**CSE 591 Final Course Project** **Sharath Kumar Naeni (110950172)**

Project consists of analyzing two below sub-problems:

1. Coin Change Problem
2. Convex Hull

**The Coin Change Problem:**

This Part of the project falls under the category "Using a systematic method to program and evaluate other algorithms".

In this part, we will consider the well-known problem of "Change-Making", study the naïve recursion solution, efficient Dynamic Programming solution. Using systematic method of "Iterate, Incrementalize, implement" we arrive at an efficient, iterative, and yet clearer solution.

At the end, we compare the clarity, running times of the above three implementations.

(1) the problem, i.e., what's given and what's wanted for your project:

**Change-Making**: Given certain coin denominations, how many ways can you make change for an amount N. This is different from it's well known twin problem of "find minimum coins to make change N".
**Input:** Set of **coin Denominations** [Dollar bills: 1 2 5 10 20] and an amount **N** [e.g. **523**$].
**Output:** Total number of ways to make change **N.**

**Existing Methods:**

**Recursive Approach:** Naïve, straight forward but highly inefficient.
**Dynamic Programming:** Highly efficient, uses Memoization approach.

Goal: Arrive at an iterative approach using III method.

(2) the method that you use to solve the problem:

Systematic Program Design of III method.

We can clearly see that recursive approach takes exponential time in size of input (N).
Initially, we start with recursive approach.
We find the minimal increment and convert recursion part into Iterative implementation.
Later we fill try to find any optimizations possible on implementation part (data structures).

We have used '0-1 knapsack' and 'longest common subsequence' systematic methodologies taught in the class as a reference.

(3) the implementation of the method.

Recursive Part of the Algorithm:

```
def getNumWays(changeTomake, coinValueList, allowedcoinsIndex):

    numOfWays = 0
    # We have reached base case, empty set is a combination. So return 1
    if changeTomake == 0:
        return 1

    if allowedcoinsIndex <= 0:
        return 0


    if changeTomake >= coinValueList[allowedcoinsIndex - 1]:
        numOfWays = getNumWays(changeTomake-coinValueList[allowedcoinsIndex - 1],
coinValueList, allowedcoinsIndex) \
                    + getNumWays(changeTomake, coinValueList, allowedcoinsIndex - 1)
    else:
        numOfWays = getNumWays(changeTomake, coinValueList, allowedcoinsIndex - 1)

    return numOfWays
```

There are mainly two differences for “Change making” recursion and “0-1 Knapsack” recursion we have seen in the class.

0-1 Knapsack recursion:

0-1 knapsack:

$$knap(i,u) = \begin{cases} 0 & \text{if } i = 0 \text{ or } u = 0 \\ knap(i-1,u) & \text{if } i > 0,\ u > 0,\ w_i > u \\ \max(v_i + knap(i-1, u-w_i), \\ \quad knap(i-1,u)) & \text{otherwise} \end{cases}$$

Change-Making:

chngMkng(i, u) ==

0, if i = 0

1, if u = 0

chngMkng (i-1, u), if(value[i] > u)

chngMkng (i, u-value[i]) + chngMkng (i-1, u), otherwise

Differences for above recursions:

1. Minimal increment for knapsack is (i, u) → (i+1, u)

For change-making recurrence: *chngMkng (i, u)* calls two occurrences of *chngMkng (i-1, u)* and one occurrence of *chngMkng (i, u-value[i])*
As '*value[i] > 0*' implying '*u-value[i] >= u-1',*
Minimal increment for chngMkng is also (i, u) → (i+1, u)

But note that for every recursive level knapsack recursion decrements from 'i' to 'i-1' , but for chngMkng 'i' do not decrement for occurrence chngMkng(i, u-value[i])

Above condition means, for computing $i^{th}$ cnhgMkng() the additional array values of $(i-1)^{th}$ iteration(rExt) alone are not sufficient.

2. The aggregator used for knapsack is 'max()' and we use 'sum()' as part of chngMkng.

Apart from the above main differences, chngMkng also have different base conditions. Base conditions play a major role in recursion formulations. The base cases need to be properly implemented for the algorithm to work.

