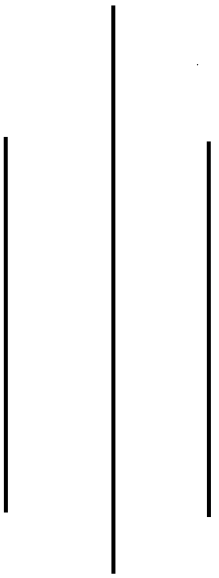
KATHMANDU UNIVERSITY

DHULIKHEL, KAVRE



Subject: COMP 314: Algorithms and Complexity

Lab no: 3

Submitted By:

Name: Ayush Kumar Shah
Roll no: 44
Group: CE 3rd year 2nd sem
Level: UNG

Submitted To:

Dr. Bal Krishna Bal

Date of Submission: 14/06/2018

Lab 3 – Understanding Greedy Algorithms via implementation in Python codes

Activity selection problem
http://www.geeksforgeeks.org/greedy-algorithms-set-1-activity-selection-problem/

Huffman coding
https://rosettacode.org/wiki/Huffman_coding#Python

What you are supposed to submit:
1. Description of the problem definitions – Activity Selection problem and Huffman coding
2. Codes – you may reuse them from the above but make sure they are in working state.
3. Test results – provide few inputs and their respective outputs.
4. Compile all 1-3 above in one PDF document and submit.

Greedy Algorithms

Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. Greedy algorithms are used for optimization problems. An optimization problem can be solved using Greedy if the problem has the following property:

At every step, we can make a choice that looks best at the moment, and we get the optimal solution of the complete problem.

If a Greedy Algorithm can solve a problem, then it generally becomes the best method to solve that problem as the Greedy algorithms are in general more efficient than other techniques like Dynamic Programming

1. Activity Selection Problem

Let us consider the Activity Selection problem as our first example of Greedy algorithms. Following is the problem statement.
You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.
Example:

Example 1 : Consider the following 3 activities sorted by

by finish time.

start[] = {10, 12, 20};

finish[] = {20, 25, 30};

A person can perform at most **two** activities. The

maximum set of activities that can be executed

is {0, 2} [These are indexes in start[] and

finish[]]

Example 2 : Consider the following 6 activities

sorted by by finish time.

start[] = {1, 3, 0, 5, 8, 5};

finish[] = {2, 4, 6, 7, 9, 9};

A person can perform at most **four** activities. The

maximum set of activities that can be executed

is {0, 1, 3, 4} [These are indexes in start[] and

finish[]]

The greedy choice is to always pick the next activity whose finish time is least among the remaining activities and the start time is more than or equal to the finish time of previously selected activity. We can sort the activities according to their finishing time so that we always consider the next activity as minimum finishing time activity.

1) Sort the activities according to their finishing time
2) Select the first activity from the sorted array and print it.
3) Do following for remaining activities in the sorted array.
…….a) If the start time of this activity is greater than or equal to the finish time of previously selected activity then select this activity and print it.

Python code:

activity.py - D:/KUCE/course_materials/6th_sem/COMP 314 Algorithms and Complexity/Lab ...

File Edit Format Run Options Window Help

```
"""Prints a maximum set of activities that can be done by a
single person, one at a time"""
# n --> Total number of activities
# s[]--> An array that contains start time of all activities
# f[] --> An array that contains finish time of all activities

def printMaxActivities(s , f ):
    n = len(f)
    print ("The following activities are selected")

    # The first activity is always selected
    i = 0
    print (i),

    # Consider rest of the activities
    for j in range(n):

        # If this activity has start time greater than
        # or equal to the finish time of previously
        # selected activity, then select it
        if s[j] >= f[i]:
            print (j),
            i = j

# Driver program to test above function
#s = [1 , 3 , 0 , 5 , 8 , 5]
#f = [2 , 4 , 6 , 7 , 9 , 9]
print ("enter start time of the activities")
s=list(map(int,input().split()))

print ("enter finish time of the activities")
f=list(map(int,input().split()))

#if not sorted applying bubble sort
for i in range(len(f)):
    for j in range(len(f)-i-1):
        if f[j]>f[j+1]:
            f[j],f[j+1],s[j],s[j+1]=f[j+1],f[j],s[j+1],s[j]

printMaxActivities(s , f)
```

