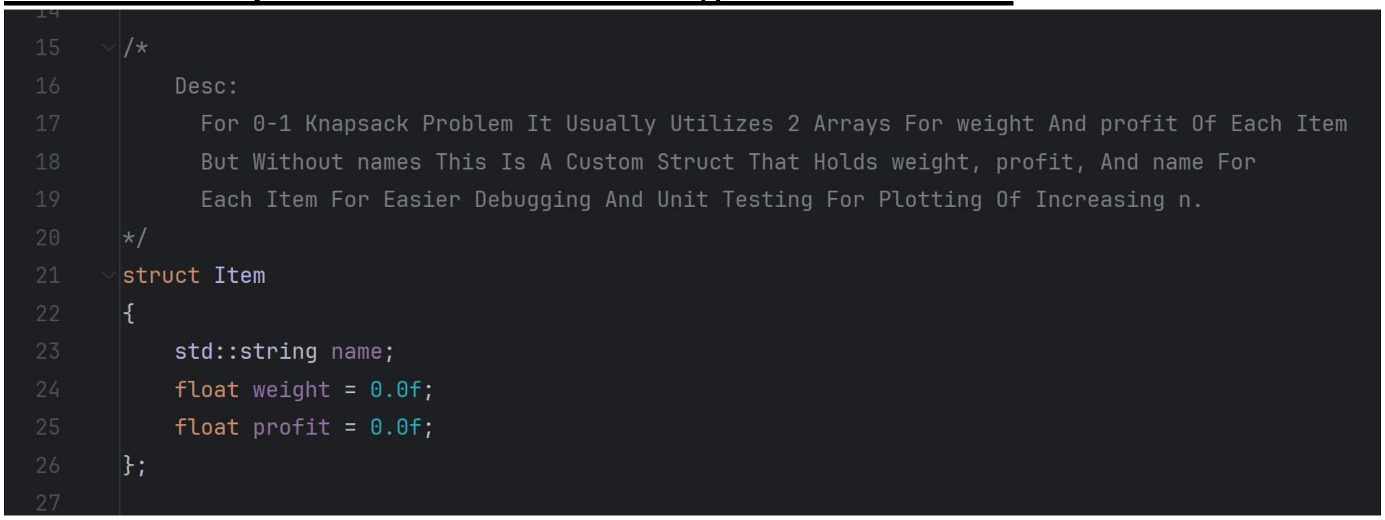
0-1 KnapSack Algorithmic & Heuristic Analysis

By: Kingerthanu

1. Background

1.1 All implementations are utilizing the Item Struct:



```
/*
    Desc:
      For 0-1 Knapsack Problem It Usually Utilizes 2 Arrays For weight And profit Of Each Item
      But Without names This Is A Custom Struct That Holds weight, profit, And name For
      Each Item For Easier Debugging And Unit Testing For Plotting Of Increasing n.
*/
struct Item
{
    std::string name;
    float weight = 0.0f;
    float profit = 0.0f;
};
```

1.2 Hardware:

Benchmarking was all done on a specific desktop so times could be varying system-to-system, but overall trends will be the same:

CPU:
Intel® Core™ i9-12900K Processor
30M Cache, up to 5.20 GHz

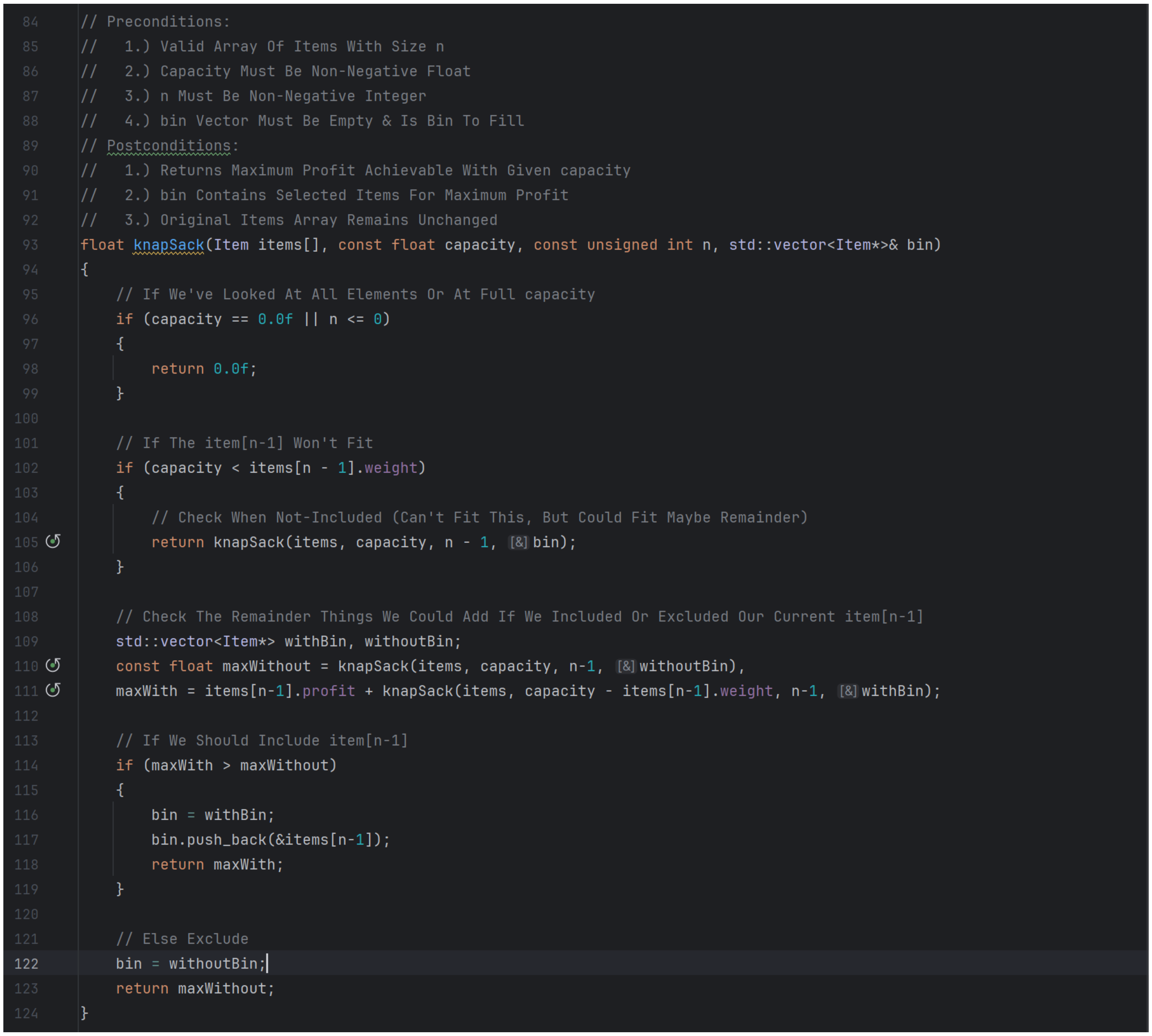
Memory:
G.SKILL Trident Z5 RGB Series (Intel XMP 3.0) DDR5 RAM
64GB overall