Converting Recursion into Iteration:

Assuming *cnchngExt_(i,u,rExt)= cnchngExt(i+1,u),* we convert the recursion into iteration using while loop as below:

```
def cnchngExt(numCnDenoms, amount):
    global coinDen

    rExt = [0 for u in range(amount+1)]
    i = 0
    while i != numCnDenoms:
        rExt = cnchngExt_(i, amount, rExt)
        i+=1

    return rExt[amount]
```

Now defining **cnchngExt_** :

```
def cnchngExt_(i, amount, rExt):
    global coinDen
    rExt_ = [0 for u in range(amount+1)]
    rExt_[0] = 1
    for c in range(1, amount+1):
        if(c >= coinDen[i]):
            rExt_[c] = rExt_[c-coinDen[i]] + rExt[c]
        else:
            rExt_[c] = rExt[c]

    return rExt_
```

Note that unlike knapsack(), here we would need two additional array values (*rExt and rExt_*) one with additional values of '(i-1)$^{th}$' iteration and one with updated values of 'i$^{th}$' iteration for recurrence '*cnChngExt(i, u-value[i])*'

The auxiliary map ***rExt*** ranges from 0 to u. We need an additional space of array ***rExt*** to maintain. Nothing needs to be done as part of 'Implement' step.

The above derived algorithm is implemented in python and attached as '*CoinChange_Implemented_III_Method.py*' file.
Using the reference[2] of "Java implementation of change making" implemented recursive and Dynamic programming algorithms in python. (*coinchangeNaiveRecurssion.py* , *CoinChange_DynamicProg.py* files)

(4) the results, including the runs and tests you performed

Clarity Comparison of above three implementations:

NaiveRecursion is easy to interpret and clear with simple recursive formulation.

Dynamic Programming approach is simple but counter intuitive.

Derived_III_implementation doesn't have complex matrix to maintain (as Dynamic programming) and is elegant with iterative formulation.

Clarity ordering:
NaiveRecursion >> Derived_III_implementation >> Dynamic Programming

Theoretical analysis of time and space complexities:

Let n be the size of the input. Assuming total available denominations close to 'n'

NaiveRecursion: time( $O(2^n)$ ), Space( O(1) )

Dynamic Programming: time( $O(n^2)$ ), Space( $O(n^2)$ )

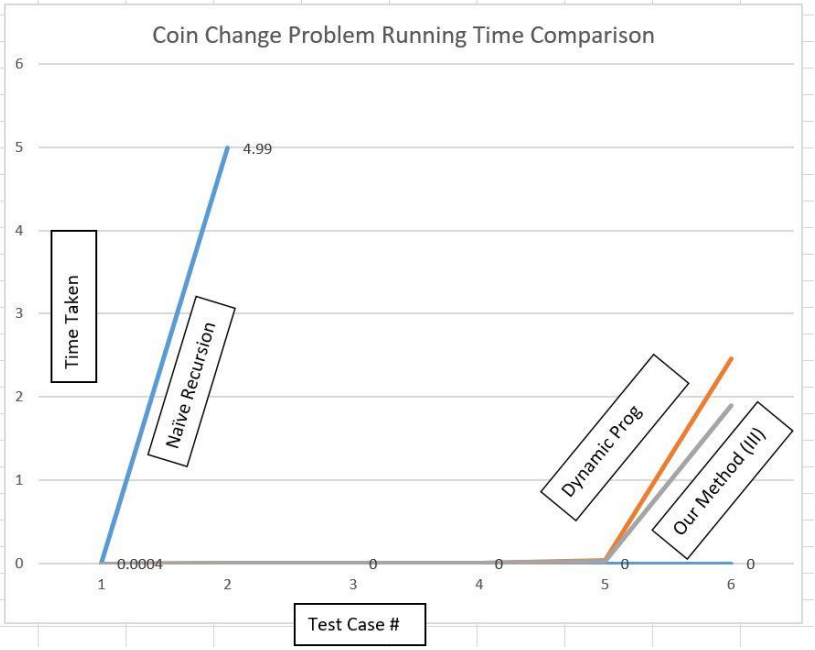
Derived_III_implementation: time( $O(n^2)$ ), Space(O(n))

Test cases performed:
Sample Coin Denominations [1, 2, 5, 10, 20]
Number of runs per each test case to calculate mean: 5

| S.no | Input | Total Ways | NaiveRecursion | DynamicProgram | Systematic Method(Mean) |
|---|---|---|---|---|---|
| 1. | 23 | 54 | 0.0004 | 0.000102 | 0.000104 |
| 2. | 273 | 150920 | 4.99 | 0.0007 | 0.0004 |
| 3. | 573 | 2556640 | 167.29 | 0.0014 | 0.0009 |
| 4. | 1277 | 58764160 | <NA> | 0.0031 | 0.0021 |
| 5. | 15234 | 1127663003369 | <NA> | 0.0346 | 0.0255 |
| 6. | 1023473 | 22861070981928707400 | <NA> | 2.456 | 1.893 |

