b) Base cases:

n = 0
- schedule 0 jobs
- max number of jobs scheduled, schedule is optimal

n = 1
- schedule 1 job on A_1
- all jobs scheduled, schedule is optimal

n = 2
- case 1: $(s_1, f_1), (s_2, f_2)$ don't conflict
  - schedule (s_1, f_1) and (s_2, f_2) on A_1
  - all jobs scheduled, schedule is optimal
- case 2: $(s_1, f_1), (s_2, f_2)$ conflict
  - schedule (s_1, f_1) on A_1 and (s_2, f_2) on A_2
  - all jobs scheduled, schedule is optimal

Induction hypothesis: assume schedule is optimal for n jobs (sorted by finish time)

Induction step: prove schedule is optimal for another job $n + 1$ (where $f_n \leq f_{n+1}$)
- case 1: (s_{n+1}, f_{n+1}) conflicts with last jobs in A_1 and A_2
  - (s_{n+1}, f_{n+1}) can't be scheduled
  - by IH, schedule was optimal with n jobs
  - since (s_{n+1}, f_{n+1}) can't be scheduled, schedule remains optimal
- case 2: (s_{n+1}, f_{n+1}) doesn't conflict with last jobs in A_1 and/or A_2
  - (s_{n+1}, f_{n+1}) can be scheduled (into a set it didn't conflict with)
  - by IH, the schedule was optimal with n jobs
  - we increase the number of jobs scheduled by 1 from scheduling (s_{n+1}, f_{n+1})
  - schedule remains optimal

(c) The data structures used are arrays, sets and linked lists. A starts as a list to be sorted and then turns into a linked list. A_1 and A_2 are sets.

GreedySchedule:

Creating a set and initializing a variable takes O(1) time. The for loop takes O(n) time since it loops over every item in A. All the lines in the loop take O(1) time. The if statement, adding an item to a set and setting the value of a variable takes O(1) time. Deleting an item from a linked list takes O(1) time (just have to keep track of item before the item to delete). Overall, GreedySchedule has a time complexity of O(n).

ScheduleTwice:

Sorting the array A = $[(s_1, f_1), (s_2, f_2), ...(s_n, f_n)]$ with mergesort takes O($nlogn$) time. Changing the array into a linked list takes O(n) time (go through each item in A and add it to the end of a linked list). The time complexity of running GreedySchedule twice is O(n). The time complexity of this algorithm is O($nlogn + 3n$) which can be simplified into O($nlogn$). The overall time complexity of O($nlogn$) is less than O(n^2).

Question 4

(a) For greedy algorithm, we pick the coin with the greatest value every time until the sum of the coins is A, or return an empty set. The reason greedy algorithm doesn't work is that because it always picks the greatest coin. When we are closed to value A, there may be not mid-value coin we can pick to make the sum to A since their value are all too large at this moment, then we have to pick many small-value coins in order to make the sum to A. However, since we have picked many small-value coins, the total number of coins may be very big compared to the case if we only pick coins from mid-value coins. For example, if we have a set of coins: { 1,1,1,1,1,15,15,25 } and we want to pick coins to make they sum to 30. By greedy algorithm, we will pick { 1,1,1,1,1,25 }, which is worse than { 15,15 }. Thus greedy algorithm doesn't work for this problem.

(b) First we sort the coins in the input list by their value in ascending order.
Step 1: Define a optimal substructure
Let OPT(m, A) be the number of coins of the optimal solution
Case 1: coin m is not in optimal solution
OPT(m, A) = OPT(m - 1, A)
Case 2: coin m is in optimal solution
OPT(m, A) = 1 + OPT(m - 1, A - c_m)

Step 2: Define a array for intermediate values
Define C[m, A] = number of minimal coins when there are m coins and we want to pick coins to make they sum to A, or ∞ if it is impossible to sum to A