1.3 Key Terms:

- n = number of Items being considered to place in the knapsack

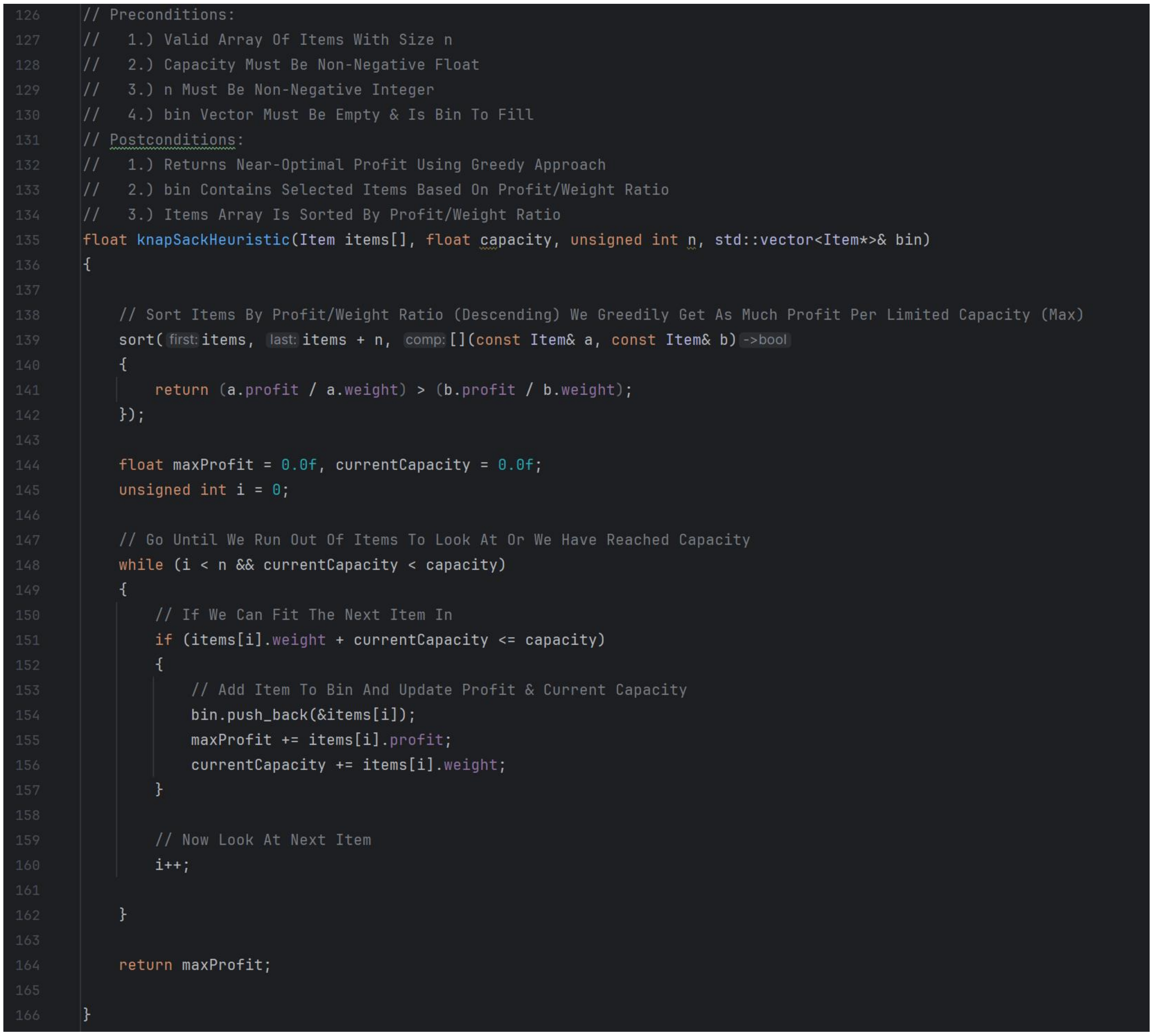
2. Implementations (C++)

2.1 *Exponential Algorithm:*



```cpp
// Preconditions:
//   1.) Valid Array Of Items With Size n
//   2.) Capacity Must Be Non-Negative Float
//   3.) n Must Be Non-Negative Integer
//   4.) bin Vector Must Be Empty & Is Bin To Fill
// Postconditions:
//   1.) Returns Maximum Profit Achievable With Given capacity
//   2.) bin Contains Selected Items For Maximum Profit
//   3.) Original Items Array Remains Unchanged
float knapSack(Item items[], const float capacity, const unsigned int n, std::vector<Item*>& bin)
{
    // If We've Looked At All Elements Or At Full capacity
    if (capacity == 0.0f || n <= 0)
    {
        return 0.0f;
    }

    // If The item[n-1] Won't Fit
    if (capacity < items[n - 1].weight)
    {
        // Check When Not-Included (Can't Fit This, But Could Fit Maybe Remainder)
        return knapSack(items, capacity, n - 1, [&] bin);
    }

    // Check The Remainder Things We Could Add If We Included Or Excluded Our Current item[n-1]
    std::vector<Item*> withBin, withoutBin;
    const float maxWithout = knapSack(items, capacity, n-1, [&] withoutBin),
    maxWith = items[n-1].profit + knapSack(items, capacity - items[n-1].weight, n-1, [&] withBin);

    // If We Should Include item[n-1]
    if (maxWith > maxWithout)
    {
        bin = withBin;
        bin.push_back(&items[n-1]);
        return maxWith;
    }

    // Else Exclude
    bin = withoutBin;
    return maxWithout;
}
```

