HMC CS Professor Develops New Search Engine

Claremont, CA: A Harvey Mudd CS Professor has developed a sophisticated new search engine called Giigle that gives entirely random responses to search queries. “It’s not very useful, but it is kinda funny,” said one dedicated user. “I put in a query for ‘pumpkin bread recipes’ and the top hit was a website on smelly socks. It made me chuckle.” The competition, however, is not laughing. “We’ve been working on this idea for over 5 years now,” said a VP for a large search engine company. “The idea was ours and we plan to sue for intellectual property infringement.” Most experts believe that such a lawsuit will not be successful. “We’ve done some research on this,” said one prominent IP attorney. “When you do a websearch on ‘search engine intellectual property’ the only thing that comes up are hits about spam donuts, how to change the air filter in your car, and a blog on cool things to make out of rotten pumpkins. As far as we can tell, there is no precedent for suing for this kind of thing.” A more likely scenario, experts agree, is that Giigle will be bought out for a large sum of money by a major competitor. “This is a big idea,” said the HMC professor. “They’ll have to offer me a three-figure sum for me to even consider it.”
Learning Goals
• Motivate the need for “care packages” in recursion
What is an alignment anyway?

<table>
<thead>
<tr>
<th>Alignment representation</th>
<th>S1  GCCTGG-</th>
<th>GCCTGG → ACCTGG</th>
<th>(change G to A) (keep the C) (keep the C)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S2  ACC-GGA</td>
<td>ACCTGG → ACCGG</td>
<td>(delete the T) (keep the G) (keep the G)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACCGG → ACCGGA</td>
<td>(insert A)</td>
</tr>
</tbody>
</table>
Next time: returning the alignment

```python
>>> align_score("GCCTGG", "ACCGGA", -4, dnamat)  # align_score only returns score
11
```

```python
>>> align("GCCTGG", "ACCGGA", -4, dnamat)  # align returns score and alignment
[11, 'gCCtGG-', 'aCC-GGa']
```
def change(amount, denominations):
    '''Returns the least number of coins required to make the given amount using the list of provided denominations.''
    if amount == 0:
        return 0
    elif denominations == []:
        return float('inf')
    elif denominations[0] > amount:
        return change(amount, denominations[1:])
    else:
        use_it = 1 + change(amount - denominations[0], denominations)
        lose_it = change(amount, denominations[1:])
        return min(use_it, lose_it)

>>> change(42, [25, 21, 1])
2

>>> show_change(42, [25, 21, 1])
[2, [21, 21]]
def show_change( amount, denominations ):
    '''Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.'''

>>> show_change(42, [25, 21, 1])
[2, [21, 21]]
def show_change( amount, denominations ):
    '''Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.''
    if amount == 0:
        
elif denominations == []:

>>> show_change(42, [25, 21, 1])
[2, [21, 21]]
def show_change(amount, denominations):
    '''Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.''
    if amount == 0:
        return [0, []]
    elif denominations == []:
        return [float('inf'), []]
    return [float('inf'), []]
def show_change( amount, denominations ):
    '''Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.'''
    if amount == 0:
        return [0, []]
    elif denominations == []:
        return [float('inf'), []]
    elif denominations[0] > amount:
        return [float('inf'), []]
    else:
        min_coins = float('inf')
        for i in range(len(denominations)):
            remaining = amount - denominations[i]
            if remaining >= 0:
                sub_result = show_change(remaining, denominations[i+1:]
                                          + denominations[:i])
                if sub_result[0] < min_coins:
                    min_coins = sub_result[0]
                    sub_coins = sub_result[1] + [denominations[i]]
        return [min_coins, sub_coins]

>>> show_change(42, [25, 21, 1])
[2, [21, 21]]
def show_change( amount, denominations ):
    '''Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.'''
    if amount == 0:
        return [0, []]
    elif denominations == []:
        return [float('inf'), []]
    elif denominations[0] > amount:
        return show_change(amount, denominations[1:])

>>> show_change(42, [25, 21, 1])
[2, [21, 21]]
def show_change( amount, denominations ):
    """Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.""
    if amount == 0:
        return [0, []]
    elif denominations == []:
        return [float('inf'), []]
    elif denominations[0] > amount:
        return show_change(amount, denominations[1:])
    else:
        use_it = show_change(amount - denominations[0], denominations)
        lose_it = show_change(amount, denominations[1:]),

>>> show_change(42, [25, 21, 1])
[2, [21, 21]]
def show_change(amount, denominations):
    '''Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.'''
    if amount == 0:
        return [0, []]
    elif denominations == []:
        return [float('inf'), []]
    elif denominations[0] > amount:
        return show_change(amount, denominations[1:])
    else:
        use_it = show_change(amount - denominations[0], denominations)

        lose_it = show_change(amount, denominations[1:])

        if use_it[0] <= lose_it[0]:
            return use_it
        else:
            return lose_it

>>> show_change(42, [25, 21, 1])
[2, [21, 21]]
def show_change(amount, denominations):
    '''Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.''
    if amount == 0:
        return [0, []]
    elif denominations == []:
        return [float('inf'), []]
    elif denominations[0] > amount:
        return show_change(amount, denominations[1:])
    else:
        use_it = show_change(amount - denominations[0], denominations)
        use_it = [use_it[0] + 1, [denominations[0]] + use_it[1]]
        lose_it = show_change(amount, denominations[1:])
        if use_it[0] <= lose_it[0]:
            return use_it
        else:
            return lose_it

>>> show_change(42, [25, 21, 1])

