CS 310 – Advanced Data Structures and Algorithms

Greedy

July 17, 2017
Greedy Algorithm

- Like dynamic programming, used to solve optimization problems.
- Problems exhibit optimal substructure (like DP).
- Locally optimal choice at each stage
  - always makes the choice that looks best at the moment
- Make a locally optimal choice in hope of getting a globally optimal solution. Does not in general produce an optimal solution
  - When it is an optimal solution, it is usually the simplest and most efficient algorithms available.
Task – buy a cup of coffee (say it costs 63 cents).

You are given an unlimited number of coins of all types (neglect 50 cents and 1 dollar).

Pay exact change.

What is the combination of coins you’d use?

1 cent  5 cents  10 cents  25 cents
Logically, we want to minimize the number of coins.

The problem is then: Count change using the fewest number of coins – we have 1, 5, 10, 25 unit coins to work with.

The ”greedy” part lies in the order: We want to use as many large-value coins to minimize the total number.

When counting 63 cents, use as many 25s as fit, \(63 = 2(25) + 13\), then as many 10s as fit in the remainder: \(63 = 2(25) + 1(10) + 3\), no 5’s fit, so we have \(63 = 2(25) + 1(10) + 3(1)\), 6 coins.
Greedy Algorithms

- A greedy person grabs everything they can as soon as possible.
- Similarly a greedy algorithm makes locally optimized decisions that appear to be the best thing to do at each step.
- Example: Change-making greedy algorithm for “change” amount, given many coins of each size:
  - Loop until change == 0:
  - Find largest-valued coin less than change, use it.
  - change = change - coin-value;
Lemma

If $C$ is a set of coins that corresponds to optimal change making for an amount $n$, and if $C'$ is a subset of $C$ with a coin $c \in C$ taken out, then $C'$ is an optimal change making for an amount $n - c$. 

Proof.

By contradiction:

Assume that $C'$ is not an optimal solution for $n - c$.

In other words, there is a solution $C''$ that has fewer coins than $C'$ for $n - c$.

So we could combine $C''$ with $c$ to get a better solution than $C'$, contradicting the assumption that $C$ is optimal.
Lemma

If $C$ is a set of coins that corresponds to optimal change making for an amount $n$, and if $C'$ is a subset of $C$ with a coin $c \in C$ taken out, then $C'$ is an optimal change making for an amount $n - c$.

Proof.
By contradiction:

- Assume that $C'$ is not an optimal solution for $n - c$.
- In other words, there is a solution $C''$ that has fewer coins than $C'$ for $n - c$.
- So we could combine $C''$ with $c$ to get a better solution than $C$, contradicting the assumption that $C$ is optimal.
- (Cut-and-Paste)
This lemma expresses the fact that the greedy algorithm for change making has the *optimal substructure property*.

For a greedy algorithm to be optimal, it also has another property which tells at each step exactly what choice to make.

This means we don’t have to memoize intermediate results for later use.

We know exactly at each step what we need to do.

This is called *The greedy choice property*.

It means that at every step the greedy choice is a safe one.
The Greedy Choice Property

Lemma
Any optimal solution involving US coins cannot have more than two dimes, one nickel and four cents.

Proof.
- If we had three dimes we could replace them by a quarter and a nickel, resulting in one fewer coins.
- Replace two nickels by a dime, resulting in one fewer coins.
- Replace five cents by a nickel, resulting in four fewer coins.

Corollary
The total sum of \( \{1, 5, 10\} \) coins cannot exceed 24 cents.
The Greedy Choice Property

- The above property can be shown for values of $n < 25$ (and only \{1, 5, 10\} coins).
- In this case, the greedy choice is to select, at every step, the largest coin we can use.
- In other words: The optimal solution for $n$ always contains the largest coin $c_i$ such that $c_i \leq n$
The Greedy Choice Property

Proof.

Again, by contradiction

- Assume there is a solution $C$ for $n$ that does not contain $c_i$.
- It means that it contains only smaller coins.
- But $c_i \leq n$ and every bigger coin can be expressed as a combination of smaller coins (see above).
- So we can always substitute $c_i$ for a combination of smaller coins (that includes the next smallest), getting a better solution.
The Greedy Choice Property

- In the case of US coins – yes, but not always. Why?
- Because while the optimal substructure always exists, the greedy choice property does not exist for all coin combinations.