2.2 *Standard Greedy Heuristic:*

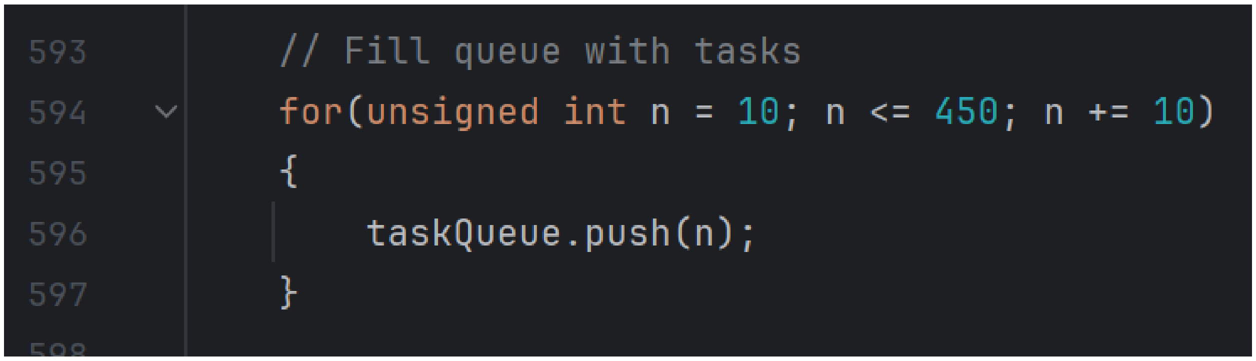


```cpp
// Preconditions:
//   1.) Valid Array Of Items With Size n
//   2.) Capacity Must Be Non-Negative Float
//   3.) n Must Be Non-Negative Integer
//   4.) bin Vector Must Be Empty & Is Bin To Fill
// Postconditions:
//   1.) Returns Near-Optimal Profit Using Greedy Approach
//   2.) bin Contains Selected Items Based On Profit/Weight Ratio
//   3.) Items Array Is Sorted By Profit/Weight Ratio
float knapSackHeuristic(Item items[], float capacity, unsigned int n, std::vector<Item*>& bin)
{

    // Sort Items By Profit/Weight Ratio (Descending) We Greedily Get As Much Profit Per Limited Capacity (Max)
    sort( first: items,  last: items + n,  comp: [](const Item& a, const Item& b) ->bool
    {
        return (a.profit / a.weight) > (b.profit / b.weight);
    });

    float maxProfit = 0.0f, currentCapacity = 0.0f;
    unsigned int i = 0;

    // Go Until We Run Out Of Items To Look At Or We Have Reached Capacity
    while (i < n && currentCapacity < capacity)
    {
        // If We Can Fit The Next Item In
        if (items[i].weight + currentCapacity <= capacity)
        {
            // Add Item To Bin And Update Profit & Current Capacity
            bin.push_back(&items[i]);
            maxProfit += items[i].profit;
            currentCapacity += items[i].weight;
        }

        // Now Look At Next Item
        i++;

    }

    return maxProfit;

}
```

I.) I utilized the "Standard Greedy" Heuristic as the study provided states how this algorithm works as a great middle-ground heuristic for accuracy (~35% error) and runtime efficiency having "above-average values and below-average times." Likewise, it has a very intuitive design and readability as a heuristic compared to others mentioned in write up (others were either impractical for large n like Max of All or Sliding Threshold).

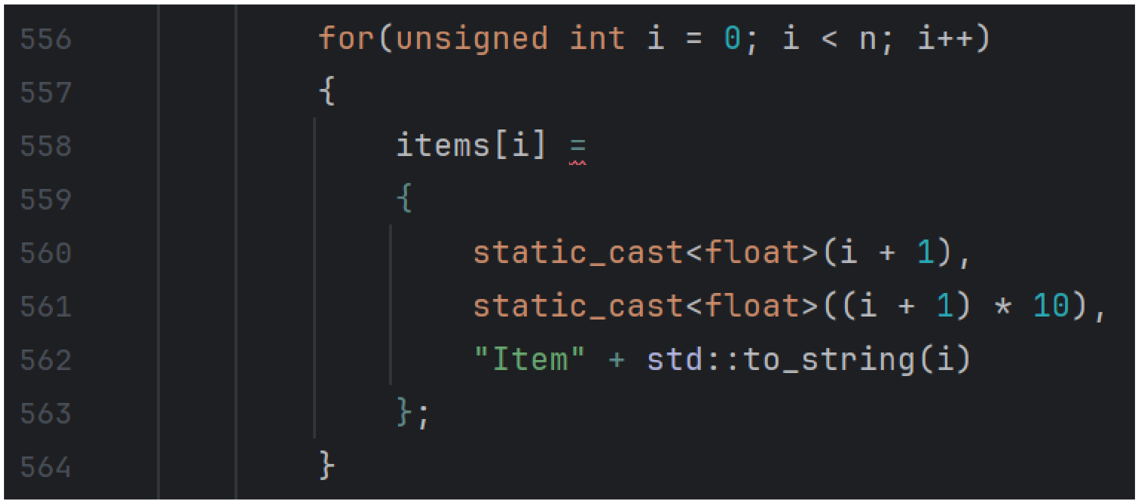
3. Datasets

- We have 45 individual data sets we are utilizing to benchmark our two implementations' runtimes of the 0-1 Knapsack problem. We utilize 45 individual values of n to give a granular and scalable analysis of our implementations for both.
    - We go from ***n** = **10*** up to ***n** = **450*** in increments of 10:
        - **for(unsigned int n = 10; n <= 450; n += 10)**



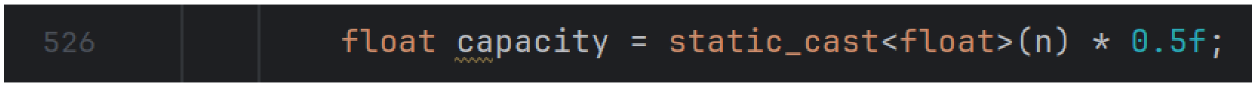
```
    // Fill queue with tasks
    for(unsigned int n = 10; n <= 450; n += 10)
    {
        taskQueue.push(n);
    }
```

- Each data set is constructed using a loop to populate arrays of Item structs, simulating Items with increasing weights and profits; for each, we really don't care that much about the specific values of weight and profit as our exhaustive algorithm will always be checking 2^n orderings and the heuristic will always no matter what sort but could have a varying linear scan of the array after sorting depending on how many items can fit in the bin but this is negligible to our analysis:
    - **(weight = i + 1, profit = (i + 1) * 10)**



