COMP3121 - Assignment 2

Mark Pollock - z3372890 - mpol525

Due: 15th April 2015

1 Question 1

Note that in the last problem we used bucket sort in for n buckets in $O(n)$ time. For this problem we do almost the same thing except that we define integer $a = \lceil \frac{1}{e} \rceil$ and we sort the n numbers into $a \times n$ buckets. Thus, the algorithm is:

1. Sort the n numbers into $a \times n$ buckets - $O(n)$.
2. For each bucket, find the minimum and maximum number - $O(n)$.
3. Output the permutation by going through each bucket in order whilst making sure the minimum number for each bucket is the first number in that bucket and the maximum number is the last.

Explanation:
This will ensure that the required condition ($\Sigma_{i=2}^{n}|y_i - y_{i-1}| < 1+e$) is met due to the following logic:

The maximum difference between any 2 elements in the same bucket is the size of that bucket which is $\frac{1}{an}$.

Thus the maximum sum of differences that can be accrued by numbers in the one bucket (in any order) is: $\frac{1}{an} \times k \leq \frac{k}{an}$ where k represents the total number of numbers in the bucket.

The sum of all the differences between numbers that are in the same bucket is thus: $\Sigma \frac{k_i}{an} \leq \frac{\Sigma k_i}{an} = \frac{n}{an} = \frac{1}{a} < e$

Then we must also add the maximum sum accrued from bucket to bucket throughout the entire array. Since the bins are in order, and the minimum of each bucket must be the first for that bucket and the maximum of each bucket must be the last of that bucket it is clear that this is less than or equal to 1.

Thus, the total sum of difference between each number in order is: $\Sigma_{i=2}^{n}|y_i - y_{i-1}| < 1 + e$ as required.

Pseudocode:

```
$x_i \to (x_i, \lfloor x_i an \rfloor)$        #Sort into buckets 0 .. an - 1
declare permutation as list
for each bucket 0 .. an - 1:
```

```
append minimum of bucket to permutation
append remaining values except for maximum to permutation
append maximum of bucket to permutation
```

2 Question 2

By writing out the bits of the i^th element in the original sequence and the i^th element in the resulting permutation I found that element i in the permutation can be found by reversing the bits of element i in the original sequence.
Consider the example given:

$$(0, 1, 2, 3, 4, 5, 6, 7) -> (0, 4, 2, 6, 1, 5, 3, 7)$$

Now lets write the original sequence in binary:

$$(000, 001, 010, 011, 100, 101, 110, 111)$$

If we then reverse the bits in each element we obtain:

$$(000, 100, 010, 110, 001, 101, 011, 111) = (0, 4, 2, 6, 1, 1, 3, 7)$$

which is the resulting permutation.

That is the resulting permutation is found by writing the original sequence in binary and reversing the bits of each element.
This sequence can be described by the recursive formula:

$$\begin{aligned} & P_n = \{2 \cdot P_{n-1}, 1 \dagger 2 \cdot P_{n-1}\} \\ Where, \quad & P_1 = \{0\} \end{aligned}$$

Where:

- $a \cdot P$ is the sequence created by multiplying every element in P by a e.g. $4 \cdot \{3, 6, 1, 9, 0\} = \{6, 12, 2, 18, 0\}$.
- $a \dagger P$ is the sequence created by adding a to every element in P e.g. $5 \cdot \{2, 3, 1, 1, 7\} = \{3, 4, 2, 2, 8\}$.
- $\{P_1, P_2\}$ is the sequence created by appending P_2 to the end of P_1 e.g. $\{\{3, 2, 6\}, \{7, 8, 7\}\} = \{3, 2, 6, 7, 8, 7\}$.

To show how this works lets produce the sequence for $n = 5$ i.e. P_5:

$$\begin{aligned}
&& P_5 &= \{2 \cdot P_4,\ 1 \dagger 2 \cdot P_4\} \\
&Now, & P_4 &= \{2 \cdot P_3,\ 1 \dagger 2 \cdot P_3\} \\
&Now, & P_3 &= \{2 \cdot P_2,\ 1 \dagger 2 \cdot P_2\} \\
&Now, & P_2 &= \{2 \cdot P_1,\ 1 \dagger 2 \cdot P_1\} \\
&& &= \{2 \cdot \{0\},\ 1 \dagger 2 \cdot \{0\}\} \\
&& &= \{\{0\},\ 1 \dagger \{0\}\} \\
&& &= \{0, 1\} \\
&Therefore, & P_3\ becomes: \\
&& P_3 &= \{2 \cdot \{0,1\},\ 1 \dagger 2 \cdot \{0,1\}\} \\
&& &= \{\{0,2\},\ 1 \dagger \{0,2\}\} \\
&& &= \{0,2,1,3\} \\
&Therefore, & P_4\ becomes: \\
&& P_4 &= \{2 \cdot \{0,2,1,3\},\ 1 \dagger 2 \cdot \{0,2,1,3\}\} \\
&& &= \{\{0,4,2,6\},\ 1 \dagger \{0,4,2,6\}\} \\
&& &= \{\{0,4,2,6,1,5,3,7\}\} \\
&Therefore, & P_5\ becomes: \\
&& P_5 &= \{2 \cdot \{0,4,2,6,1,5,3,7\},\ 1 \dagger 2 \cdot \{0,4,2,6,1,5,3,7\}\} \\
&& &= \{\{0,8,4,12,2,10,6,14\},\ 1 \dagger \{0,8,4,12,2,10,6,14\}\} \\
&& &= \{0,8,4,12,2,10,6,14,1,9,5,13,3,11,7,15\}
\end{aligned}$$

3 Question 3

This question is solved by recursion. For each node we first *balance* the left and right subtrees which will inturn *balance* their left and right subtrees until we reach the leaf nodes at which point the recursion will begin to unravel.
Note: In this context *balance* describes the act of increasing the edges in the tree in such a way that ensures that the signal reaches the leafs in the same time from a given node.

