**Greedy Vs. Tabular Knapsack**

**1. Motivation – Rationale**

*What is the importance of this problem in the world.*

Sorting is necessary for other important operations like searching. Without knowing what sorting algorithm should be used on a given program or operations, it has the potential to be extremely inefficient causing the programs and operations that require sorting to run slower / inefficiently, which makes the computers and phones run slower when running these operations and programs. This then creates a gap that without solving this problem could only solved by developing more advanced computers, but even this is a tireless effort since sorting / searching through data sets with millions of data points with an n^2 complexity instead of nlog(n) complexity would require constant technological innovation whereas solving this problem, so that we know to use the nlog(n) sorting algorithm could accomplish the same result.

*What do we want to do?*

We want to know when 0-1 knapsack greedy algorithm and tabular algorithm should be used instead of the other, in order to accomplish this we are going design and implement both of these algorithms into c++, using assert statements and invariants to ensure correctness, and test them for the worst case, average case and best case arrays of size n with a max weight of W, gathering data for the time it takes to find the highest value possible given the algorithm, the number of items n, and the maximum weight W, then compare this data with the expected time complexity. This will tell us to when to use the greedy algorithm and when to use the tabular algorithm given the parameters of an application

*Why?*

Finding the time complexity for greedy and tabular approachs will allow us to figure out when to use the appropriate sorting algorithm. Therefor allowing us to sort any given size of data more efficiently, essentially increasing the speed for anything that requires sorted data, such as searching.

**2. Background**

*What is the historical perspective of algorithms, when and why developed?*

The first ever algorithm was developed around 1700 BC by the Egyptians to multiply two numbers together (source 1). However the first sorting algorithm, Radix sort, was developed in 1901 by Herman Hollerith the founder of IBM, for the purpose of sorting the columns on the punch cards that were used by a machine he made to tally for the U.S. census (source 2). The rational for developing new algorithms is to solve new problems or solve old problems better or at least better in some aspect.

*Which algorithm is easier to understand, why.*

Greedy is easier to understand, because someone trying to understand the algorithm just needs to have a basic understanding of sorting and how the algorithm itself wroks, whereas tabular requires someone to understand a more complex algorithm that uses a 2D array and recurrsion which is still not that hard to understand, but when compared with the basic things needed for greedy algorithm, 2D array’s and recurrsion for tabular is harder to understand.

*How do they work?*

Greedy algorithm works by sorting the items in desending order based on each items value/weight, then starting from the first element in the array of sorted items, if the weight of the item <= the maximum weight then the value associated is added then this is repeated until the weight of all items excedes the remaining weight limit, except the weight of the item must be <= the remaining weight.

Tabular algorithm works by creating a 2D array with 1 dimension being the number of items and the other being the maximum weight, each index in the 2D array is filled in row by row (each item) and each index is these rows is found by seeing if that rows item fits into the columns weight limit if the item has less weight than the limit add the value to the index along with the value in the row above but in the column representing the new weight after adding the last item, keep doing this till there are no more items or no more weight left, then fill the entry with this value or the value of the index above, whichever is larger. The last entry in the 2D array is the optimal value

*How do they work what is the complexity.*

Greedy in all cases is of complexity O(nlg(n)) since the main part of this algorithm is sorting which is also of complexity O(nlg(n)) for any case even though there is slight variation given the arrays any comparisons after this are of linear complexity

Tabular in all cases is of complexity O(nW) since the 2D array must be filled out to compute the optimal knapsack value, and the array has n \* W indexs to fill

**3. Procedures**

*Overall Structure of the Program*

*Classes with two 0-1 Knapsack methods.*

class GreedyCls

{

public:

void valPerWeightSort(int vals[], int weights[], int n);

void MaxHeapify(int vals[], int weights[], int pos, int n);

void MakeMaxHeap(int vals[], int weights[], int n);

int knapGreedy(int vals[], int weights[], int n, int W);

};

class TabularCls

{

public:

int knapTabular(int vals[], int weights[], int n, int W);

};

*The Driver that uses these methods to 0-1 Knapsack the data.*

int main()