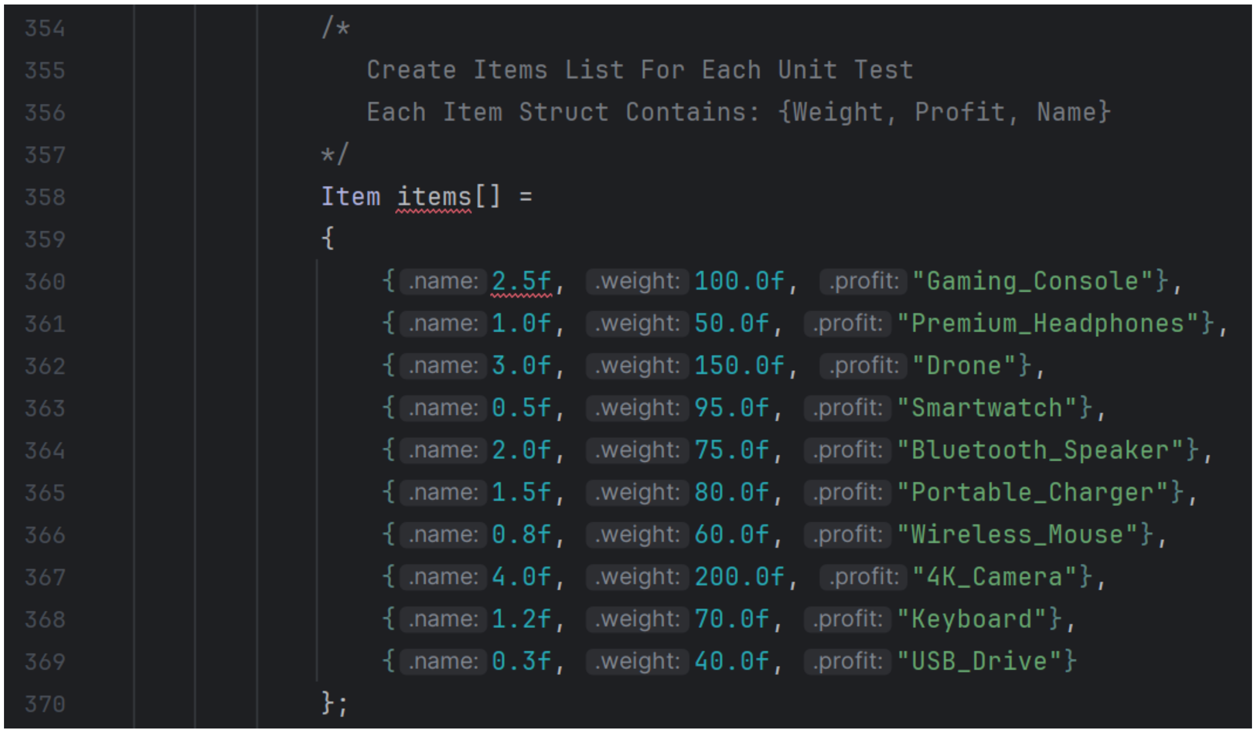
```
        for(unsigned int i = 0; i < n; i++)
        {
            items[i] =
            {
                static_cast<float>(i + 1),
                static_cast<float>((i + 1) * 10),
                "Item" + std::to_string(i)
            };
        }
```

- Capacity also scales with the larger data sets of n:



```
float capacity = static_cast<float>(n) * 0.5f;
```

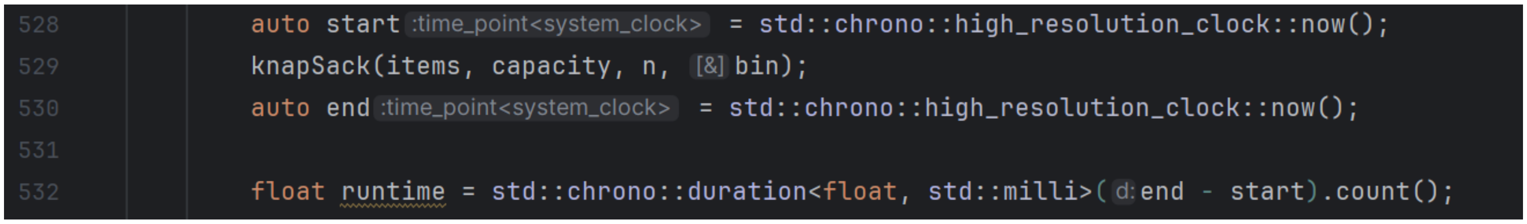
- **We are utilizing unique data sets with known solutions in our Unit Test functions to ensure proper functionality and not as much runtime performance as the benchmarking functions…**



```
/*
   Create Items List For Each Unit Test
   Each Item Struct Contains: {Weight, Profit, Name}
*/
Item items[] =
{
    { .name: 2.5f, .weight: 100.0f, .profit: "Gaming_Console"},
    { .name: 1.0f, .weight: 50.0f, .profit: "Premium_Headphones"},
    { .name: 3.0f, .weight: 150.0f, .profit: "Drone"},
    { .name: 0.5f, .weight: 95.0f, .profit: "Smartwatch"},
    { .name: 2.0f, .weight: 75.0f, .profit: "Bluetooth_Speaker"},
    { .name: 1.5f, .weight: 80.0f, .profit: "Portable_Charger"},
    { .name: 0.8f, .weight: 60.0f, .profit: "Wireless_Mouse"},
    { .name: 4.0f, .weight: 200.0f, .profit: "4K_Camera"},
    { .name: 1.2f, .weight: 70.0f, .profit: "Keyboard"},
    { .name: 0.3f, .weight: 40.0f, .profit: "USB_Drive"}
};
```

4. Results

- Utilizing std::chrono we are getting the millisecond runtime of each of our data set entries both for heuristic & for our exponential implementations.



```
    auto start :time_point<system_clock> = std::chrono::high_resolution_clock::now();
    knapSack(items, capacity, n, [&] bin);
    auto end :time_point<system_clock> = std::chrono::high_resolution_clock::now();

    float runtime = std::chrono::duration<float, std::milli>( d: end - start).count();
```

