



MERRIMACK COLLEGE

CSC6023 - Advanced Algorithms

# Greedy algorithms

# Greedy Algorithms



This is a very fast paced course, After this four week topic and we are at the half the course. So, let's see the greedy algorithms today.



MERRIMACK COLLEGE

# Agenda

## Week 4

## Presentation

### Greedy Algorithms

- The Basics
  - a. Suboptimal solution
  - b. Difference between Brute Force and Greedy
- Computer Algorithms:
  - a. Minimal Number of Coins

### Examples

- First Example - Egyptian Fractions
  - a. The Problem
  - b. The Greedy Solution
- Second Example - Knapsack Problem
  - a. The Problem
  - b. The Greedy Solution



## The Basics

# Basics of Greedy Algorithms



MERRIMACK COLLEGE

### Do what you can now, and what you must later

- Sophisticated approaches such as Dynamic Programming can be overkill, especially for optimization problems
- Greedy algorithms is a simpler approach that tries to solve the current stage of the problem with local information, and the solution of the next stage with need to solve it based on local information again
- Greedy algorithms do not yield optimal solutions for all problems
  - suboptimal solution, that may be enough or the best given time/space constraints, or even the best known strategy yet

## The Basics

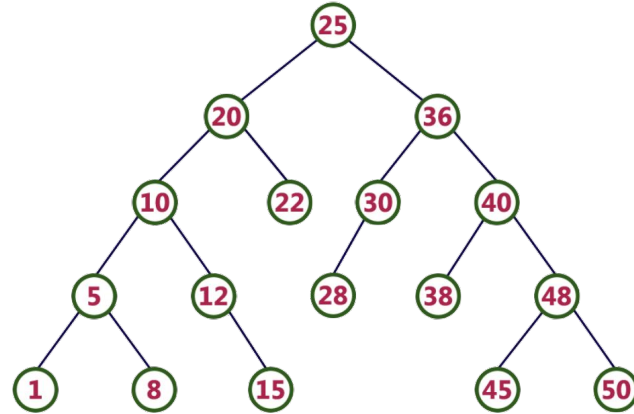
# Basics of Greedy Algorithms



MERRIMACK COLLEGE

### Find the greatest sum

- You try to get the maximum at each choice



- From the first you choose 36
- From the second you choose 40
- From the third you choose 48
- From the fourth you choose 50
- It adds up **199**, which is the best you can do

## The Basics

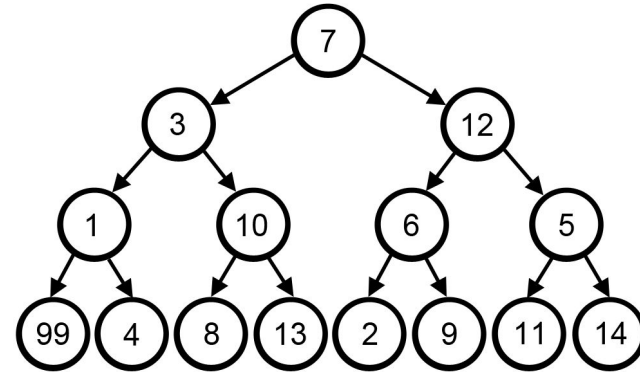
# Basics of Greedy Algorithms



MERRIMACK COLLEGE

### Find the greatest sum

- You try to get the maximum at each choice



- From the first you choose 12
- From the second you choose 6
- From the third you choose 9
- It adds up **34**, which is worse than **110**

## The Basics

# Greedy Algorithms vs. Brute Force Algorithms


---

### From a theoretical point of view

- Brute Force is a simple approach that usually encompass the whole problem, while Greedy tackles only a portion of the problem at a time
- Brute Force necessarily delivers the optimal solution, since it analyzes all possible solutions, while Greedy may not deliver optimal solution
- Brute Force algorithms usually have the worst (sometimes the only) possible complexity, while Greedy usually have a reasonably good complexity
- Brute Force simplifies the solution by simplifying the implementation of the problem definition, while Greedy simplifies the solution by disregarding a complex decomposition of the solution



## Minimal Number of Coins

quarters (\$ .25) 	dimes (\$ .10) 
nickels (\$ .05) 	pennies (\$ .01) 