Step 3: Rewrite recursive relation
C[m, A] = min(OPT(m - 1, A), 1 + OPT(m - 1, A - c_m))
Pseudocode:

```
MincoinRe(m, A):
    if m == 0:
        return ∞
    if A == 0:
        return ∞
    if C[m, A] is defined:
        return C[m, A]
    if c_m > A:
        C[m, A] = MincoinRe(m − 1, A)
        return C[m, A]
    if c_m == A:
        C[m, A] = 1
        return C[m, A]
    else:
        C[m, A] = min(MincoinRe(m − 1, A), 1 + MincoinRe(m − 1, A − c_m))
        return C[m, A]
```

Step 4: Bottom-up approach
Pseudocode:

```
Mincoin(m, A):
    Define a 2–D array C[m, A]
    for i = 0 to A:
        C[0, i] = ∞
    for l = 0 to m:
        C[l, 0] = ∞
    for j = 1 to m:
        for k = 1 to A:
            if k < c_j:
                C[j, k] = C[j − 1, k]
            if k = c_j:
                C[j, k] = 1
            else:
                C[j, k] = min(C[j − 1, k], 1 + C[j − 1, k − c_j])
    if C[m, A] == ∞:
        return 0, C
    else:
        return C[m, A], C
```

Step 5: Actual solution

Let n, C = Mincoin(m, A)

If n = 0, then output of final solution is $\emptyset$

If n $\neq$ 0, define a empty list I

Check if C[m, A] = C[m - 1, A - c_m], then add coin m to output list I and check the value of C[m - 1, A - c_m] (if C[m - 1, A - c_m] = C[m - 2, A - c_m] or C[m - 1, A - c_m] = C[m - 2, A - c_m - c_{m-1}])

Else if C[m, A] = C[m - 1, A], then do nothing with output list I and check the value of C[m - 1, A] (if C[m - 1, A] = C[m - 2, A] or C[m - 1, A] = C[m - 2, A - c_{m-1}])

Keep going these check steps until the we reach C[0, x]

Then return I and this is the final solution contains minimal number of coins to sum to A

Argument: This algorithm divides the original problem into subproblems recursively, at each recursive step, the algorithm considers two cases: (1)the current coin is included in the optimal solution (2)the current coin is not included in the optimal solution. The algorithm always takes the optimal case at each step. And for the most basic subproblem, it is that the value of the coin is same as the sum we want it to be, so in that case we just take that coin and this is obviously the optimal solution for this most basic subproblem. Since the algorithm keeps take optimal solution recursively form subproblem, the final solution it returns must be optimal.

(c) There are three loops in the algorithm, one is nested loop, the first two loops is to initialize the array for 0 condition and they iterate A times and m times respectively, the nested is to assign the value to the remaining part of the array and it iterate m·A times. In the worst case, the algorithm go through all the loops and the runtime is $O(m \cdot A)$

Question 5

Step I: Optimal Substructure Definition

Let m be the maximum depth

Let n be the maximum horizontal location of the dig site

Let OPT(i, j, d) denote the value of an optimal solution, where i is the depth being dug, j is the column being dug, and d is the durability remaining prior to digging the 10cm x 10cm x 10cm block located at (i,j). Let the case where m = 0 define the case where digging has not yet begun.

If $i = 0$

$$OPT(i, j, d) = \max_{1 \leq k \leq n} (OPT(i + 1, k, d))$$

If $1 \leq i \leq m$

$$OPT(i, j, d) = \begin{cases} 0, \text{ if } d < H[i, j] \\ \max\limits_{\max(1, j-1) \leq k \leq \min(j+1, n)} (G[i, k] + OPT(i + 1, k, d - H[i, j])), \text{ otherwise} \end{cases}$$

If $i = m$

$$OPT(i, j, d) = \begin{cases} 0, \text{ if } d < H[i, j] \\ G[i, k], \text{ otherwise} \end{cases}$$

In Step I, we try to define smaller, similar subproblems which we can operate over to find the optimal amount of gold at each subproblem. We know that in order to do this, we have to calculate the optimal gold amount for every single potential path, since if an optimal subpath exceeds the durability constraint of our drill when longer paths are calculated, then that optimal subpath will not be used at all in the most optimal overall path.