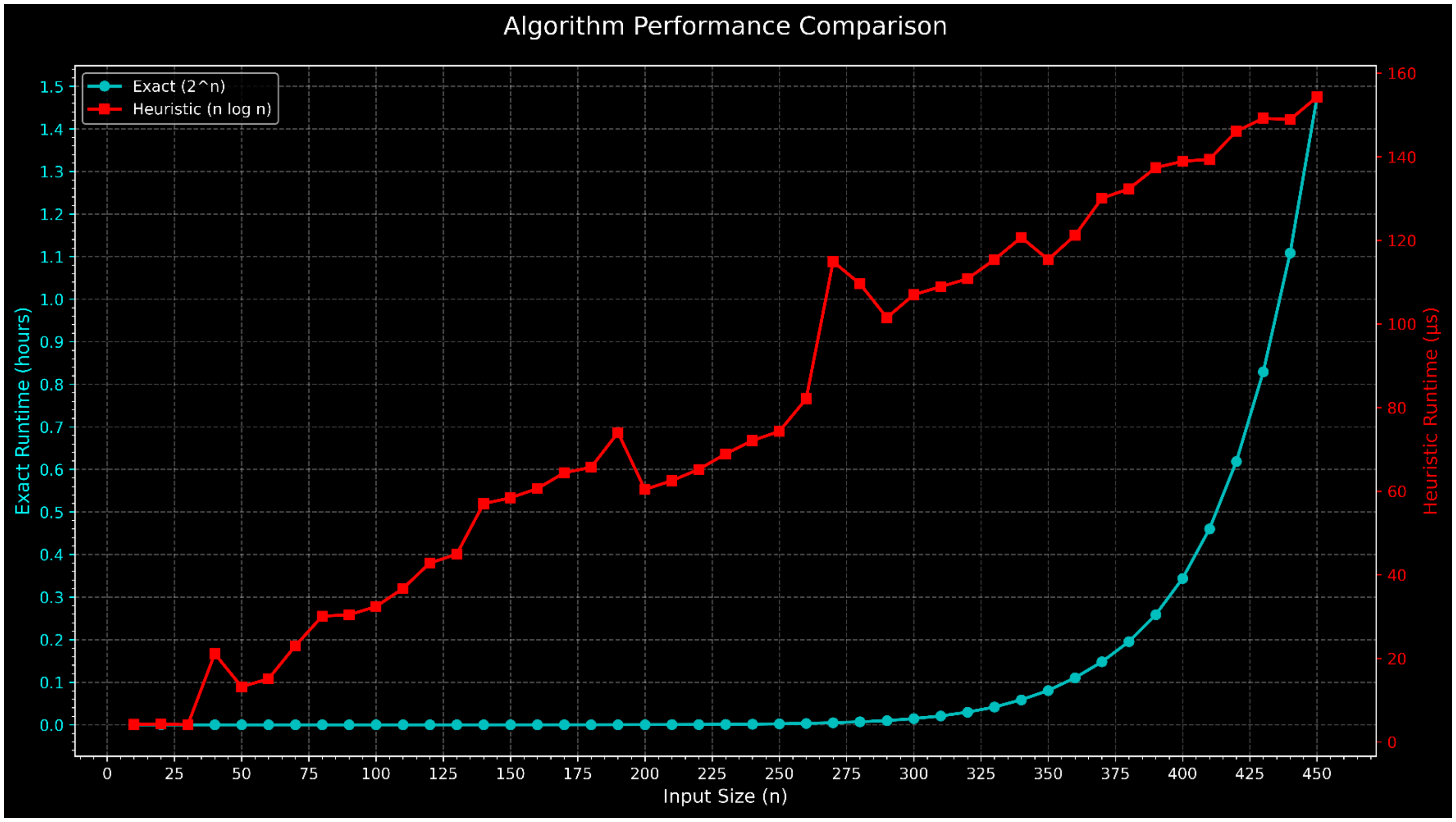
    - Milliseconds allow good granularity for plotting both implementations.

Benchmark Data:

Input Size (n)	Exact Algorithm (ms)	Heuristic Algorithm (ms)
10	0.0028	0.0042
20	0.0147	0.0043
30	0.0353	0.0041
40	0.0614	0.0211
50	0.2371	0.0132
60	0.5172	0.0151
70	1.0001	0.023
80	1.6946	0.0301
90	4.3365	0.0304
100	7.3182	0.0324
110	11.9984	0.0367
120	24.4226	0.0428
130	42.0059	0.0449
140	57.0581	0.057
150	120.128	0.0584
160	168.164	0.0606
170	247.572	0.0644
180	401.229	0.0657
190	662.441	0.074
200	1048.28	0.0604
210	1529.19	0.0625
220	2341.3	0.0652
230	3739.25	0.0689
240	5233.63	0.0721
250	7976.04	0.0743
260	11856.6	0.0821
270	17280.5	0.1149
280	25048.4	0.1096
290	35431.5	0.1015
300	52194.8	0.107
310	74622.9	0.1089
320	106957	0.1108
330	149898	0.1154
340	211455	0.1207
350	290075	0.1154
360	397756	0.1212
370	532472	0.1301
380	703095	0.1323
390	931123	0.1374