### A Greedy Approach - a good example

- Given a set of possible coins and a target amount, generate the minimal sized set of coins that adds up this amount
- At every step try to put the largest amount coin
- For example, how to minimally add up 81 cents?
- 5 coins



remaining	coin
81	25
56	25
31	25
6	5
1	1





## Minimal Number of Coins



### A bad example

- Things can go very wrong for some amounts and sets of possible coins
- At every step try to put the largest amount coin
- For example, how to minimally add up 81 cents?
- 9 coins



remaining	coin
81	25
56	25
31	25
6	1
5	1
4	1
3	1
2	1
1	1

## Minimal Number of Coins



### A Non Greedy Approach

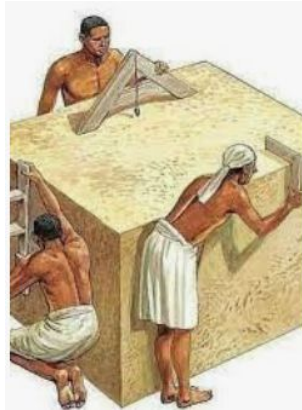
- Let's say you can try a better solution exhaustively (for example, Brute Force) to find an optimal solution
- Try all combinations deriving changes from the Greedy changing a choice at a time
- For example, how to minimally add up 81 cents?
- 6 coins



remaining	coin
81	25
56	25
31	10
21	10
11	10
1	1

## Examples

# Egyptian Fractions



MERRIMACK COLLEGE

## The problem

- Any irreducible rational number (a number that can be expressed in a ratio of two Integers that are not multiples) can be expressed as a sum of unit fractions (rational number where the numerator is 1)
  - For example:  $\frac{5}{6} = \frac{1}{2} + \frac{1}{3}$
- This is a very old problem that was as the name says very much used by the ancient Egyptians
  - Solutions for this problem were introduced by Leonardo Fibonacci in 1202
    - For example  $\frac{7}{15} = \frac{1}{3} + \frac{1}{8} + \frac{1}{120}$

Among the possible solutions, Fibonacci's Greedy Algorithm was the best solution

# Application and History of Egyptian Fractions

- Rhind Mathematical Papyrus acquired in mid 1800s
- Dates to about 1500 BC
- Can be used for equal division of food:
  - 8 people sharing 5 pizzas
  - $\frac{5}{8} = \frac{1}{2} + \frac{1}{8}$ 
    - Each person gets  $\frac{1}{2}$  pizza and  $\frac{1}{8}$  pizza
- This seems silly but the first example from the Rhind papyrus was like this:
  - Varying numbers of loaves divided by ten men



[https://www.britishmuseum.org/collection/object/Y\\_EA10057](https://www.britishmuseum.org/collection/object/Y_EA10057)

## Examples

# Egyptian Fractions



MERRIMACK COLLEGE

## The solution - A Greedy Approach

- Starts with the largest possible fitting unit fraction, then deal with the remainder (recursively)
  - For example:  $7/15$ 
    - $1/2$  won't fit,  $1/3$  fits!
  - $7/15 = 1/3 + ? \Rightarrow 7/15 = 5/15 + 2/15$ 
    - Apply it to  $2/15$ :
      - $1/2, 1/3, 1/4, 1/5, 1/6$ , and  $1/7$  won't fit,  $1/8$  fits!
  - $7/15 = 1/3 + 1/8 + ? \Rightarrow 56/120 = 40/120 + 15/120 + 1/120$ 
    - Apply it to  $1/120$  (elementary solution)
  - $7/15 = 1/3 + 1/8 + 1/120$  (final solution)
- Basically divide the denominator by the numerator and round up!