We begin by having three cases:

- one for the "initial" state where nothing is dug, since we can choose from any of the 1 to n columns to dig

- one for the "intermediate" states where we have to choose from one of the three adjacent lower columns to dig

- one for the "last" state where we reach the maximum depth, since this is already the smallest subproblem

We then further break down the "intermediate" and "last" states into two more cases in order to use the first case to report that a block's optimal path has "0 value" if the durability constraint will be violated once we try digging the block. We do not do this to the "initial" state, since the durability constraint cannot be violated if nothing is dug.

If the durability constraint will not be violated in the "intermediate" states, then we move on to the second case and find the most optimal subpath after the current block is dug. We then use the cumulative amount of the most optimal subpath after the current block and the current block's value to calculate the current block's most optimal gold amount. We make sure to decrement the durability value when we look for an optimal subpath to find out what the most optimal subpath is given our durability constraint at that block.

We use the "last" state to report the value of the gold contained within the block, but we do not check for optimal subpaths since there are none at the final depth.

Step II: Define Array for Cache

M[i, j, d] = Optimal value of obtainable gold if the drill were to start with durability d at the block located at (i,j)

Since in Step I our optimal substructure is defined to calculate the most amount of gold to gain from each subpath, it makes sense to have our array store the most amount amount of gold to gain at the starting blocks of each optimal subpath.

Step III: Rewrite the Recurrence Relation

If $i = 0$

$$M[i, j, d] = \max_{1 \le k \le n} (M[i + 1, k, d - H[i, k]])$$

If $1 \le i \le m$

$$M[i, j, d] = \begin{cases} 0, \text{ if } d < H[i, j] \\ \max\limits_{\max(1, j-1) \le k \le \min(j+1, n)} (G[i, k] + M[i + 1, k, d - H[i, j]]), \text{ otherwise} \end{cases}$$

If $i = m$

$$M[i, j, d] = \begin{cases} 0, \text{ if } d < H[i, j] \\ G[i, k], \text{ otherwise} \end{cases}$$

This is essentially the same thing as Step I, but instead of just calculating the value of the optimal substructure, we make sure to store the value within the array defined in Step 2 for the sake of dynamic programming.

Step IV: Bottom-Up Approach

```
Let G be the array of gold content
Let H be the array of hardness
Let h be the initial durability

Bottom-Up-Gold-Digging(G, H, h):
    Let m = maximum depth of G and H
    Let n = maximum horizontal location of G and H

    Define M[1..m, 1..n, 0..d] = 0

    for i = m to 1:
        for j = 1 to n:
            for d = 0 to h:
                if i = m:
                    if d < H[i, k]:
                        M[i, j, d] = 0
                    else:
                        M[i, j, d] = G[i, j]
                else:
                    q = 0
                    for k = max(1, j - 1) to min(j + 1, n):
                        if d < H[i, k]:
                            p = 0
                        else:
                            p = G[i, j] + M[i + 1, k, d - H[i, k]]
                        if q < p:
                            q = p
                    M[i, j, d] = q

    return M
```

In Step IV, we have to calculate the optimal substructures' values using a bottom-up approach. This means we'll have to start from the end of the recursive definition of Step I (i.e. the maximum depth) and move to the beginning (i.e. depth 1), calculating the values of every block as we go. This works out well, since as we move up towards the surface, we can calculate the value of the optimal path by using the already-calculated lower depths. The only problem is that if we run out of durability, we'll simply stop short of the surface, making the entire path pointless. To combat that problem, we'll simply calculate every optimal path for every block for every possible durability. And so, our pseudocode does just that, calculating the "max" gold of each block using pseudocode similar to what was shown in class for the rod cutting problem. Most importantly, the hardness is always checked to make sure it's between 0 to our initial durability, so we never try accessing a nonexistant index!

Step IV $\frac{1}{2}$: Proof of Correctness of Our Algorithm Proof by Complete/Strong Induction

Predicate, P(x): For any given array of gold values G and any given array of hardness values H, if the maximum depth, $m > x$ and $m > 0$, then the above algorithm will calculate an array of the most amount of gold we can possibly earn for every block (i.e. the value of the most optimal substructure) and every possible drill durability of depth $m - x$.

Base Case: $x = 0$

First, define the maximum depth of G and H as m.

For the purposes of our proof, assume $m > x$.