{

const int MAX\_NUM\_ELEMENTS = 1000;

GreedyCls Greed;

TabularCls Tab;

int \* valsG;

int \* weightsG;

int \* valsT;

int \* weightsT;

int temp;

int totalWeight;

srand(time(NULL));

for(int i = 0; i <= MAX\_NUM\_ELEMENTS; i++)

{

valsG = new int[i];

weightsG = new int[i];

valsT = new int[i];

weightsT = new int[i];

totalWeight = 0;

for(int j = 0; j < i; j++)

{

temp = rand() % MAX\_NUM\_ELEMENTS + 1;

valsG[j] = temp;

valsT[j] = temp;

}

for(int j = 0; j < i; j++)

{

temp = rand() % MAX\_NUM\_ELEMENTS + 1;

weightsG[j] = temp;

weightsT[j] = temp;

totalWeight += temp;

}

for( int j = 0; j <= totalWeight; j++)

{

Greed.knapGreedy(valsG, weightsG, i, j);

Tab.knapTabular(valsT, weightsT, i, j);

}

delete [] valsG;

delete [] weightsG;

delete [] valsT;

delete [] weightsT;

}

return 0;

}

*Graphical display for visualizing the layout of classes/methods.*

*Pseudocode with Correct Program Headers for usability*

int knapGreedy(int vals[], int weights[], int n, int W)

{

int Value = 0

valPerWeightSort(vals(1..n), weights(1..n), n)

reverse(vals(1..n))

reverse(weights(1..n))

for i = 1 up to n

if weights(i) <= W

Value += vals(i)

W = W – weights(i)

return Value

}

void valPerWeightSort(int vals[], int weights[], int n)

{

MakeMaxHeap(vals(1..n), weights(1..n), n)

for i = n down to 2

swap vals(1) and vals(i)

swap weights(1) and weights(i)

MaxHeapify(vals(1..n), weights(1..n), 1, i-1)

}

void MaxHeapify(int vals[], int weights[], int pos, int n)

{

index = 2\* pos

if index > n

return

if index < n

if vals(index + 1) / weights(index + 1) > vals(index) / weights(index)

index += 1

if vals(index) / weights(index) > vals(pos) / weights(pos)

swap vals(index) and vals(pos)

swap weights(index) and weights(pos)

MaxHeapify(vals(1..n), weights(1..n), index, n)

}

void MakeMaxHeap(int vals[], int weights[], int n)

{

for i = floor (n / 2) down to 1

MaxHeapify(vals(1..n), weights(1..n), i, n)

}

int knapTabular(int vals[], int weights[], int n, int W)

{

if n == 0 or W == 0

return 0

if (weights(n) > W)

return knapTabular(vals(1..n), weights(1..n), n-1, W)

else

return max( vals(n) + knapTabular(vals(1..n), weights(1..n), n-1, W – weights(n)) ,

knapTabular(vals(1..n), weights(1..n), n – 1, W) )

}

*Pre/Post Conditions, Invariants in Pseudocode major loop invariants*

PreCondition:

vals(1..n) and weights(1..n) are arrays of numbers

int knapGreedy(int vals[], int weights[], int n, int W)

{

int Value = 0

valPerWeightSort(vals(1..n), weights(1..n), n)

reverse(vals(1..n))

reverse(weights(1..n))

for i = 1 up to n

Invariant: sum(vals(1..i-1)) = Value or W – sum(weights(1..i-1)) <= 0

if weights(i) <= W

Value += vals(i)

W = W – weights(i)

Invariant: sum(vals(1..i)) = Value or W – sum(weights(1..i)) <= 0

return Value

}

PostCondtion:

Value >= sum(vals(1..k)) for k = the index at which W – weights(1..k) < 0

PreCondition:

vals(1..n) and weights(1..n) are arrays of numbers

void valPerWeightSort(int vals[], int weights[], int n)

{

MakeMaxHeap(vals(1..n), weights(1..n), n)

for i = n down to 2

Invariant: vals(1..i) and weights(1..i) is max heap by val / weight, vals(i+1..n) and weights(i+1..n) is

sorted by val / weight

swap vals(1) and vals(i)

swap weights(1) and weights(i)