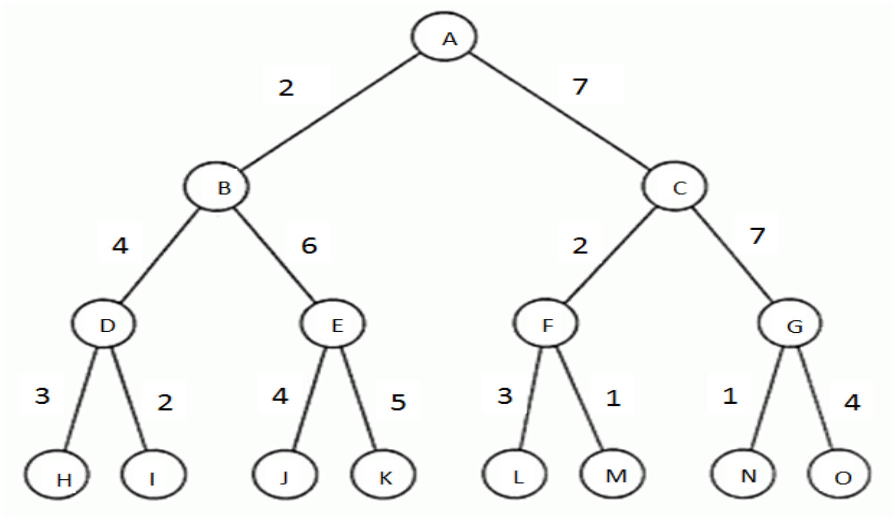
The base case for this recursion involves a tree of depth 1 with a root node and left child and a right child. To *balance* this base case we simply increase the lowest edge length until it equals the higher edge length.

Pseudocode:

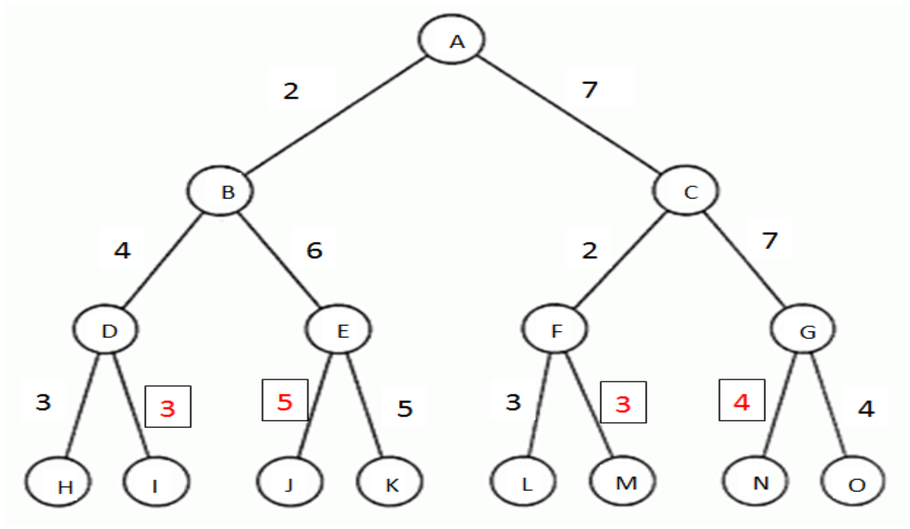
```
typedef struct Node{
    Node *left;
    Node *right;
    int leftLength;
    int rightLength;
}
```

```
void balanceBinTree(binTree t){
    balance(t.root);
}

int balance (Node* n){
    if(n->right->right == NULL){
        if(n->leftLength < n->rightLength)
            n->leftLength == n->rightLength;
        else if(n->rightLength < n->leftLength)
            n->rightLength = n->leftLength;
        return n->rightLength;

    }else
        int left = balance(n->left) + n->leftLength;
        int right = balance(n->right) + n->rightLength;
        while(left < right){
            n->leftLength++;
            left++;
        }while(right < left){
            n->rightLength++;
            right++;
        }
        return left;
    }
}
```