## Examples

# Egyptian Fractions



## The Greedy solution - Python implementation

```
# Egyptian fraction Greedy
from math import ceil

# n is the numerator, d is the denominator
def egyptian(n, d):

    print("The Egyptian Fraction of {}/{}".format(n, d))
    ans = []
    # while numerator is not 0
    while (n > 0):
        x = ceil(d / n)          # compute the minimal larger denominator
        ans.append(x)           # hold it to the numerator list
        n, d = x * n - d, d * x # update the remainder to n and d
    for a in ans:
        print("1/{}".format(a), end=" ")
```



MERRIMACK COLLEGE

- The ceiling function to find the larger denominator fitting is the basic Greedy operation

## Examples

# Egyptian Fractions



## The Greedy solution - Python implementation

```
def main():
    print("Greedy Algorithm to Compute Egytian Fractions")
    stay = True
    while stay:
        n = int(input("Enter the numerator: "))
        d = int(input("Enter the denominator: "))
        egyptian(n, d)
        stay = ("n" != (input("\nCompute another? (no to stop) ")+" ")[0])

main()
```

- The ***egyptian(n, d)*** function is repeatedly called
- Full code available [here](#)



## Greedy Algorithms

# Egyptian Fractions



### Task #1 for this week's In-class exercises

- Run the Egyptian Fractions for the following examples:
  - $5/6$ ,  $7/15$ ,  $23/34$ ,  $121/321$ ,  $5/123$
  - Write remarks in your new code with the solution of such cases

Go to IDLE and edit the code with the required remarks  
Save your program in a .py file and submit it in the appropriate delivery room



MERRIMACK COLLEGE

Deadline: This Friday 11:59 PM EST



## Examples

# Egyptian Fractions



MERRIMACK COLLEGE

## Limits of the Implementation

```
# Egyptian fraction Greedy
from math import ceil

# n is the numerator, d is the denominator
def egyptian(n, d):

    print("The Egyptian Fraction of {}/{}".format(n, d))
    ans = []
    # while numerator is not 0
    while (n > 0):
        x = ceil(d / n)          # compute the minimal larger denominator
        ans.append(x)           # hold it to the numerator list
        n, d = x * n - d, d * x # update the remainder to n and d
    for a in ans:
        print("1/{}".format(a), end=" ")
```

- The precision of the operation  $n = x * n - d$
- Try it out for the fraction 5/121

## Examples

# Egyptian Fractions

$$\frac{5}{121} = \frac{1}{25} + \frac{1}{757} + \frac{1}{763\,309} + \frac{1}{873\,960\,180\,913} + \frac{1}{1\,527\,612\,795\,642\,093\,418\,846\,225}$$

**WE RISK ALL IN BEING  
TOO GREEDY**

JEAN DE LA FONTAINE

« On hasard  
de perdre  
en voulant  
trop gagner. »

Jean de La Fontaine  
Extrait de la fable *Le Héron*

## Limits of the Approach

- The fraction  $5/121$  if computed correctly should be

- A better solution could be

$$\frac{5}{121} = \frac{1}{33} + \frac{1}{121} + \frac{1}{363}$$

- Remember  $1/25$  is larger than  $1/33$  ...



MERRIMACK COLLEGE

# Egyptian Fractions

### Task #2 for this week's In-class exercises

- Run the Egyptian Fractions for the following example  $5/121$  using the code
  - Check it out manually the expected result, which should be:

$$\frac{5}{121} = \frac{1}{25} + \frac{1}{757} + \frac{1}{763\,309} + \frac{1}{873\,960\,180\,913} + \frac{1}{1\,527\,612\,795\,642\,093\,418\,846\,225}$$



Write it down why you think it didn't work as intended  
Save your text in a file and submit it in the appropriate delivery room