MaxHeapify(vals(1..n), weights(1..n), 1, i-1)

Invariant: vals(1..i-1) and weights(1..i-1) is max heap by val / weight, vals(i..n) and weights(i..n) is

sorted by val / weight

}

PostCondition:

vals(1..n) and weights(1..n) are sorted by val / weight

PreCondition:

vals(pos+1…n) and weights(pos+1 …n) have the max heap property by val / weight

void MaxHeapify(int vals[], int weights[], int pos, int n)

{

Invariant: vals(pos+1…n) and weights(pos+1…n) have the max heap property by val / weight

index = 2\* pos

if index > n

return

if index < n

if vals(index + 1) / weights(index + 1) > vals(index) / weights(index)

index += 1

if vals(index) / weights(index) > vals(pos) / weights(pos)

swap vals(index) and vals(pos)

swap weights(index) and weights(pos)

MaxHeapify(vals(1..n), weights(1..n), index, n)

Invariant: vals(pos…n) and weights(pos…n) have the max heap property by val / weight

}

PostCondition:

vals(pos…n) and weights(pos…n) have the max heap property by val / weight

PreCondition:

vals(1..n) and weights(1..n) are arrays of numbers

void MakeMaxHeap(int vals[], int weights[], int n)

{

for i = floor (n / 2) down to 1

Invariant: vals(i+1…n) and weights(i+1…n) have the max heap property by val / weight

MaxHeapify(vals(1..n), weights(1..n), i, n)

Invariant: vals(i…n) and weights(i…n) have the max heap property by val / weight

}

PostCondition:

vals(1…n) and weights(1…n) have the max heap property by val / weight

PreCondition:

vals(1..n) and weights(1..n) are arrays of numbers

int knapTabular(int vals[], int weights[], int n, int W)

{

Tablet[1..n+1][1..w+1]

for i = 1 to n+1

for j = 1 to w + 1

Invariant: Tablet[1..i-1][1..j-1] = 0 or = Tablet[i-2][j-1] or = vals(i-2) + Tablet[i-2][j-1-weights(i-2)]

if i == 1 or j == 1

Tablet[i][j] = 0

if (weights(i-1) > W)

Tablet[i][j] = Tablet[i-1][j];

else

Tablet[i][j] = max( vals(i-1) + Tablet[i-1][j – weights(i-1)], Tablet[i-1][j])

Invariant: Tablet[1..i][1..j] = 0 or = Tablet[i-1][j] or = vals(i-1) + Tablet[i-1][j-weights(i-1)]

return Tablet[n+1][w+1]

}

PostCondition:

Tablet[n][w] = Tablet[n][w]

*Invariants will be implemented in the program.*

*Specify how the invariants will be implemented using C++ assert statements.*

There are 4 types of invariants in the pseudocode

1. That some part of two arrays is sorted by value / weight

bool isSorted(vals(1..n), weights(1..n),start, stop)

this function will itterate from start to stop checking to make sure that the next element in vals / an element in weights at the same index is >= the previous, if this is ever false the function returns false

assert(isSorted(vals(1..n), weights(1..n),start, stop))

2. That some part of an array has the max heap property

bool isMaxHeap(vals(1..n), weights(1..n),start, stop)

this function will itterate from start to stop checking that if an index has child nodes that those child nodes that have a value aquired by taking vals(i) / weights(i) with i being the index of the child node are <= the node being indexed, if this is ever false the function returns false

assert(isMaxHeap(vals(1..n), weights(1..n),start, stop))

3. That an array is made of numbers

This does not need an assert statement since a c++ program will not compile if the functions that are expecting an array of integers get anything that is not promotable to an array of integers

4. That a value is equal to a sum of values

int sumTotal(vals(1..n), stop)

this function returns the sum of values from vals from the index 1 to stop

int finalIndex(weights(1..n),n,W)

this function returns the index,i, for which W – sum(weights(1..i)) < 0

or i = n if W – sum(weights(1..n)) >= 0

assert( Value == sumTotal(vals(1..n), finalIndex(weights(1..n),n,W)))