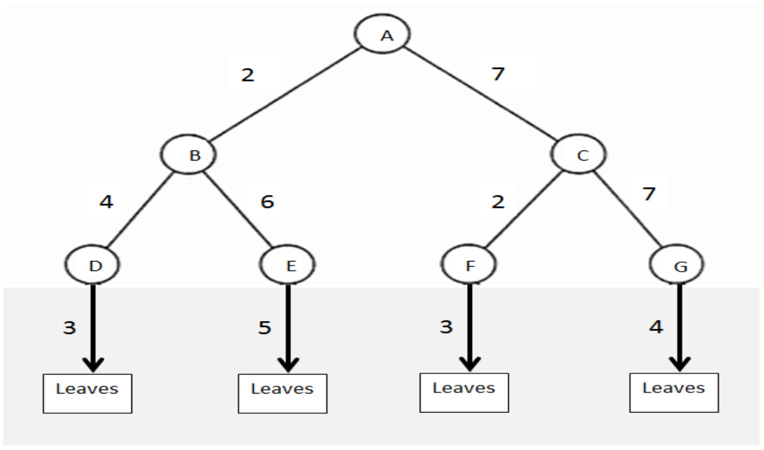
Example walk through: Lets say we have the un-*balanced* binary tree as shown here:



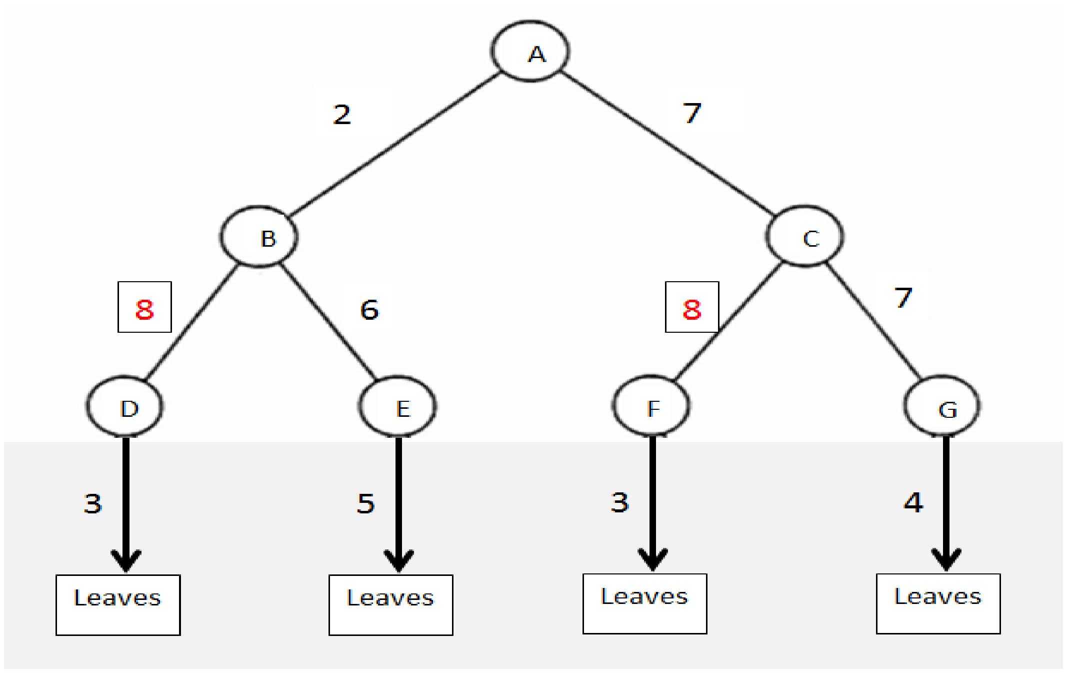
Through the recursive algorithm the subtrees with nodes D, E, F, G will be *balanced* first by increasing the smaller edge length (leftLength or rightLength) to the larger edge length.



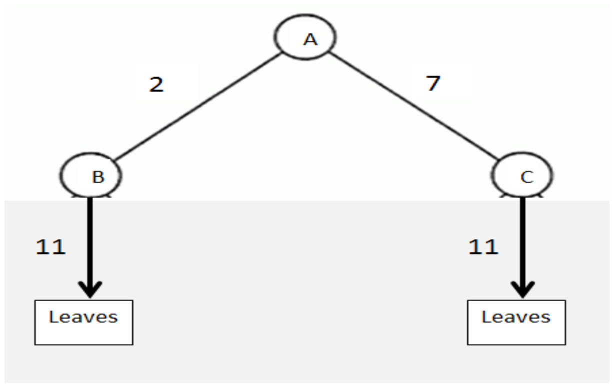
Then the recursion will unravel one step and the tree can be simplified as shown:



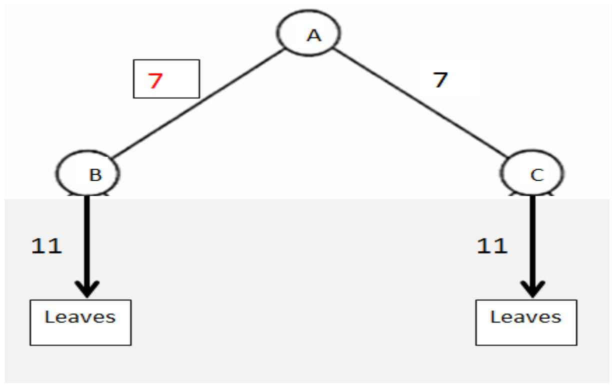
Thus we then *balance* the subtrees with root nodes B and C in the same fashion as above.
For example: Looking at node B. The leftLength is now $4+3=7$ and the rightLength is $6+5=11$ thus we need to add $11-7=4$ to the leftLength which is done by adding 4 to the length B-D. This is repeated for node C.