This holds because $m > x = 0$, and $m - x = m$. The initial for loop starts at i=m, so we assign values to all blocks of depth m since we loop through all m-depth elements of M using the two for loops "for j = 1 to n" and "for d = 0 to h", noting that with every loop, assignments are done to M[i,j,d].

If there is not enough durability left, then the value is 0, as it should be since we can get no worth out of the block. If there is enough durability, then the optimal value is equal to the gold contained in the block, as there is no other gold to dig. As these correct values are assigned to their respective blocks, the base case holds for all G and H such that the maximum depth, $m > x$ (as we have made this assumption).

Inductive Hypothesis: Assume P(x) holds for all x such that $0 \leq$ x $<$ m

Inductive Step:

Goal: Prove P(x+1) holds given our Inductive Hypothesis

First, assume $m > x + 1$.

Note that $m > x + 1$, so $m > m - x - 1 > 0$. This means that if we were to loop over every value of i = m to 1, we would eventually each m-x-1.

Similar to the base case, every block of depth m-x+1 of every hardness is assigned a value since we eventually end up at the loop where i=m-x-1. Then, we loop through all m-x-1-depth elements of M using the two for loops "for j = 1 to n" and "for d = 0 to h", noting that with every loop, assignments are done to M[i,j,d], where i = m-x-1.

By our Inductive Hypothesis, we have already calculated an array of the most amount of gold we can possibly earn for every block (i.e. the value of the most optimal substructure) and every possible drill durability of depth $m - x$.

So, when the algorithm compares the values of $M[i + 1, k, d - H[i, k]]$, we just need to pull the values from the array M since $i + 1 = m - x - 1 + 1 = m - x$, and by our Inductive Hypothesis, values of depth m-x are already calculated. We then compare for the max of the optimal values of the blocks of depth i+1 connected to the current block to get the most optimal path for the current block. If the durability is enough to dig the subpath, then we assign this optimal value to the block, and if not, then we assign it 0, to designate this subpath is not diggable. Thus, all blocks of depth m-(x+1) are assigned an optimal value, where $m > x + 1$. So P(x+1) holds.

Hence, P(x) holds for all x such that $0 \leq$ x $<$ m

Step V: Actual Solution

```
Let G be the array of gold content
Let H be the array of hardness
Let h be the initial durability

Gold-Digging-Path(G, H, h):
    Let m = maximum depth of G and H
    Let n = maximum horizontal location of G and H

    M = Bottom-Up-Gold-Digging(G, H, h)

    q = 0  // optimal gold value
    j = 0  // optimal solution's horizontal column number
    c = 0  // optimal solution's hardness value

    for k = 1 to n:
        p = M[1, a, h]
        if q < p:
            q = p
            j = k
            c = H[1, a]

    if q = 0:
        Return ""
    else:
        L = "" + j
        d = h - c  // durability remaining
        for i = 2 to m:
            q = 0
            c = 0
            for k = max(1, j - 1) to min(j + 1, n)
                p = M[i, k, d]
                if q < p:
                    q = p
                    j = k
                    c = H[j, k]
            if q = 0:
                Return L
            else:
                L = L + "," + j
                d = d - c
        Return L
```

For Step V, we find the actual solution by observing that we already have pseudocode that finds the most optimal path for any given starting block in Step IV. This means we simply have to start at the most profitable surface block and move down the depths, choosing the most profitable of the 1-3 connected lower blocks to trace back the most optimal path in reverse of what was done in Step IV. In the pseudocode for Step V, we do this by first finding the starting block from 1 to n and then proceed to find the most optimal next-block to dig for our given durability. We make sure to remember the block we came from, j, at each step, of course, and the most optimal subpath so far, also j, while we try to find the most optimal subpath to go down.

We have two "if q = 0" checks in our pseudocode, since q represents the "maximum gold" of the most optimal subpath we've found. If q = 0 after checking all the subpaths, then clearly either none of the subpaths are diggable or they're not worth digging in the first place if we can't gain any gold from going down them. Hence, in those cases, we simply return the path we have, since no other subpath is left to explore (or worth exploring) in an optimal solution.