In general, if we have a set of coins \( \{a_1, a_2, \ldots, a_m\} \) such that \( a_t < a_{t-1} \) and for each pair \( a_t, a_{t-1} \) define \( m_t = \lceil \frac{a_{t-1}}{a_t} \rceil \) and \( S_t = a_t \times m_t \), then the greedy solution is optimal only if for every \( t \in 2..m \), \( G(S_t) \leq m_t \) where \( G(S_t) \) is the greedy solution for \( S_t \).

For example – if we add a 7-cent piece, then \( \lceil \frac{10}{7} \rceil = 2 \), and \( S_t = 7 \times 2 = 14 \), and \( G(14) = 5 > 2 \).

Also, for the set \( \{1, 10, 25\} \) we cannot guarantee the greedy choice property for a similar reason: \( \lceil \frac{25}{10} \rceil = 3 \), \( S_t = 10 \times 3 = 30 \) and \( G(30) = 6 > 3 \).

- Can we use DP to solve it?
Another Example – Activity Selection

- **Input:** Set $S$ of $n$ activities $\{a_1, a_2, \ldots, a_n\}$.
- $s_i =$ start time of activity $i$.
- $f_i =$ finish time of activity $i$.
- **Output:** Subset $A$ of maximum number of compatible activities.
- Two activities are compatible, if their intervals do not overlap.

Example (activities in each line are compatible):

```
Time
----------------------------------------
```

```
\[ \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \]
```

```
\[ \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \]
```

```
\[ \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \]
```

```
\[ \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \]
```

```
\[ \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \quad \overrightarrow{\text{Time}} \]
```
Assume activities are sorted by finishing times – \( f_1 \leq f_2 \leq \cdots \leq f_n \).

Suppose an optimal solution includes activity \( a_k \).

This generates two subproblems:

- Selecting from \( a_1, \ldots, a_{k-1} \), activities compatible with one another, and that finish before \( a_k \) starts (compatible with \( a_k \)).
- Selecting from \( a_{k+1}, \ldots, a_n \), activities compatible with one another, and that start after \( a_k \) finishes.

The solutions to the two subproblems must be optimal.

Prove using the cut-and-paste approach.
Possible Recursive Solution

- Let $S_{ij} =$ subset of activities in $S$ that start after $a_i$ finishes and finish before $a_j$ starts.
- Subproblems: Selecting maximum number of mutually compatible activities from $S_{ij}$.
- Let $c[i,j] =$ size of maximum-size subset of mutually compatible activities in $S_{ij}$.
- The recursive solution is:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{i < k < j} \{c[i,k] + c[k,j] + 1\} & \text{otherwise} \end{cases}$$

- This is highly inefficient, but it can lead us to the next step...
The problem also exhibits the greedy-choice property.

There is an optimal solution to the subproblem $S_{ij}$, that includes the activity with the smallest finish time in set $S_{ij}$.

It can be proved easily.

Hence, we can use greedy:

1. Given activities sorted by finishing time:
2. Select the activity $a_i$ with the smallest finishing time, add it to the solution.
3. Remove from consideration all the activities that are incompatible with $a_i$ (every activity $a_m$ such that $s_m < f_i$).
4. Repeat with remaining activities until no activities are left.
Knapsack Example

<table>
<thead>
<tr>
<th></th>
<th>item1</th>
<th>item2</th>
<th>item3</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>value</td>
<td>60</td>
<td>100</td>
<td>120</td>
</tr>
</tbody>
</table>

- **0 - 1 knapsack:**
  - take item2 and item3
  - total weight: $20 + 30 = 50$
  - total value: $100 + 120 = 220$

- **Fractional knapsack:**
  - take item1 and item2 and $\frac{2}{3} \times$ item3
  - total weight: $10 + 20 + 30 \times \frac{2}{3} = 50$
  - total value: $60 + 100 + 120 \times \frac{2}{3} = 240$
Fractional Knapsack