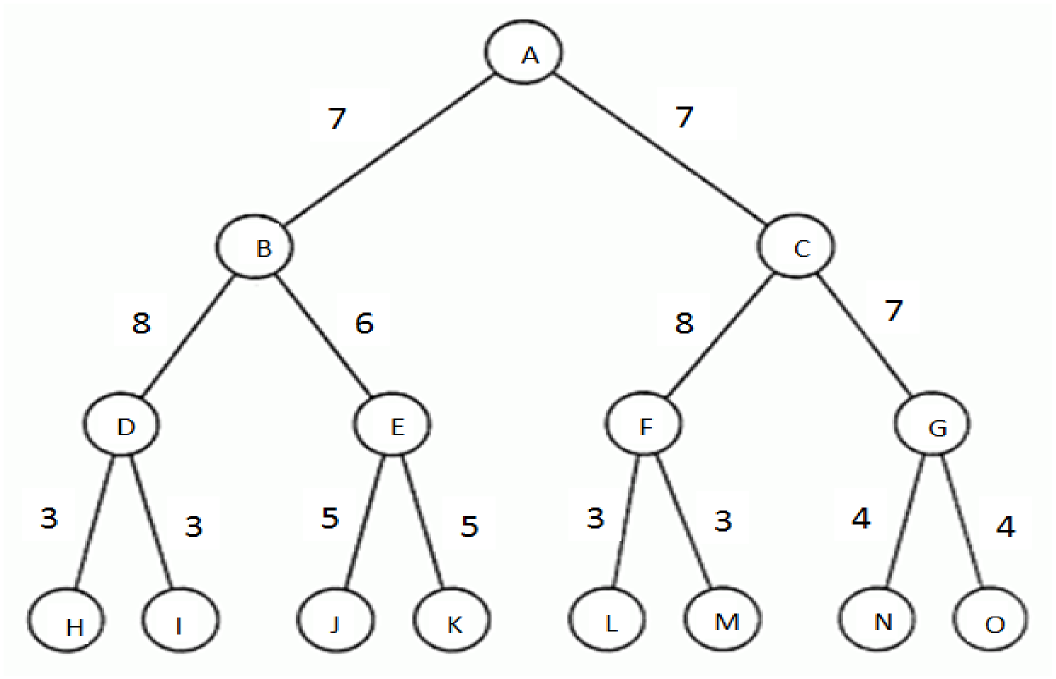
The recursion unravels again and the tree is simplified to the following:



Thus, we can repeat the process again for node A.



Now we have fully *balanced* the tree:



4 Question 4

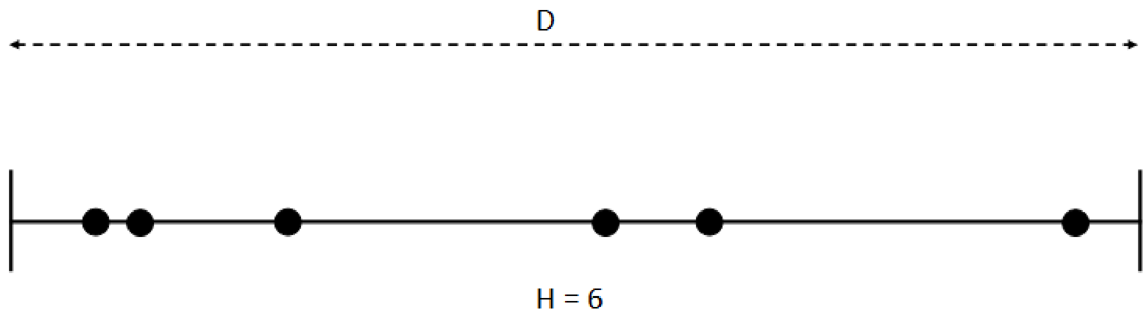
The algorithm required for this question is simple:

1. Start at one end
2. Put first base station so the first house (H_1) is just close enough to receive reception (i.e. 5km further down the road from H_1).
3. Problem is now reduced to a road of length $D - d$ with $H - h$ houses. Where D is the total length of the road, d is the length from the start of the road to the furthest reception provided by the base station, H is the total number of houses on the road, h is the number of houses who are provided with reception by the base station. D, d, H, h are further explained in the figure below.
4. Check if any houses still require service. If so, repeat steps 2, 3, 4.