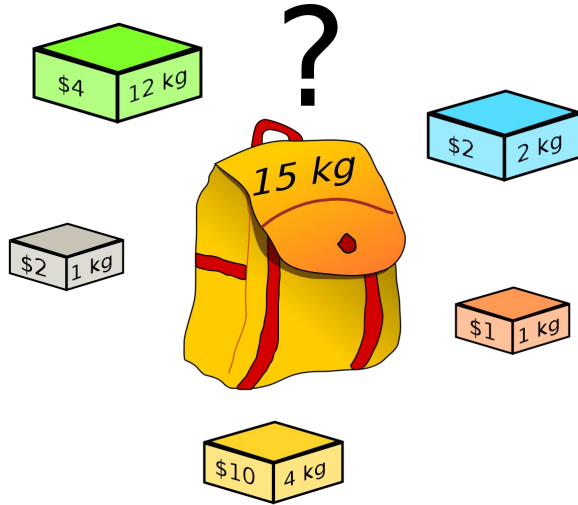
MERRIMACK COLLEGE

Deadline: This Friday 11:59 PM EST

**5 Minute Break**

## Examples

# Knapsack Problem



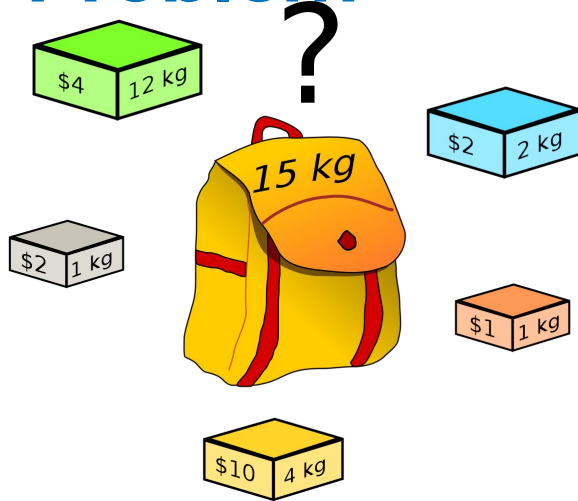
## The Problem

- Given a set of items  $i$  with a weight  $w_i$  and an associated value  $v_i$
- Given a knapsack with a limited weight capacity
- Which items can be stored in the knapsack carrying the maximum overall value?
  - Formal definition from Wikipedia [here](#)
  - Several problems (including the minimal number of coins) are variations of the knapsack problem
- While the optimal solution is not known, the Greedy Algorithms delivers reasonable results



## Examples

# Knapsack Problem



MERRIMACK COLLEGE

White board example

## The Solution - Brute Force approach

- Generate all possible combinations of items
  - If the problem is unbounded (any number of object is available) keep on piling up objects until the overall weight is about to be exceed
- Keep checking the maximum value of each combination
  - Optimal solution, but a very high cost to generate all possible combinations
- This kind of problem complexity is not able to be described by a polynomial formula
  - Therefore it is called a Nondeterministic Polynomial (NP) problem, and this has been formally proved
    - A NP-complete problem
    - There is no algorithm that is both fast (polynomial or better) and correct in all cases
    - There is no known polynomial algorithm that can verify that a solution is optimal

## Examples

# Knapsack Problem

- Compute item's ratio between weight and value
- Sort the elements by the ratio (non decrescent)
- Repeatedly pick the item with highest ratio that fits the maximum weight capacity

## The Solution - Greedy approach

```
# knapsack unbounded – Greedy approach

def knapsack(v, w, cap):
    rwv = []          # triplet ratio, weight, value, index
    for i in range(len(v)):
        rwv.append([v[i]/w[i], w[i], v[i], i])
    rwv.sort(reverse=True)  # sort from high to low rate
    ans = []             # the list of added items
    tw = 0               # total weight
    found = True
    while (found):       # until no fitting item is found
        found = False
        for t in rwv:    # search an item to add
            if (t[1] + tw) <= cap:  # if the item fits
                ans.append(t[3])    # add it
                tw += t[1]
                found = True
                break
    return ans           # returns the list of added items
```



## Examples

# Knapsack Problem

- Get the number of items, plus their value and weight
- Get the capacity
- Call *knapsack*

[Full code here](#)



MERRIMACK COLLEGE

## The Solution - Greedy approach

```
def main():
    items = int(input("Number of distinct items: "))
    values, weights = [], []
    for i in range(items):
        v = int(input("Value of item "+str(i+1)+": "))
        w = int(input("Weight of item "+str(i+1)+": "))
        values.append(v)
        weights.append(w)
    capacity = int(input("Maximum weight (capacity): "))
    answer = knapsack(values, weights, capacity)
    tv, tw = 0, 0
    for a in answer:
        print("Item - Value:", values[a], "- Weight:", weights[a])
        tv += values[a]
        tw += weights[a]
    print("Items:", len(answer), "- Value:", tv, "- Weight:", tw)
```



## Greedy Algorithms

# Knapsack Problem



### Task #3 for this week's In-class exercises

- Run the Knapsack Greedy code for the following cases (list of items - [value,weight])
  - ([5,10] [8,20] [12,30]) - capacity 838
  - ([3,17] [5,23] [7,29] [11,31] [13,37]) - capacity 997
  - ([5,25] [6,36] [7,49] [8,64]) - capacity 250
  - ([5,25] [6,36] [7,49] [8,64]) - capacity 360