- Calculate the ratio = value/weight for each item
- Sort the items by decreasing ratio
- Take the item with highest ratio and add them
- Until we can’t add the next item as whole and at the end add the next item as much as we can.
  - Observe that the algorithm may take a fraction of an item, which can only be the last selected item.
  - The total cost for this set of items is an optimal cost.
Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there is always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem ⇒ optimal solution to the problem.
- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order.
- Example: Sorting activities by finish time.
File compression using reduced representation of characters

Let $F$ be a file with $n$ characters (size $n$ bytes, or $8n$ bits)

Each byte is a binary representation of the ASCII code of a character

Represent every character using a unique code of $m$ bits ($m < 8$), and write a file $F'$ with the original characters replaced by their codes

The new file size is $mn < 8n$ bits

Lossless compression

We should be able to decompress $F'$ and get $F$ back
Intuitively, we can assign shorter codes to frequent characters and save more space.

With this distribution, we would like short codes for sp, nl, and 0, and longer ones for the other digits.

But how can we ever uncompress if we don’t know the length of the codes?

The answer is to use **prefix codes** (or as they are sometimes refer to, ”prefix-free codes”).

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Prefix Codes

- Prefix just means some initial substring.
- For example 110 is a prefix of 11011.
- A set of prefix codes has the property that no code (a bit string) is the prefix of another code.
- Some sources call them prefix-free code, which is probably a more accurate name.
- With a set of prefix codes, if you match up the initial bits of the compressed data with all the bits of a certain code, it can only be that code.
- Then you move to the next bits, etc.
Prefix Codes

For example \( \{00, 10, \text{and } 110\} \) is a set of prefix codes, because all three pass the test:

- Testing 00: neither 10 nor 110 start with 00
- Testing 10: neither 00 nor 110 start with 10
- Testing 110: neither 00 nor 10 start with 110

\( \{0, 10, 11\} \) is also a set of prefix codes.

The set \( \{0, 01, 11\} \) is not a set of prefix codes because 0 is a prefix of 01.
How can we generate a set of prefix codes for a certain use?

Answer: Compose a binary tree with the right number of leaves.

Each code is determined by a path from the root to the leaf, where going left gives a 0 and going right a 1. For example:
Here is our 12-symbol example again:

Suppose digits 1 through 9 are about equally likely, though of declining frequency with size (this is actually observed), but 0, sp, and nl are much more frequent:

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>(95 total)</td>
</tr>
</tbody>
</table>

We can set up a binary tree with these symbols at the leaves, like this for our set of 12 symbols:
Generating Prefix codes

```
      0 1
     / \ / \  \
  2  3  4  5  6  7  8  9
      \   \   \   \
    space newline
```
Generating Prefix Codes

- From the binary tree, we read off the codes.
- \(\text{nl}\) is reached by traversing down the right hand side of the tree, going right 2 times, so its code is 11.
- The \(\text{sp}\) is reached by going right and then left, so its code is 10.
- 1 is reached by going left, then right, then right, so its code is 011, and so on.
- Total bits = 291, much better than \(4 \times 95 = 380\)

<table>
<thead>
<tr>
<th>char</th>
<th>code</th>
<th>freq.</th>
<th>total bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp</td>
<td>10</td>
<td>30</td>
<td>60</td>
</tr>
<tr>
<td>nl</td>
<td>11</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>0</td>
<td>010</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>011</td>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>2</td>
<td>00000</td>
<td>6</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>00001</td>
<td>5</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>00010</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>00011</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>00100</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>00101</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>00110</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>00110</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Total</td>
<td>-</td>
<td>95</td>
<td>291</td>
</tr>
</tbody>
</table>
Huffman’s Algorithm

The algorithm works as follows:

1. Sort the character by frequencies.
2. At each stage take the two least frequent characters and merge them into one “super-character” whose frequency is the sum of the frequencies of the original two characters.
3. Replace the original two characters with the merged super-character. Keep all the characters sorted at all times.
4. Repeat until all the characters are merged into one big super-super-character.
5. Build the coding tree such that for every merging operation involving a character, its code size increases by one.