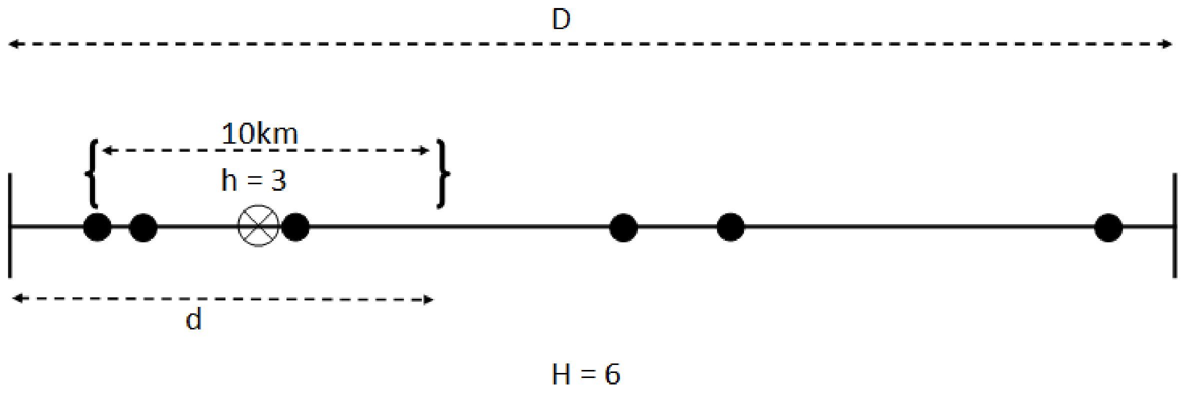
Further explanation:
Consider the following set of figures which demonstrate how the algorithm will work on a given road. Note that the symbols have the following definitions:

$\bullet$ = a house on the road.
$\otimes$ = location of a base station.
$\{ \quad \}$ = range of the base station.
$\vdash - \dashv$ = start, section, end of road.

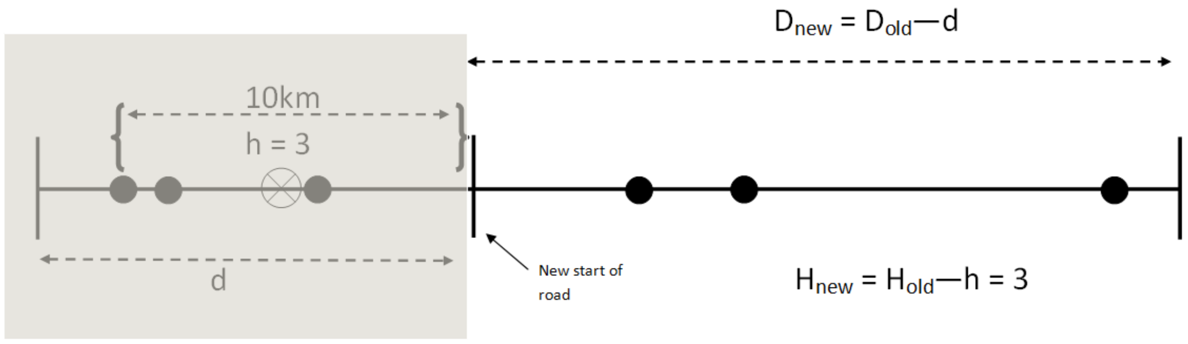
Consider a road of total length D with total Houses H:



Now we apply step 1 & 2 of the algorithm:



Thus, the problem can now be simplified to the following:



Note:
This is a greedy algorithm since the algorithm always chooses the location of the base station to be the furthest along the road whilst still covering the first house that has not yet been covered.

5 Question 5

Basically I have solved this question using a greedy algorithm which involves selecting the most valuable coin possible at each selection.

Algorithm:
Let amount be n and c be the collection of coins to pay with.

```
while($n > 0$){
    if($n >= 2$)
        add a \$2 coin to $c$
        $n$ -= 2
    else if($n >= 1$)
        add a \$1 coin to $c$
        $n$ -= 1
    else if($n >= 0.5$)
        add a 50$c$ coin to $c$
        $n$ -= 0.5
    else if($n >= 0.2$)
        add a 20$c$ coin to $c$
        $n$ -= 0.2
    else if($n >= 0.1$)
        add a 10$c$ coin to $c$
        $n$ -= 0.1
    else if($n >= 0.05$)
        add a 5$c$ coin to $c$
        $n$ -= 0.05
}
```

Argue that it is optimal:
Since each coin is **a multiple of smaller coins and the difference between each coin is non-decreasing**, we can show that $\$(n-c)$ can be paid with **less** coins than $\$n$ where c represents the largest coin that is less than or equal to n: $c \leq n$. Thus, the greedy solution above will produce an optimal solution.

More formal proof:
Assume an optimal and greedy solutions for a price of p are given by

$$\{o_1, o_2, o_3, ..., o_n\}$$
$$\{g_1, g_2, g_3, ..., g_m\}$$

respectively. Where the coins in the optimal solution are sorted from largest value (o_1) to the lowest value (o_n) since order for this problem does not matter. (Note that the greedy solution will already be sorted in this manner).
Now, since the greedy solution always takes the highest value coin possible we can say that:

$$\begin{aligned} & g_1 \geq o_1 \\ Thus, \quad & p - g_1 \leq p - o_1 \end{aligned}$$

Thus, the greedy solution must now have the same or less money required to pay. This can be recursed to show that the greedy solution is optimal.

6 Question 6

Since in the denominations $1,\ c,\ c^2,\ c^3,\ ...,\ c^n$ each denomination is a multiple of smaller denominations (since $c^k = c \times c^{k-1}$) and the difference between the denominations is non-decreasing, this question can be argued with the same logic as question 5. **As such the greedy algorithm is optimal.**

7 Question 7

A set of denominations containing the single cent coin for which the greedy algorithm does not always produce an optimal solution is $1c, 6c, 7c, 8c$.
Take for example an amount of $13c$:
The greedy algorithm would produce $8c, 1c, 1c, 1c, 1c, 1c$ whilst the optimal solution is $7c, 6c$.