[2, [21, 21]]
change and show_change
return different types

```python
>>> change(42, [25, 21, 1])
2

>>> show_change(42, [25, 21, 1])
[2, [21, 21]]
```
A common mistake...

```python
def change( amount, denominations ):
    '''Returns the least number of coins required to make the given
    amount using the list of provided denominations.'''
    if amount == 0: return 0
    elif denominations == []: return float('inf')
    elif denominations[0] > amount:
        return change(amount, denominations[1:])
    else:
        use_it = 1 + change(amount - denominations[0], denominations)
        lose_it = change(amount, denominations[1:])
        return min(use_it, lose_it)

def show_change( amount, denominations ):
    '''Takes an integer amount and a list of possible denominations.
    Returns a list containing two elements. First the minimum number of coins
    required, and second a list of the actual coins used in that solution.'''
    if amount == 0: return [0, []]
    elif denominations == []: return [float('inf'), []]
    elif denominations[0] > amount:
        return show_change(amount, denominations[1:])
    else:
        use_it = show_change(amount - denominations[0], denominations)
        use_it = [ use_it[0] + 1, [denominations[0]] + use_it[1] ]
        lose_it = show_change(amount, denominations[1:])
        if use_it[0] <= lose_it[0]:
            return use_it
        else:
            return lose_it
```

TypeError

schmype error!
Another example with `subset`

def subset( target, lst ):
    '''Returns True if there exists a subset of lst that adds up to target and returns False otherwise.'''
    if target == 0:
        return True
    elif lst == []:
        return False
    elif lst[0] > target:
        return subset(target, lst[1:])
    else:
        use_it = subset(target - lst[0], lst[1:])
        lose_it = subset(target, lst[1:])
        return use_it or lose_it

>>> subset(9, [2,3,5])
False
show_subset

def show_subset( target, lst ):
    '''Returns a care package indicating whether there is a subset of lst that adds up to target.'''

>>> show_subset(9,[2,3,5])
[False, []]

>>> show_subset(10,[2,3,5])
[True, [2, 3, 5]]
def show_subset( target, lst ):  
    '''Returns a care package indicating whether there is a subset of lst that adds up to target.''
    if target == 0:  
        return [True, []]  
    elif lst == []:  
        return [False, []]  
    elif lst[0] > target:  
        return show_subset(target, lst[1:])  
    else:  
        use_it = show_subset(target - lst[0], lst[1:])  
        if use_it[0]:  
            # don't add if False  
            use_it = [use_it[0], [lst[0]] + use_it[1]]  

        lose_it = show_subset(target, lst[1:])  
        if use_it[0]:  
            return use_it  
        else:  
            return lose_it

>>> show_subset(9, [2, 3, 5])  
[False, []]  

>>> show_subset(10, [2, 3, 5])  
[True, [2, 3, 5]]
A final example with LCS

```python
def LCS(S1, S2):
    '''Return the length of the longest common subsequence of strings S1 and S2.'''
    if S1 == '' or S2 == '':
        return 0
    elif S1[0] == S2[0]:
        return 1 + LCS(S1[1:], S2[1:])
    else:
        option1 = LCS(S1[1:], S2)
        option2 = LCS(S1, S2[1:])
        return max(option1, option2)

>>> fancyLCS("human", "chimpanzee")
[4, 'h#man', '#h#man###']
```
def fancyLCS(S1, S2):
    """Returns a care package of the form [number, string1, string2] where number is the LCS length and string1 and string2 are pounded out versions of S1 and S2."""

    if S1 == '':
        if S2 == '':
            elif S1[0] == S2[0]:
                else:

                    >>> fancyLCS("human", "chimpanzee")
                    [4, 'h#man', '#h#m#an###']
def fancyLCS(S1, S2):
    """Returns a care package of the form [number, string1, string2] where number is the LCS length and string1 and string2 are pounded out versions of S1 and S2."""

    if S1 == '':
        return [0, '', len(S2)*'#']
    elif S2 == '':
        return [0, len(S1)*'#', '']
    elif S1[0] == S2[0]:
        match = fancyLCS(S1[1:], S2[1:];)
        match = [1+match[0], S1[0]+match[1], S2[0]+match[2]]
        return match
    else:
        option1 = fancyLCS(S1[1:], S2)
        option1 = [option1[0], '#'+option1[1], option1[2]]

        option2 = fancyLCS(S1, S2[1:])
        option2 = [option2[0], option2[1], '#'+option2[2]]

        if option1[0] > option2[0]:
            return option1
        else:
            return option2
Homework: superLCS

```python
>>> superLCS("human", "chimpanzee")
[4, '-hu-m-an---', 'ch-impanzee']

>>> superLCS("A", "AT")
[1, 'A-', 'AT']

>>> superLCS("CG", "G")
[1, 'CG', '-G']
```
Homework: align

```python
>>> align("GCCTGG","ACCGGA",-4,dnamat)
[11, 'gCCtGG-', 'aCC-GGa']

gCCtGG-
aCC-GGa
```
Population power. Extreme throughput. $1,000 human genome.

The HiSeq X Ten is a set of ten ultra-high-throughput sequencers, purpose-built for large-scale human whole-genome sequencing.
UK Biobank data on 500,000 people paves way to precision medicine

Treatments tailored to individuals rely on the wisdom of crowds.
Read mapping is really an alignment problem

Reference: CGCCGAATAGAACGGGAGCCGG-CGAGGCGGATAAA
Read: ggCGTGAG-CGGACGAGcaat