NA – Indefinitely running for long time.

| Amount | Niave | Dynamic Prog | III methoc |
|---|---|---|---|
| 23 | 0.0004 | 0.000102 | 0.000104 |
| 273 | 4.99 | 0.0007 | 0.0004 |
| 573 | | 0.0014 | 0.0009 |
| 1277 | | 0.0031 | 0.0021 |
| 15234 | | 0.0346 | 0.0255 |
| 1023473 | | 2.456 | 1.893 |




**Conclusion**: For Higher values of input, Derived_III_implementation is out performing in terms of running time.

*References:*


1. http://algorithms.tutorialhorizon.com/dynamic-programming-coin-change-problem/
2. http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/
3. Chapter 4: "Systematic Program Design: From Clarity to Efficiency"


## Convex Hull:

In this part of the project, we will discuss mainly two implementations of Convex Hull Algorithms and compare their clarity and Running times.

(1) the problem, i.e., what's given and what's wanted for your project:

Convex Hull: Given a set P of n points on a plane, find smallest convex polygon containing all points.
Input: Set P of n points on a plane. (2-D convex hull problem)
Output: Set C of h points representing a smallest convex polygon enclosing P.

Some of the convex Hull algorithms:
Jarvis Method: Uses an iterative approach to find vertices of convex hull and is an output sensitive algorithm meaning the running time depends on output set C or more specifically on 'h'. O(nh)

Grahams Method: This method is based on sorting of all n points and forming an enclosed polygon based on the sorting order. Running time: O(nlogn)
Eddy convex hull: Partitioning based algorithm proposed by William F. Eddy from CMU. This method is always upper bounded by O(nh) and in reality, requires very few operations than O(nh)

Akl and Toussaint Method: An iterative approach based on “throw-away pre-processing step”. This preprocessing is related to filtering out points from input data that can never be part of Convex hull. The proposed method had expected running time of linear on n.

There are various new algorithms claiming linear running time O(n). We will mainly focus on Jarvis and Grahams methods.

(2) the method that you use to solve the problem:

Jarvis Method:

Algorithm is described at reference[1] and the implementation from link[6] is used for analysis.

Grahams Method:

Algorithm is described at reference[1] and the implementation from link[7] is used for analysis.

(3) the implementation of the method.

Jarvis Implementation uses below code for determining orientation, instead of finding the angle which is compute intensive:

```
int orientation(Point p, Point q, Point r)
{
	int val = (q.y - p.y) * (r.x - q.x) -
			(q.x - p.x) * (r.y - q.y);

	// collinear case
	if (val == 0)
		return 0;

	// check for clock or counterclock wise
	return (val > 0)? 1: 2;
}
```

Grahams Implementation uses qsort() available from c++ standard library. It uses following compare

```
int compare(const void *vp1, const void *vp2)
{
	Point *p1 = (Point *)vp1;
	Point *p2 = (Point *)vp2;

	// Find orientation
	int o = orientation(p0, *p1, *p2);
	if (o == 0)
		return (distSq(p0, *p2) >= distSq(p0, *p1))? -1 : 1;

	return (o == 2)? -1: 1;
}
```

function.

(4) the results, including the runs and tests you performed:

Clarity comparison of Jarvis and Grahams Method:

Jarvis is a simple iterative approach, while Grahams is a two phased approach where initially sorting happens and then we find the enclosing convex hull iteratively.

Clarity ordering: Jarvis >> Grahams

Running time analysis:

Clearly both the methods have different approaches and Jarvis is an output sensitive algorithm, the running time analysis largely depends on the input data set.