8 Question 8

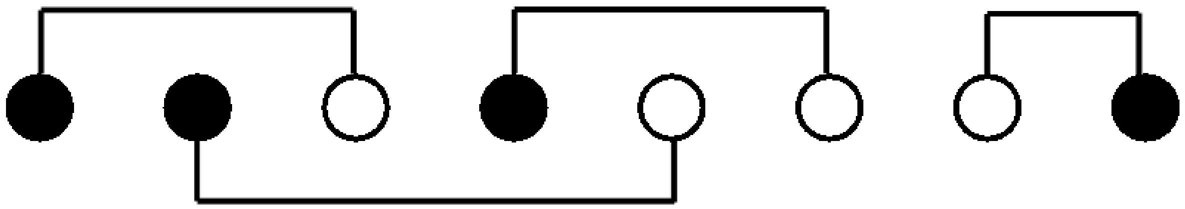
TODO

9 Question 9

Greedy solution:

1. Starting from the left pick the first black dot. Connect this dot to the white dot that is closest to the start.
2. Pick the next left-most black dot and connect it to the left-most white dot that is available (i.e. not already connected to a black dot).
3. Repeat 2 until all black dots are connected to a corresponding white dot.

For example, my greedy algorithm would produce the following given the example case:



Logically, it is optimal due to the following argument: Say we don't connect the left-most black dot with the left-most white dot and we go further along the line by s spaces. Then we have already worsened the greedy solution by length s. We can never get back more than these s spaces (however it is very possible to get back exactly these s-spaces in which case the solution is optimal but not greedy).

10 Question 10

Let A(x) be the polynomial of degree 16 and B(x) be the polynomial of degree 8. i.e.

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_{16}x^{16}$$
$$B(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + ... + b_8x^8$$

Note that if we were to multiply this polynomial out in the current forms it would require $17 \times 9 = 153$ multiplications.
The algorithm to multiply these two polynomials using only 25 multiplications is:

1. Pad A(x) with degree of B 0's. i.e. with 8 0's and pad B(x) with degree of A 0's. i.e. with 16 0's:

$$A(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + ... + a_{16}x^{16} + 0x^{17} + 0x^{18} + ... + 0x^{24}$$
$$B(x) = b_0 + b_1x + b_2x^2 + b_3x^3 + ... + b_8x^8 + 0x^9 + 0x^{10} + ... + 0x^{24}$$

2. Use FFT to obtain the value representation of the padded A(x) and the padded B(x). i.e.:

$$A(x) \xRightarrow{FFT} \{A(1), A(\omega_{25}), A(\omega_{25}^2), A(\omega_{25}^3), ..., A(\omega_{25}^{24})\}$$
$$B(x) \xRightarrow{FFT} \{B(1), B(\omega_{25}), B(\omega_{25}^2), B(\omega_{25}^3), ..., B(\omega_{25}^{24})\}$$

   Note that process for FFT is described in the appendix below.

3. Obtain the value representation of the polynomial C ($C(x) = A(x) \cdot B(x)$) by multiplication of each term in the above value representations. i.e.:

$$\begin{aligned} C(x)\ (\textit{in value representation}) = \{A(1)B(1),&A(\omega_{25})B(\omega_{25}), A(\omega_{25}^2)B(\omega_{25}^2),\\ &A(\omega_{25}^3)B(\omega_{25}^3), ..., A(\omega_{25}^{24})B(\omega_{25}^{24})\}\end{aligned}$$

   Note that this step requires precisely 25 multiplications.

4. Use IFFT to obtain the polynomial form of C(x). i.e.:

$$\begin{aligned}\{A(1)B(1),&A(\omega_{25})B(\omega_{25}), A(\omega_{25}^2)B(\omega_{25}^2),\\ &A(\omega_{25}^3)B(\omega_{25}^3), ..., A(\omega_{25}^{24})B(\omega_{25}^{24})\} \xRightarrow{IFFT} C(x)\end{aligned}$$

   Note that process for IFFT is described in the appendix below.

11 Question 11

$$\begin{aligned}
\omega_n^k &= (\omega_{dn})^{dk} \\
\therefore i(\omega_{64})^7(\omega_{32})^{15} &= i(\omega_{64})^7(\omega_{d\times 32})^{d\times 15} \\
&= i(\omega_{64})^7(\omega_{2\times 32})^{2\times 15} \qquad (d=2) \\
&= i(\omega_{64})^7(\omega_{64})^{30} \\
&= i(\omega_{64})^{7+30} \\
&= i(\omega_{64})^{37} \\
&= (\omega_{64})^{37+\frac{64}{4}} \qquad (\textit{since multiplication by } i \textit{ is an anticlockwise rotation of } 90 \text{ deg}) \\
&= (\omega_{64})^{53} \\
&= \omega_{64}^{53}
\end{aligned}$$

12 Question 12

12.1 12a

Note that if we were to multiply this polynomial out in the current forms it would require $4\times 4 = 16$ multiplications.