Go to IDLE and edit the code with the required remarks  
Save your text in a file and submit it in the appropriate delivery room



MERRIMACK COLLEGE

Deadline: This Friday 11:59 PM EST

## Greedy Algorithms

# Knapsack Problem



### Task #4 for this week's In-class exercises

- Come up with one additional knapsack test case of your own. It must result in more than one type of item being placed in the knapsack. Code it in your .py file after task 3. Be sure to indicate the expected result.

Go to IDLE and edit the code with the required remarks  
Save your text in a file and submit it in the appropriate delivery room

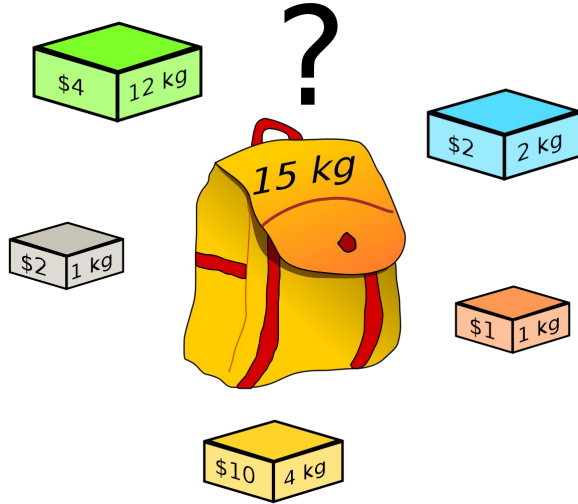


MERRIMACK COLLEGE

Deadline: This Friday 11:59 PM EST

## Examples

# Knapsack Problem



MERRIMACK COLLEGE

## The Limits of the Approach

- Trying it out for:
  - ([5,25] [6,36] [7,49] [8,64]) - capacity 72
    - The program delivers:
      - 2 [5,25] items - Value: 10, Weight: 50
    - A best solution would be:
      - 2 [6,36] items - Value: 12, Weight: 72
- The limitation usually appears when the answer is a small number of items, as small grain problems are usually effectively handled by Greedy
- There is a Dynamic programming algorithm that is more stable to deliver good solutions

## Greedy Algorithms

# Fourth Assignment

```
small,35,3,4,2  
medium,40,4,5,4  
large,45,5,6,5  
jumbo,58,6,6,6
```



MERRIMACK COLLEGE

### Project #4 - this week's Assignment

- Create a program that receives the list of possible named items with the following information:
  - Value (\$), Height (in), Width (in), Depth (in)
- The limit of the optimal solution is expressed by the volume in cubic inches ( $\text{in}^3$ ) and the program has to maximize the value within the cubic limit
- Your program should read textual file with one item kind per line with the information separated by comma, for example this [file](#) lists four items with values 35, 40, 45, and 58 dollars and increasing dimensions
- Your program should read any file with this format (name,value,height,width,depth) per line

# Fourth Assignment



### Project #4 - this week's Assignment

- This program must be your own, do not use someone else's code
- Any specific questions about it, please bring to the Office hours meeting this Friday or contact me by email
- This is a challenging program to make sure you are mastering your Python programming skills, as well as your asymptotic analysis understanding
- Don't be shy with your questions

Go to IDLE and try to program it  
Save your program in a .py file and submit it in the appropriate delivery room

Deadline: Next Monday 11:59 PM EST



That's all for today folks!

---

## This week's tasks

- Discussion: initial post by this Friday/replies by next Tuesday
- Tasks #1, #2, #3 and #4 for the In-class exercises
  - Deadline: Friday 11:59 PM EST
- Quiz #4 to be available this Friday
  - Deadline: Next Monday 11:59 PM EST
- Project #4 assignment
  - Deadline: Next Monday 11:59 PM EST
- Try all exercises seen in class and consult the reference sources, as the more you practice, the easier it gets

## Next week

- Amortized algorithms
- Don't let work pile up!
- Don't be shy about your questions



MERRIMACK COLLEGE

# Have a Great Week!