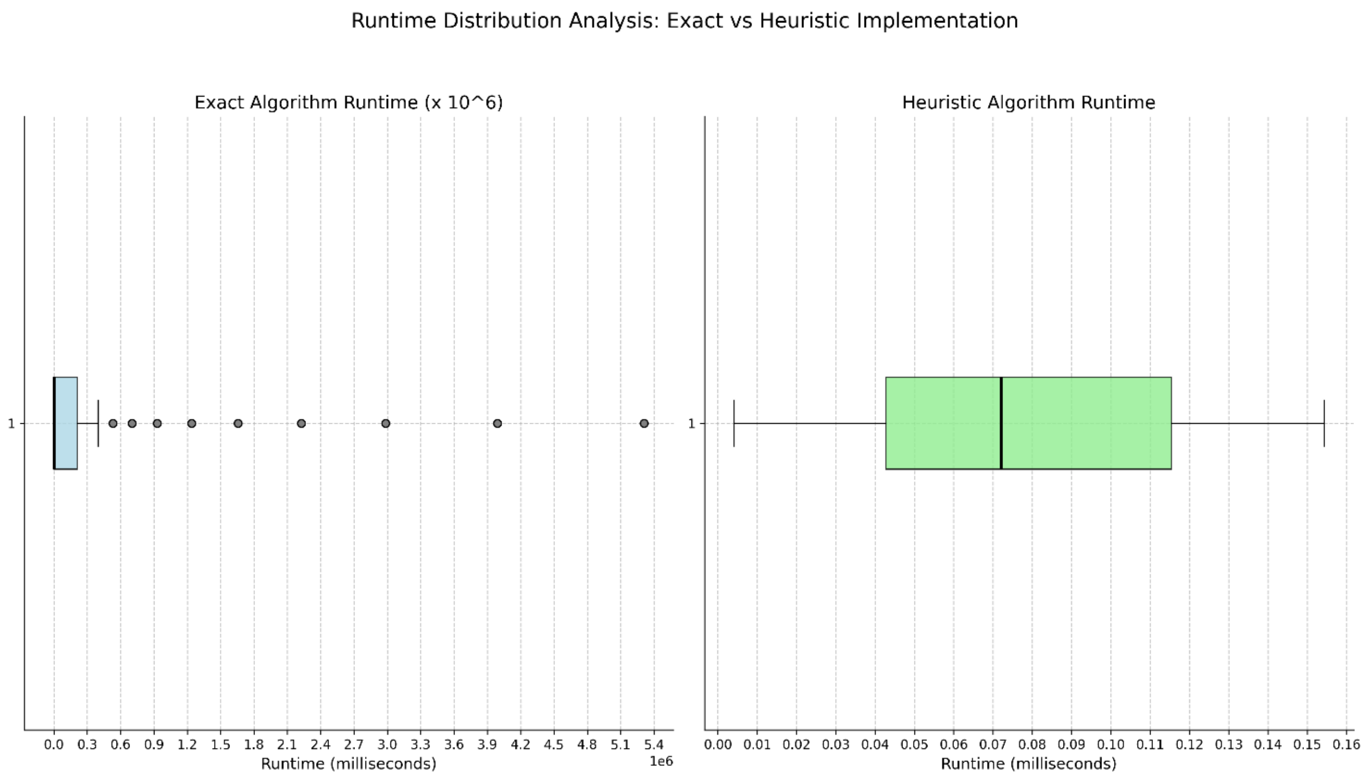
400	1238500	0.1389
410	1657390	0.1394
420	2227490	0.1461
430	2985750	0.1492
440	3991300	0.1489
450	5309700	0.1543

Time complexity differences between heuristic (Standard Greedy) vs Optimal Inclusion-Exclusion Exhaustive Search:

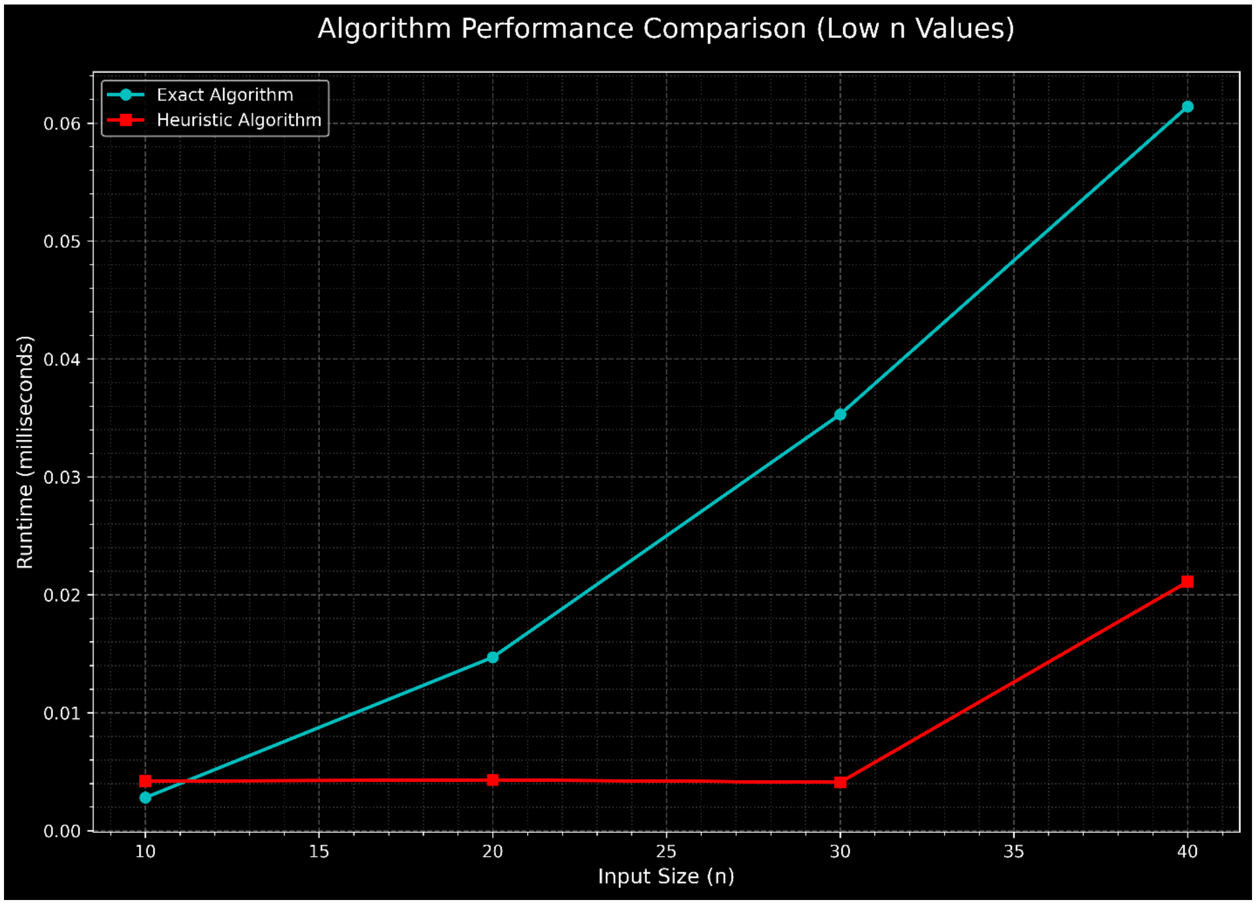


Both implementations when utilizing the same ever increasing sized data sets show starkly different means of rates of change. The line in the blue is charted with a y-axis in the time units of hours (shown on the left), while the line in the red is charted with the y-axis in the time units of microseconds (shown on the right)—both share the same x-axis units.

The heuristic, suboptimal implementation ran in microsecond runtimes for the same data set showing its triumph in runtime complexity compared to our exact optimal exhaustive algorithm as even though for small n the runtime barely changes as shown in the blue line, as it increases, we see that exponential curve shooting up to ~1.5 hours. This is different than the heuristic solution in which doesn't have an exponential curve in runtime and instead is much more linear (logarithmic pseudo-linear) (albeit with peaks and divots and peaks at a value of ~150 microseconds).