The algorithm to multiply these two polynomials using only 7 multiplications is:

1. Let $y = x^2$ and solve for
$P(y) = a_0 + a_2y + a_4y^2 + a_6y^3 \quad andQ(y) = b_0 + b_2y + b_4y^2 + b_6y^3$
2. Pad P(y) with degree of Q 0's. i.e. with 3 0's and pad Q(x) with degree of A 0's. i.e. with 3 0's:

$$P(y) = a_0 + a_2y + a_4y^2 + a_6y^3 + 0y^4 + 0^y5 + 0y^6$$
$$Q(y) = b_0 + b_2y + b_4y^2 + b_6y^3 + 0y^4 + 0^y5 + 0y^6$$

3. Use FFT to obtain the value representation of the padded P(y) and the padded Q(y). i.e.:

$$P(y) \xrightarrow{FFT} \{P(1), P(\omega_7), P(\omega_7^2), P(\omega_7^3), ..., P(\omega_7^6)\}$$
$$Q(y) \xrightarrow{FFT} \{Q(1), Q(\omega_7), Q(\omega_7^2), Q(\omega_7^3), ..., Q(\omega_7^6)\}$$

Note that process for FFT is described in the appendix below.

4. Obtain the value representation of the polynomial C ($C(y) = P(y)\cdot Q(y)$) by multiplication of each term in the above value representations. i.e.:

$$C(y)\ (in\ value\ representation) = \{P(1)Q(1), P(\omega_7)Q(\omega_7), P(\omega_7^2)Q(\omega_7^2), \\ P(\omega_7^3)Q(\omega_7^3), ..., P(\omega_7^6)Q(\omega_7^6)\}$$

Note that this step requires precisely 7 multiplications.

5. Use IFFT to obtain the polynomial form of C(y). i.e.:

$$\{P(1)Q(1), P(\omega_7)Q(\omega_7), P(\omega_7^2)Q(\omega_7^2), \\ P(\omega_7^3)Q(\omega_7^3), ..., P(\omega_7^6)Q(\omega_7^6)\} \xRightarrow{IFFT} C(y)$$

Note that process for IFFT is described in the appendix below.

6. Now find $C(x)$ by replacing everyone value of y in $C(y)$ with x^2

12.2 12b

Note that if we were to multiply this polynomial out in the current forms it would require $5 \times 5 = 25$ multiplications.

The algorithm to multiply these two polynomials using only 16 multiplications is:

1. Write the polynomials like so:

$$P(x) = a_0 + x^{17}(a_{17} + a_{19}x^2 + a_{21}x^4 + a_{23}x^6)$$
$$Q(x) = b_0 + x^{17}(b_{17} + b_{19}x^2 + b_{21}x^4 + b_{23}x^6)$$

Therefore, we can multiply the two as such:

$$\begin{aligned} P(x)Q(x) &= (a_0 + x^{17}(a_{17} + a_{19}x^2 + a_{21}x^4 + a_{23}x^6)) \times (b_0 + x^{17}(b_{17} + b_{19}x^2 + b_{21}x^4 + b_{23}x^6)) \\ &= a_0b_0 + a_0x^{17}(b_{17} + b_{19}x^2 + b_{21}x^4 + b_{23}x^6) + b_0x^{17}(a_{17} + a_{19}x^2 + a_{21}x^4 + a_{23}x^6) \\ &\quad + x^{17}x^{17}(a_{17} + a_{19}x^2 + a_{21}x^4 + a_{23}x^6)(b_{17} + b_{19}x^2 + b_{21}x^4 + b_{23}x^6) \\ &= a_0b_0 + a_0b_{17}x^{17} + a_0b_{19}x^{19} + a_0b_{21}x^{21} + a_0b_{23}x^{23} + b_0a_{17}x^{17} + b_0a19x^{19} + b_0a_{21}x^{21} \\ &\quad + b_0a_{23}x^{23} + x^{34}(a_{17} + a_{19}x^2 + a_{21}x^4 + a_{23}x^6)(b_{17} + b_{19}x^2 + b_{21}x^4 + b_{23}x^6) \end{aligned}$$

Now, if we let

$$(a_{17} + a_{19}x^2 + a_{21}x^4 + a_{23}x^6) = A(x)(b_{17} + b_{19}x^2 + b_{21}x^4 + b_{23}x^6) = B(x)$$

Then the last line of the above equation becomes (here I will explicitly put a $\times$ for each multiplication of coefficients):

$$P(x)Q(x) = a_0 \times b_0 + a_0 \times b_{17}x^{17} + a_0 \times b_{19}x^{19} + a_0 \times b_{21}x^{21} + a_0 \times b_{23}x^{23} + b_0 \times a_{17}x^{17} \\ + b_0 \times a_{19}x^{19} + b_0 \times a_{21}x^{21} + b_0 \times a_{23}x^{23} + x^{34}A(x)B(x)$$

Thus, we have 9 multiplications + the multiplications required for $A(x)B(x)$.

It is very important to note that $A(x)$ and $B(x)$ are the exact same form as the equations $P(x)$ and $Q(x)$ in (12a). In 12a we showed that these two equations can me multiplied together with only 7 multiplications, I will not repeat the work here.

Thus, $\#multiplications = 9 + 7 = 16$ as required.