Jarvis method running time : O(nh)

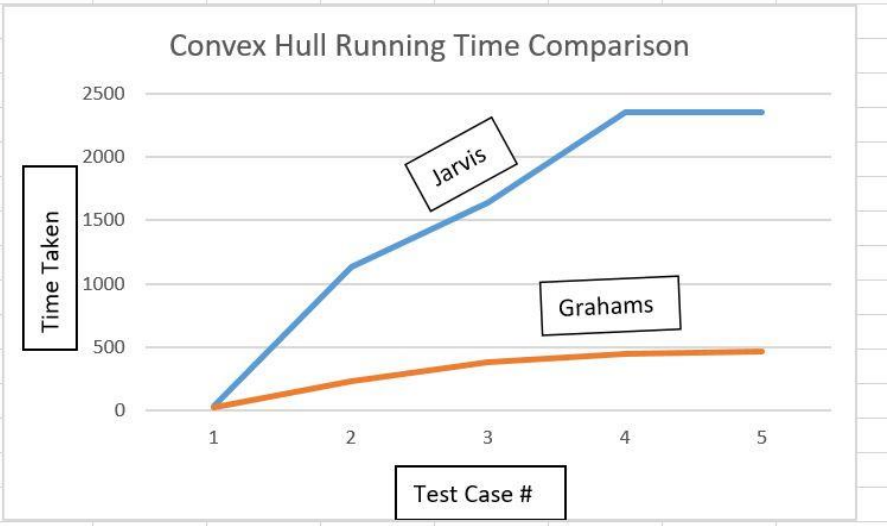
Grahams Method : O(nlogn)

In terms of space usage, assuming Grahams uses in-place sorting, both the methods use O(n) space.
For an input data set with most points in interior region, Jarvis method finds convex hull in quick time.
Similarly, if the input data set is concentrated on the boundaries Grahams performs better.
So, for the analysis, we have chosen five large sets of randomly generated points on a plane.
[Data sets were obtained from (Graphs) reference[8]]
Performance analysis of some convex hull algorithms are studied at reference[5].

Number of runs for each test case: 5

| TestCase | Input Points | Hull Size | Jarvis RunningTime(Mean) | Grahams RunningTime(Mean) |
|---|---|---|---|---|
| 1. | 12458 | 109 | 30 | 24 |
| 2. | 121275 | 961 | 1134 | 230 |
| 3. | 204587 | 788 | 1640 | 382 |
| 4. | 245000 | 990 | 2356 | 450 |
| 5. | 246708 | 983 | 2358 | 463 |

| Input Size | Jarvis | Grahams |
|---|---|---|
| 12458 | 30 | 24 |
| 121275 | 1134 | 230 |
| 204587 | 1640 | 382 |
| 245000 | 2356 | 450 |
| 246708 | 2358 | 463 |




Conclusion: For the above test case data, Grahams has outperformed Jarvis Method in terms of running time.

Improvement on Jarvis method:

Preprocessing like “throw-away pre-processing” discussed by “Akl and Toussaint Method” would largely improve the performance of Jarvis method.

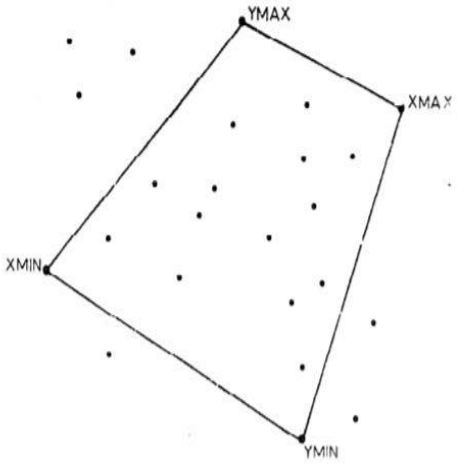



Image Reference: http://www-cgrl.cs.mcgill.ca/~godfried/publications/fast.convex.hull.algorithm.pdf

Finding a quadrilateral like above with (Xmin,Ymin,Ymax,Xmax) will reduce the size of the input set P and thus improves the performance. Points inside the quadrilateral can never be part of convex hull.

References:


1. *http://jeffe.cs.illinois.edu/teaching/373/notes/x05-convexhull.pdf*
2. *https://pdfs.semanticscholar.org/df44/e1cacaf8ef11db7c84042d1bebbd68a1d39d.pdf*
3. *https://www.cs.swarthmore.edu/~adanner/cs97/s08/pdf/ANewConvexHull.pdf*
4. *http://www-cgrl.cs.mcgill.ca/~godfried/publications/fast.convex.hull.algorithm.pdf*
5. *https://www.researchgate.net/publication/226570858_Some_performance_tests_of_convex_hull_algorithms*

6. http://www.geeksforgeeks.org/convex-hull-set-1-jarviss-algorithm-or-wrapping/
7. http://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/
8. http://www.info.univ-angers.fr/pub/porumbel/graphs/