5. That a value is equal to another value

assert( Value == Value2)

**4. Correctness of Program**

For this part of the assignment see the 3 submitted files

Proj2.cpp – The entire expected program with comments except the assert statements are commented out for accurate time calculations

Proj2Assert.cpp – The entire expected program with comments and assert statements

finalData.txt – Sample output which contains the data used to create the graphs

**5. Implementation of Invariants assert(bool) from C++. Good Program Structure**

*List of Problems encountered*

array indexing, I designed my driver to where it would test and gather around 10010 data points, testing n from 0 to 1000 and w from 0 to 1000 steping every 100, since 1001 \* 1001 is about 1000000 data points it would be slow, but my solution of stepping constantly made my program index out of an array so I had to consistently account for this

Assert( ) / Pre / Post / Invariants : I could not figure out the invariants / assert statements for either knapsack algorithm since neither of them have to do something with an item that it is given, so the only option for invariants was something obviouslly true or if statements, for which I opted for the former

*Performance*

*Find c, n*

Greedy = O(nlg(n)), and Tabular O(nW)

so this question does not apply if both n and W are varying to any number, but

if W<40 then Tabular will be faster for all n >= k, k is an int 1-40

there are lot of ways to answers this question though

*Conclusion*

After implementing and working with both algorithms / methods of solving knapsack, greedy and tabular, it is clear that if we wish to implement the faster algorithm then the greedy algorithm is faster for almost all n, and since its comparisons depend very little on the size W, then if we are looking to implement knapsack in an environment with changing number of items, n, and changing amount of weight, W, or a high W then greedy is far and above the faster algorithm. However, if the optimal value is required or heavily prefered for the implementation of knapsack then the tabular approach is the way to go especially if W or n is fixed or low, but it is worth saying that Greedy often does not get the optimal often, but is close to optimal in all cases.

*References*

“Write a Program to Reverse an Array or String.” *GeeksforGeeks*, 8 Sept. 2020, www.geeksforgeeks.org/write-a-program-to-reverse-an-array-or-string/.

“0-1 Knapsack Problem | DP-10.” *GeeksforGeeks*, 25 Mar. 2021,

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/.

**6. Testing Plan**

*Describe what kind of data will be tested?*

The number of comparisons needed to complete 0-1 knapsack and the time taken to complete 0-1 knapsack, both tabular and greedy approachs will have their own data.

*Is it real data or manufactured synthetic data?*

It is a manufactured synthetic test since the data is not real and has no meaning since it is randomly generated, but the data will be a good representation of how real data will work.

*boundary cases what happens if n=0 or n> maximum size?*

The driver for this test is set to test both approachs with value and weight arrays ranging in size form 0 to 1000, and can be set higher with a constant and past this limit no further testing will be done, the same applies for W, or the weight alloted, which is also tested from 0 to 1000 except it steps 100 every iteration.

n=0 and W = 0 will be tested and will not create seg faults or crash when input into the algorithms.

*best case, worst case, average case data – random*

For Tabular, since it is a brute force type algorithm where every entry is filled there is no best, worst, or average case, all uses of this algorithm are O(nW). For Greedy approach, there is a best, worst, and average case since it is dependent on sorting but I have choosen to ignore it since I have implemented heap sort for sorting, which as seen in the Project 1, has very little distinction between its best, worst, and average case

**Sources:**

*Source 1*

[Timeline of algorithms - Wikipedia](https://en.wikipedia.org/wiki/Timeline_of_algorithms)

*Source 2*

[Byte Sized Episode 3: The First Ever Sorting Algorithm - DEV Community](https://dev.to/bytesized/byte-sized-episode-3-the-first-ever-sorting-algorithm-55kh)

*Source 3: reverse function*

[Write a program to reverse an array or string - GeeksforGeeks](https://www.geeksforgeeks.org/write-a-program-to-reverse-an-array-or-string/)

*Source 4: knapSack dynamic/ tabular*

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/