12.3 12c

Note that if we were to multiply this polynomial out in the current forms it would require $2 \times 2 = 4$ multiplications.

The algorithm to multiply these two polynomials using only 7 multiplications is:

1. Let $y = x^1 00$ and solve for
   $P(y) = a_0 + a_1 00y \quad and Q(y) = b_0 + b_1 00y$
2. Pad P(y) with degree of Q 0's. i.e. with 1 0 and pad Q(x) with degree of A 0's. i.e. with 1 0:

$$P(y) = a_0 + a_{100}y + 0y^2$$
$$Q(y) = b_0 + b_{100}y + 0y^2$$

3. Use FFT to obtain the value representation of the padded P(y) and the padded Q(y). i.e.:

$$P(y) \xRightarrow{FFT} \{P(1), P(\omega_3), P(\omega_3^2)\}$$
$$Q(y) \xRightarrow{FFT} \{Q(1), Q(\omega_3), Q(\omega_3^2)\}$$

   Note that process for FFT is described in the appendix below.

4. Obtain the value representation of the polynomial C ($C(y) = P(y) \cdot Q(y)$) by multiplication of each term in the above value representations. i.e.:

$$C(y)\ (in\ value\ representation) = \{P(1)Q(1), P(\omega_3)Q(\omega_3), P(\omega_3^2)Q(\omega_3^2)\}$$

   Note that this step requires precisely 3 multiplications.

5. Use IFFT to obtain the polynomial form of C(y). i.e.:

$$\{P(1)Q(1), P(\omega_3)Q(\omega_3), P(\omega_3^2)Q(\omega_3^2)\} \xRightarrow{IFFT} C(y)$$

   Note that process for IFFT is described in the appendix below.

6. Now find $C(x)$ by replacing every value of y in $C(y)$ with x^{100}

13 Appendix

13.1 FFT

Fast Fourier Transform is a process that carries out a discrete Fourier transform on a polynomial. That is, it takes the coefficient form of a polynomial and produces the value form of that polynomial. I have used it several times in the above assignment and thus will describe the algorithm here.

Algorithm (taken from lecture notes):

```
$FFT(A)$
    $n \leftarrow length[A]$
    $if\ n = 1$
        $return\ A$
    $A^{[0]} \leftarrow (A_0, A_2, ..., A_{n-2})$
    $A^{[1]} \leftarrow (A_1, A_3, ..., A_{n-1})$
    $y^{[0]} \leftarrow FFT(A^{[0]})$
    $y^{[1]} \leftarrow FFT(A^{[1]})$
    $\omega_n \leftarrow e^{\frac{2\pi}{n}}$
    $\omega \leftarrow 1$
    $for\ k = 0\ to\ k = n/2 - 1\ do:$
        $y_k \leftarrow y_k^{[0]} + \omega \cdot y_k^{[1]}$
        $y_{k+n/2} \leftarrow y_k^{[0]} - \omega \cdot y_k^{[1]}$
        $\omega \leftarrow \omega \cdot \omega_n$
    $return\ y$
```

Another way to look at it is:

$$A(x) = A_0 + A_1x + A_2x^2 + ... + A_{n-1}x^{n-1} \xRightarrow{FFT} \{A(1), A(\omega_n), A(\omega_n^2), ..., A(\omega_n^{n-1})\}$$

This process can be carried out by using the FFT matrix:

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^{2\cdot 2} & \dots & \omega_n^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix}$$

With the following equation:

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^{2\cdot 2} & \dots & \omega_n^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} A(1) \\ A(\omega_n) \\ A(\omega_n^2) \\ \vdots \\ A(\omega_n^{n-1}) \end{pmatrix}$$

13.2 IFFT

Inverse Fast Fourier Transform is the inverse of the above process. That is, it takes the value representation of a polynomial and provides the coefficient form of a polynomial. This can be done using the same algorithm as shown for FFT but changing all $\omega^i to \omega^{-1}$.

In the matrix form, the equation is the same as for FFT except we use the inverse of the matrix:

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_n \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^{2\cdot 2} & \dots & \omega_n^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix}^{-1} \begin{pmatrix} A(1) \\ A(\omega_n) \\ A(\omega_n^2) \\ \vdots \\ A(\omega_n^{n-1}) \end{pmatrix}$$

However,

$$\begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^{2\cdot 2} & \dots & \omega_n^{2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)(n-1)} \end{pmatrix}^{-1} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \dots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-2\cdot 2} & \dots & \omega_n^{-2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \dots & \omega_n^{-(n-1)(n-1)} \end{pmatrix}$$

Thus,

$$\begin{pmatrix} A_0 \\ A_1 \\ A_2 \\ \vdots \\ A_n \end{pmatrix} = \frac{1}{n} \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n^{-1} & \omega_n^{-2} & \dots & \omega_n^{-(n-1)} \\ 1 & \omega_n^{-2} & \omega_n^{-2\cdot 2} & \dots & \omega_n^{-2\cdot(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & \omega_n^{-(n-1)} & \omega_n^{-2(n-1)} & \dots & \omega_n^{-(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} A(1) \\ A(\omega_n) \\ A(\omega_n^2) \\ \vdots \\ A(\omega_n^{n-1}) \end{pmatrix}$$