From plotting our results into a whisker plot we can also get a good idea on how the data trends amongst its individual entries. With our heuristic, we see how the runtime spread is very reasonable (backing its generally pseudo-linear nature) as we don't have these extrema points as we see in our exact algorithm. With the exact algorithm, we see a much less clean spread of our runtime with it having many extremes not covered in our box. The plotting of this exact solution also shows just how aggressively our runtime inflates as n gets bigger.



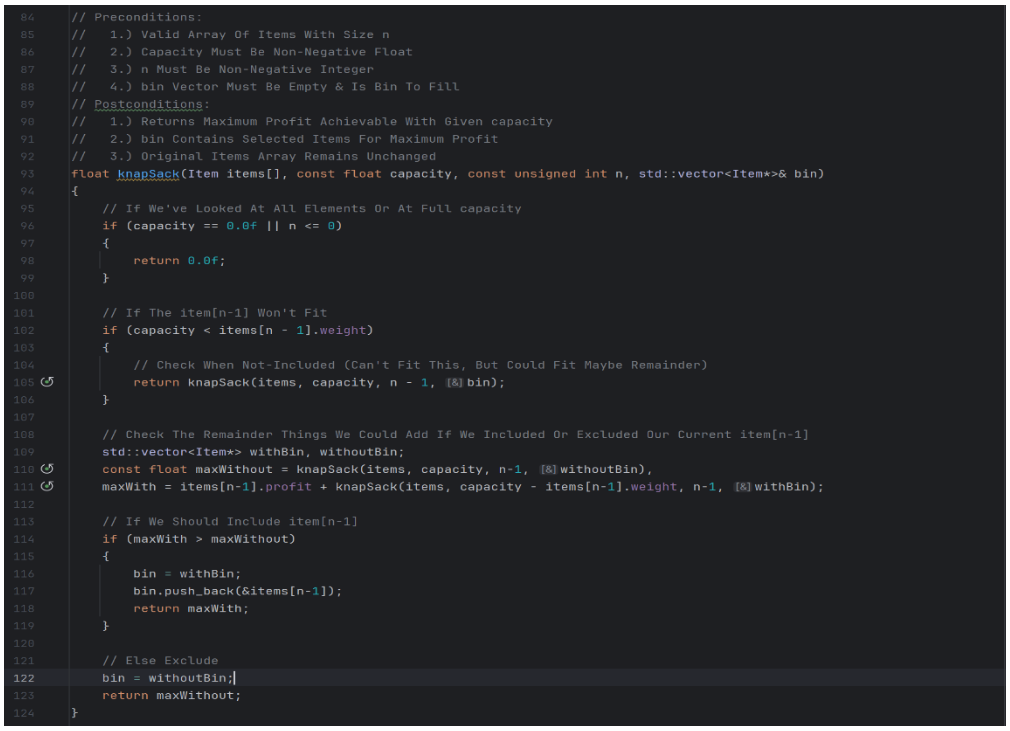
When shrinking down our dataset being plotted to only the first 4 entries, we see for our first case though that we actually have better runtime (albeit minor) showing how for quite low input sizes we could actually see benefits as we wouldn't be incurring the sorting which could be more expensive than a simple 2^n with a small n—of ~10 items which is quite impractical…

5. Implementation Analysis

The knapsack problem orientates around finding the optimal way to pack items of specific size and profit into a container of a specific size to maximize the total profit. The bin assumes optimal packing as well so don't need to worry about insertion order of items. The problem is complex (NP-complete) as items do not have any ordering to help with packing so we could have one item that is size **10** and profit of **100**, being inserted into a knapsack of size **10**; if all other items profit were below **100**, we may infer that adding the one item of profit **100** would be most optimal for the highest profit out of the item list. But if we have **2** other items of size **5** and profit **90** each, adding these two items would actually give us a higher profit of **180**. We wouldn't know this (same with our computer) though, unless we exhaustively check all subsets we could make as if we terminate before we look at these **2** other items we may prematurely stop before seeing their profit being higher.

When checking the optimal packing for maximum profit, for each item we must ask the question of do we include this item or exclude it. This simple question brings deep nuance though, as enforcing this in code can be quite troublesome from a code complexity view as well as a runtime complexity view. Means of tabular bottom-up dynamic-programming are available to be enforced (at the expense of code comprehension) to provide better runtime efficiency from a lack of stack cleanup from recursive means, but even this is pseudo polynomial and realistically isn't as comprehensive as the more exhaustive algorithm from the sphere of code complexity (and even runtime if our W is bigger than our n in the **O(n*W)** complexity of this DP) which can help in the analysis of the upper-bound this problem can have. While means of heuristic can run much faster, this is because it's a suboptimal answer and unlike the exhaustive—or DP implementation—which are always the best packing, we could frequently get good packings of items with the heuristic but not the best in leu of us being able to get a solution much faster with the heuristic.

Exponential Algorithm:



```
// Preconditions:
//   1.) Valid Array Of Items With Size n
//   2.) Capacity Must Be Non-Negative Float
//   3.) n Must Be Non-Negative Integer
//   4.) bin Vector Must Be Empty & Is Bin To Fill
// Postconditions:
//   1.) Returns Maximum Profit Achievable With Given capacity
//   2.) bin Contains Selected Items For Maximum Profit
//   3.) Original Items Array Remains Unchanged
float knapSack(Item items[], const float capacity, const unsigned int n, std::vector<Item*>& bin)
{
    // If We've Looked At All Elements Or At Full capacity
    if (capacity == 0.0f || n <= 0)
    {
        return 0.0f;
    }

    // If The item[n-1] Won't Fit
    if (capacity < items[n - 1].weight)
    {
        // Check When Not-Included (Can't Fit This, But Could Fit Maybe Remainder)
        return knapSack(items, capacity, n - 1, [&] bin);
    }

    // Check The Remainder Things We Could Add If We Included Or Excluded Our Current item[n-1]
    std::vector<Item*> withBin, withoutBin;
    const float maxWithout = knapSack(items, capacity, n-1, [&] withoutBin),
    maxWith = items[n-1].profit + knapSack(items, capacity - items[n-1].weight, n-1, [&] withBin);

    // If We Should Include item[n-1]
    if (maxWith > maxWithout)
    {
        bin = withBin;
        bin.push_back(&items[n-1]);
        return maxWith;
    }

    // Else Exclude
    bin = withoutBin;
    return maxWithout;
}
```