Notice that at every stage we extract two characters and replace them by one, so after \( n-1 \) stages we’re done.
Huffman’s Coding

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>0</th>
<th>4</th>
<th>5</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>2</th>
<th>3</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Huffman’s Coding

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>0</th>
<th>3</th>
<th>8</th>
<th>9</th>
<th>4</th>
<th>5</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

```
  9
 /|
/  \
/    \
  4   7
 /|
/  \
/    \
  6
```

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>6</th>
<th>7</th>
<th>2</th>
<th>0</th>
<th>3</th>
<th>8</th>
<th>9</th>
<th>4</th>
<th>5</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>7</td>
</tr>
</tbody>
</table>

```
  12
 /|
/  \
/    \
  6

  9
 /|
/  \
/    \
  4
```

<table>
<thead>
<tr>
<th>Char</th>
<th>sp</th>
<th>nl</th>
<th>4</th>
<th>5</th>
<th>1</th>
<th>6</th>
<th>7</th>
<th>2</th>
<th>0</th>
<th>3</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>30</td>
<td>20</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

```
  14
 /|
/  \
/    \
  7

  12
 /|
/  \
/    \
  6
```

```
  9
 /|
/  \
/    \
  4
```
Huffman’s Coding

<table>
<thead>
<tr>
<th>Char</th>
<th>nl</th>
<th>0</th>
<th>8</th>
<th>9</th>
<th>3</th>
<th>sp</th>
<th>1</th>
<th>4</th>
<th>5</th>
<th>2</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Freq.</td>
<td>20</td>
<td>10</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>30</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

The solution is not unique! Convention to Assign code: larger weight = 0, smaller weight = 1, random code for same weights.
## Huffman’s Coding

<table>
<thead>
<tr>
<th>char</th>
<th>code</th>
<th>freq.</th>
<th>total bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>sp</td>
<td>00</td>
<td>30</td>
<td>60</td>
</tr>
<tr>
<td>nl</td>
<td>10</td>
<td>20</td>
<td>40</td>
</tr>
<tr>
<td>0</td>
<td>110</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>1</td>
<td>0101</td>
<td>7</td>
<td>28</td>
</tr>
<tr>
<td>2</td>
<td>0110</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>1110</td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>01000</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>01001</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>01110</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>01111</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>11110</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>11111</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

Total: 95 bits, 287 total bits

![Huffman Tree]

Tong Wang
UMass Boston CS 310
July 17, 2017 31 / 35
Two observations we have to make regarding an optimal coding tree (not necessarily Huffman tree):

1. There are no parent nodes with a single child. Every node is either a leaf or has two children (why?)

2. The two least frequent characters will always be on a longest path (why?)*.

* This last property is what makes the greedy choice safe in this case
Why is Huffman’s Algorithm Optimal?

The important thing to realize about Huffman’s algorithm:

1. The characters get processed by frequency, so given two characters $x$ and $y$ with frequencies $f_x$ and $f_y$, and code lengths $l_x$ and $l_y$ respectively, then the algorithm makes it so that if $f_x \leq f_y$, then $l_x \geq l_y$. In other words – less frequent characters get longer codes.

2. The property above guarantees that the tree is optimal and no strictly better tree can be constructed for this frequency (equally good, yes. But not better).
Why is Huffman’s Algorithm Optimal?

- There cannot be a tree with a better overall weight.
- Any tree that has a less frequent character on a shorter branch than a more frequent character cannot be better than the Huffman tree $T$.
- If we have two characters $x$ and $y$ with frequencies $f_x$ and $f_y$, and code lengths $l_x$ and $l_y$ respectively, then $x$'s contribution to the overall weight of the tree is $f_x \times l_x$ and $y$'s contribution is $f_y \times l_y$.
- Let's assume w.l.o.g that $f_x < f_y$, which in a Huffman tree means that $l_x \geq l_y$. 
Why is Huffman’s Algorithm Optimal?

- Then if we exchange $x$ and $y$ and $T$’s overall weight was originally $w$, then it is now

$$w - (f_x \cdot l_x + f_y \cdot l_y) + (f_x \cdot l_y + f_y \cdot l_x) = w + f_x (l_y - l_x) + f_y (l_x - l_y) \geq w$$

- Notice that we add to $w$ a non-positive term multiplied by $f_x$ (the smaller frequency) and a non-negative term multiplied by $f_y$ (the bigger frequency).