Introduction
Today

• Algorithm Design Techniques
  ▪ Divide and Conquer
  ▪ Greedy
  ▪ Dynamic Programming

• Break:
  ▪ 18:30(ish) – 15 Minute Break
Why Algorithm Design?

GCSE syllabus requires problem solving and the understanding of algorithms.

Also:
• Helps with structuring programs!
• Practice using pseudocode.
Divide and Conquer
Divide & Conquer: Overall Idea

- *Divide* a problem into simpler subproblems.
- *Conquer* (Solve) the subproblems recursively.
- *Combine* the solutions to subproblems into a solution for the overall problem. (Optional step)
Recap Merge Sort

To complete this type of sort we are all going to follow the same instructions:

- If you have 1 card – pass it back to the person who passed it to you.
- If you have more than 1 card, split the cards in half and pass each pack to two different people.
- When you get the two piles back they will be sorted so merge (in order) the packs and pass them back to the person who passed them to you.
Merge Sort
Merge Sort Steps

1. Divide the problem in half.
2. Sort each sub problem by recursively calling merge sort.
3. Combine (merge) two smaller sorted sequences.

Merge-Sort (A)

1. if Length(A) <= 1 then: return A
2. else:
   3. mid = length(A)/2
      % % Divide
   4. left = A[1...mid], right = A[mid+1...length(A)]
      % % Conquer
   5. a = Merge-Sort (left)
      % % Conquer
   6. b = Merge-Sort (right)
      % % Conquer
   7. return Merge (a,b) % % Combine
Divide and Conquer: Searching

Task:

Given a sorted list, can you think of a Divide and Conquer algorithm to search for a particular element?

Hint: Think back to last week...
When looking for a value \( V \):

1. Start with a sorted list.
2. Pick the middle value then
   a. if it is the value you are looking for return it.
   b. if it is less than \( V \), recurse one values before \( V \).
   c. if it is greater than \( V \), recurse on values after \( V \).

Which are the divide and conquer steps?

Picture source: http://railspikes.com/assets/2008/10/3/binary_search.png
def bSearch(array, searchValue, left, right):
    if right < left:
        return -1

    mid = (left + right) / 2  # Divide
    if searchValue > array[mid]:
        return bSearch(array, searchValue, mid + 1, right)  # Conquer
    elif searchValue < arr[mid]:
        return bSearch(array, searchValue, left, mid - 1)  # Conquer
    else:
        return mid
Being Greedy
Greedy Algorithms: Overall Idea

In order to find a globally optimal solution, repeatedly choose locally optimal solutions!

Hint: Whenever you have a choice, take the *greedy* option.
Example: Making Change

Task:

Given a list of coin denominations and an amount of change, can you think of a Greedy algorithm that produces the change using the smallest number of coins?

Denominations= 1p, 2p, 5p, 10p.

Hint: think about various amounts of change.
Making Change Greedy Strategy

**Greedy Strategy:** Repeatedly include in the solution the largest coin whose value doesn’t exceed the remaining amount.

Examples:
- Change = 15p, Solution = 10p, 5p
- Change = 13p, Solution = 10p, 2p, 1p
- Change = 9p, Solution = 5p, 2p, 2p

Question: Does this algorithm always work?
Making Change: Python

while amount != 0 :

    # Give a 10p Coin
    if amount > 9 :
        print("10p ")
        amount = amount - 10

    # Give a 5p Coin
    elif amount > 4 :
        print("5p ")
        amount = amount - 5

    ....

    # Give a 1p Coin
    elif amount > 0 :
        print("1p ")
        amount = amount - 1
Making Change Doesn’t Always Work

Consider Denominations = 6p, 5p, 2p, 1p

Does our algorithm give optimal solution?
Greedy Making Change Doesn’t Always Work

Consider Denominations = 6p, 5p, 2p, 1p

The Greedy approach doesn’t work!

For example: Change = 10p, Greedy Gives = 6p, 2p, 2p.

Optimal Solution = 5p, 5p!
Dynamic Programming
Dynamic Programming: Overall Idea

*Greedy* fails as we need to consider many possible options for the solution: We need some sort of bookkeeping over possible solutions.