Output:

```
enter start time of the activities
0 5 8 5 1 3
enter finish time of the activities
6 7 9 9 2 4
The following activities are selected
0
1
3
4
>>>
```

2. Huffman coding

Huffman encoding is a way to assign binary codes to symbols that reduces the overall number of bits used to encode a typical string of those symbols.

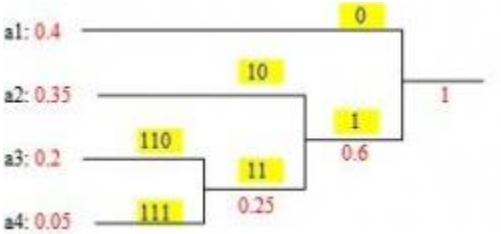
For example, if you use letters as symbols and have details of the frequency of occurrence of those letters in typical strings, then you could just encode each letter with a fixed number of bits, such as in ASCII codes. You can do better than this by encoding more frequently occurring letters such as e and a, with smaller bit strings; and less frequently occurring letters such as q and x with longer bit strings.

Any string of letters will be encoded as a string of bits that are no-longer of the same length per letter. To successfully decode such as string, the smaller codes assigned to letters such as 'e' cannot occur as a prefix in the larger codes such as that for 'x'.

If you were to assign a code 01 for 'e' and code 011 for 'x', then if the bits to decode started as 011... then you would not know if you should decode an 'e' or an 'x'.

The Huffman coding scheme takes each symbol and its weight (or frequency of occurrence), and generates proper encodings for each symbol taking account of the weights of each symbol, so that higher weighted symbols have fewer bits in their encoding.

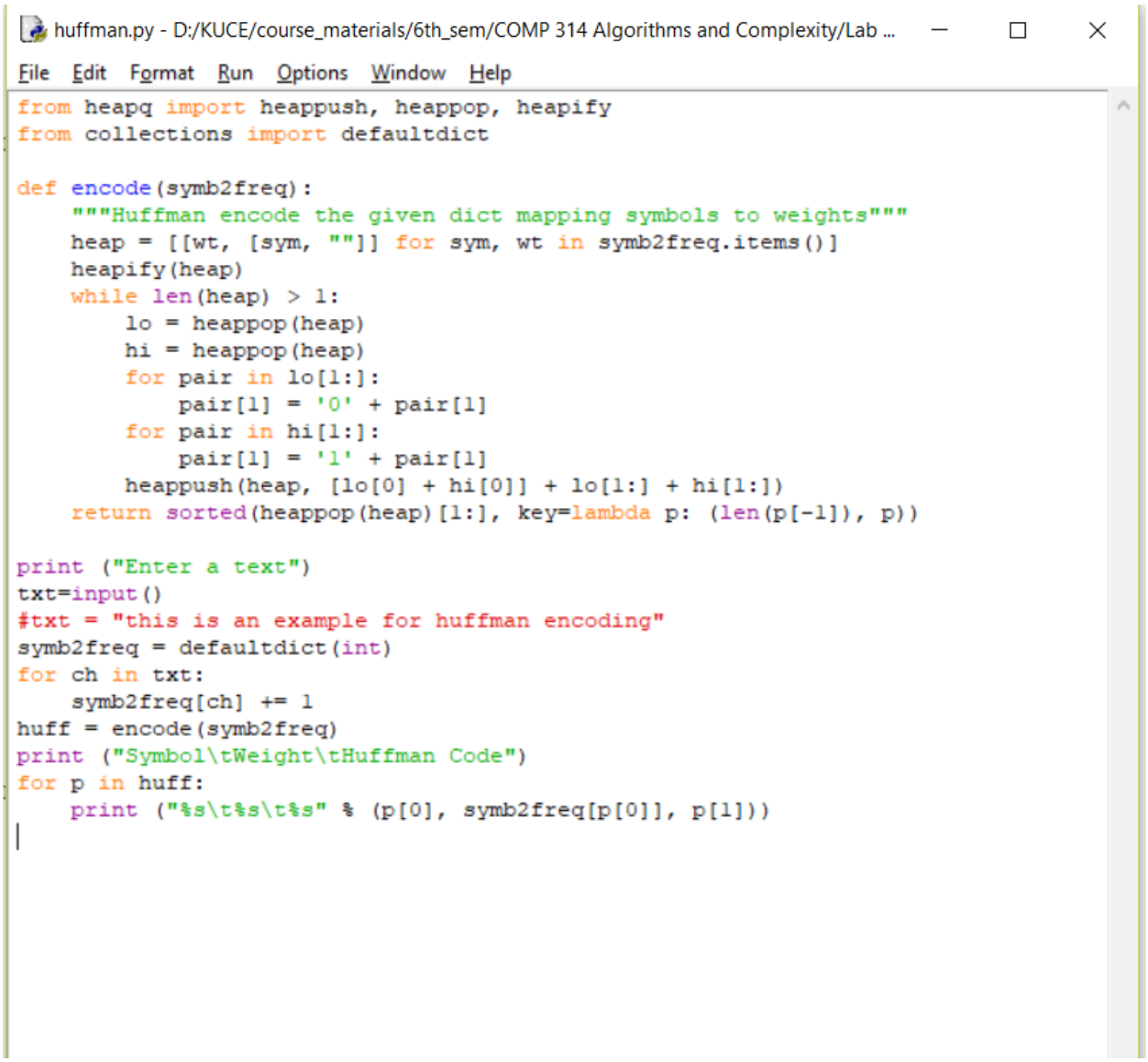
A Huffman encoding can be computed by first creating a tree of nodes:



1. Create a leaf node for each symbol and add it to the priority queue.
2. While there is more than one node in the queue:
    1. Remove the node of highest priority (lowest probability) twice to get two nodes.
    2. Create a new internal node with these two nodes as children and with probability equal to the sum of the two nodes' probabilities.
    3. Add the new node to the queue.
3. The remaining node is the root node and the tree is complete.

Traverse the constructed binary tree from root to leaves assigning and accumulating a '0' for one branch and a '1' for the other at each node. The accumulated zeros and ones at each leaf constitute a Huffman encoding for those symbols and weights:

Python code:


huffman.py - D:/KUCE/course_materials/6th_sem/COMP 314 Algorithms and Complexity/Lab ...
File Edit Format Run Options Window Help
from heapq import heappush, heappop, heapify
from collections import defaultdict
def encode(symb2freq):
"""Huffman encode the given dict mapping symbols to weights"""
heap = [[wt, [sym, ""]] for sym, wt in symb2freq.items()]
heapify(heap)
while len(heap) > 1:
lo = heappop(heap)
hi = heappop(heap)
for pair in lo[1:]:
pair[1] = '0' + pair[1]
for pair in hi[1:]:
pair[1] = '1' + pair[1]
heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
print ("Enter a text")
txt=input()
#txt = "this is an example for huffman encoding"
symb2freq = defaultdict(int)
for ch in txt:
symb2freq[ch] += 1
huff = encode(symb2freq)
print ("Symbol\tWeight\tHuffman Code")
for p in huff:
print ("%s\t%s\t%s" % (p[0], symb2freq[p[0]], p[1]))

Output:

```
Enter a text
My name is Ayush Kumar Shah
Symbol  Weight  Huffman Code
        5       00
a       3       010
h       3       011
m       2       1011
s       2       1101
u       2       1110
y       2       1111
A       1       10000
K       1       10001
M       1       10010
S       1       10011
e       1       10100
i       1       10101
n       1       11000
r       1       11001
>>>
```