Solution: *Dynamic Programming*

Construct a table of possible solutions in a bottom-up manner, then choose optimal.
Making Change Revisited

Denominations = 1p, 2p, 5p, 6p
Example: Change = 10p

Dynamic Programming Table -- 3 coins is optimal:

<table>
<thead>
<tr>
<th>Amount</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>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
Computing The Table

Step 1. Fill first column with 0’s

Step 2. Fill first row with solutions for first denomination.

<table>
<thead>
<tr>
<th>Amount</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>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing The Table

Step 3. In a “Bottom up” manner fill grid based on following:

Compare the number directly above with result of paying out coin and using solution for smaller value (minus coin).

<table>
<thead>
<tr>
<th>Amount</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>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2p</td>
<td>0</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Above: 1 vs 1+Cost of sub problem: --
Can’t Happen!

Can’t pay out 2p so:
Answer = 1
Computing The Table

Step 3. In a “Bottom up” manner fill grid based on following:

Compare the number directly above with result of paying out coin and using solution for smaller value (minus coin).

<table>
<thead>
<tr>
<th>Amount</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>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2p</td>
<td>0</td>
<td>1</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Above: 2 vs 1+Cost of sub problem: 1
Answer = 1
Computing The Table

Step 3. In a “Bottom up” manner fill grid based on following:

Compare the number directly above with result of paying out coin and using solution for smaller value (minus coin).

<table>
<thead>
<tr>
<th>Amount</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>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Above: 5 vs 1+Cost of sub problem: 1
Answer = 3
Computing The Table

Step 3. In a “Bottom up” manner fill grid based on following:

Compare the number directly above with result of paying out coin and using solution for smaller value (minus coin).

<table>
<thead>
<tr>
<th>Amount</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>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>?</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6p</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Above: 3 vs 1+Cost of sub problem: 1

Answer = 1
Getting The Solution

Table only gives number of coins. To compute which coins:

Step 1: Start in square for solution.
Step 2: If square above is same move up to that square. If not, pay out coin for that row and move that many columns left in that row.

<table>
<thead>
<tr>
<th>Amount</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>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1p</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>2p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>5p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6p</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Result = 5p, 5p.
Dynamic Programming: Task

Solve the following via *Dynamic Programming*:

Denominations = 1p, 2p, 5p, 10p
Change to be given = 14p

Show the table with optimal number of coins, and how to compute the coins to pay.
A Challenge Task
To Discuss: The Knapsack Problem

A robber has a choice of stealing several items, obviously they want to maximize their profit. Unfortunately, they can only carry 15kg. Which items should they choose?

Notes:

-- Each item has a value and weight:

- Item 1: v = £4, w = 12kg
- Item 2: v = £2, w = 1kg
- Item 3: v = £2, w = 2kg
- Item 4: v = £1, w = 1kg
- Item 5: v = £10, w = 4kg
(Harder) Tasks

1. Give a greedy algorithm to solve the knapsack problem.

2. Does this greedy algorithm always work?

3. Compute the Dynamic Programming Table for the given knapsack problem.
   
   Hint 1: use table on next slide as a starting point!
   
   Hint 2: Each item can only be taken once, hence if item is taken need to move up a row too!

4. How do we compute the solution from this table?
### Hint: Knapsack Table

*Weight* Across, *Item* Number Down, *Total Value* in each square.

Think about what comparisons? Take or do not take?

<table>
<thead>
<tr>
<th></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>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Greedy (by value) Knapsack Doesn’t Always Work

Consider a robber who can carry 15kg and has the choice over the following items:

- Item 1: $v = £4$, $w = 11$kg
- Item 2: $v = £2$, $w = 1$kg
- Item 3: $v = £2$, $w = 2$kg
- Item 4: $v = £1$, $w = 1$kg
- Item 5: $v = £10$, $w = 4$kg
Knapsack Solution: Table

Value in square above
vs
Item Value + (value in square of row above - item weight)

<table>
<thead>
<tr>
<th></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>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
Knapsack Items to Choose

Which items to take can be computed in a similar manner to the making change example, except we can only take item once. (Same value move up a row, otherwise take item).

<table>
<thead>
<tr>
<th></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>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>
Conclusion

- Algorithm Design Techniques.
  - Divide and Conquer (Merge Sort, Binary Search)
  - Greedy (Making Change, Knapsack)
  - Dynamic Programming (Making Change, Knapsack)

- Next Week: Data Representation