Algorithm Complexity:

In the recursive exhaustive algorithm for the 0-1 knapsack, it works quite straight forward. We are provided 4 arguments:

1. The Items to be considered for the knapsack
2. The current capacity of the knapsack
3. The amount of items to be considered
4. The current knapsack (more optional as not specifically required by problem)

We start by looking at the last item we have in our items to consider list (n-1) and looking if this item can possibly be added into the current knapsack, this is done by checking if either our knapsack's current status is either fully filled (in which case we can stop) or our item isn't able to fit in the current knapsack; but we cannot stop early here as just because this item doesn't fit doesn't mean all the items don't fit as we have no assumptions with the ordering of our item list so instead of stopping we look at what the knapsack would look like if we exclude this item and continue packing the other items preceding it, this is done through a recursive call back to our function with the same arguments but instead removing this item that didn't fit (n-1). This overall structure is to ensure we create all subset combinations of our items in the knapsack. Because we are using a recursive structure as well, we must do bounds check on our main recursion variable (n <= 0). These base cases incur ~ **3* O(1)** in overhead, but are negligible.

After this though, we then start our main section of the 0-1 knapsack which is our inclusion-exclusion. For this, we simply recursively call our function twice—one including the current item and the second excluding the current item. Because we are packing based on the best profit, with our call that includes our item we will recursively call our function with the capacity minus the current items weight we are including symbolizing the current available space after the packing. We will also include the profit gained from this given item as now that its in our knapsack our knapsack's total profit will increase. Our exclusion of our item will simply be calling back our function with the capacity without misusing the current items weight and without the addition of our profit as excluding the item from our knapsack will mean we don't gain any profit or change in current capacity. After the calling of both inclusion, and exclusion of the specific item, we look at which provides us a better overall profit this is then saved away and returned. For our specific implementation as well we integrate 2 containers to hold what items are put in the knapsack for easier display of functionality as when leaving the recursive chain each of these will help us remember the items being added and not just the maximum profit.

After this first item is considered for inclusion-exclusion, we will have 2 offshoot recursive calls for the given including of the item (with a reduced capacity and addition of the item's profit) and the exclusion of the item (with no reduction in capacity and no addition of the item's profit). This offshoot will then happen again for each knapsack in construction for our subset combinations as for the next item considered both inclusion and exclusion must be considered as our computer doesn't "know" if this item is a part of the optimal solution; this will then happen again for all items greatly inflating our overall computations being done. This is why this problem is seen as NP-complete as the inherent problem of including or excluding a provided item causes a lot of computational overhead.

Because we are including-excluding each item based on the current knapsack being constructed we will be incurring an **O(2^n)** runtime complexity.

Code Complexity:

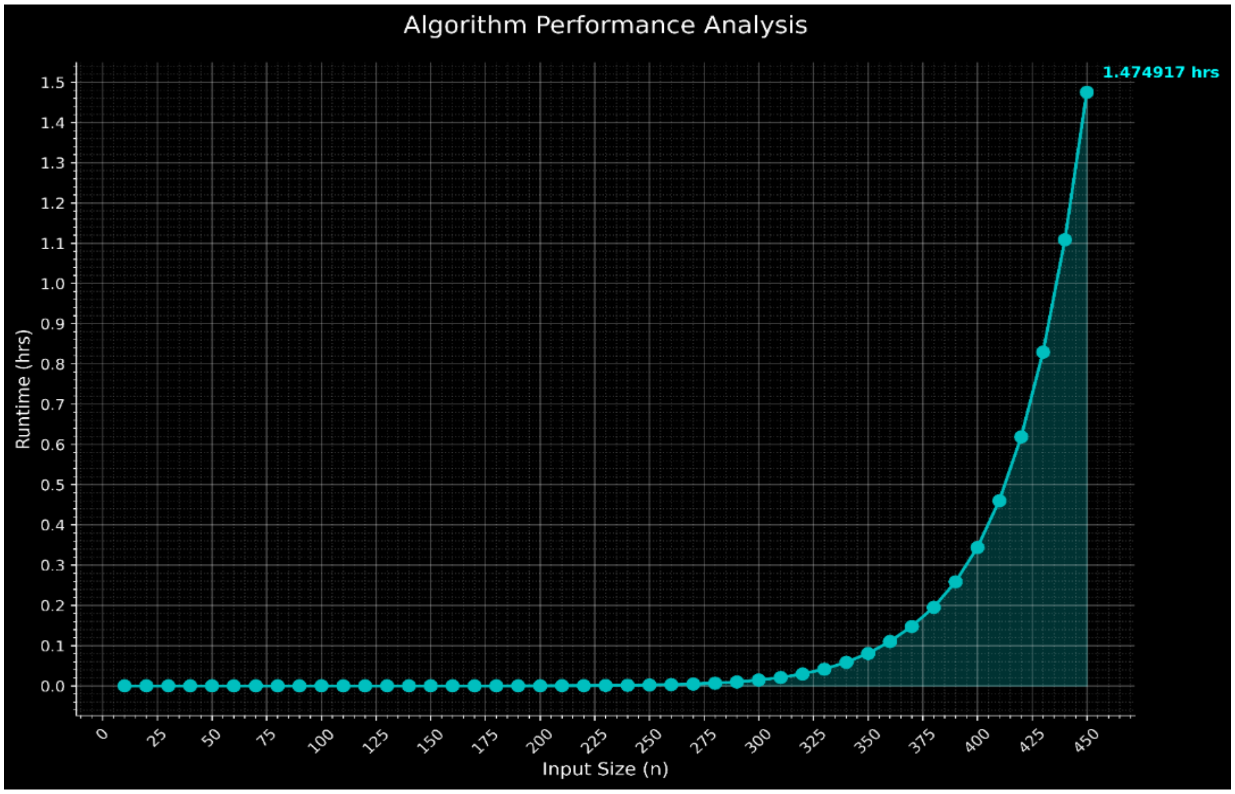
The code complexity of the provided recursive exhaustive algorithm is quite minor if you have a good understanding of top-down recursion. This mainly has to do with the problem overall being quite binary in nature as we must include an item from the knapsack or exclude it, this maps to one recursive call including and one excluding. The code structure of this recursion is quite easily digestible as knowing it's a maximization problem of the profit it's easily understandable that the inclusion case will have a recursive call adding the profit (and reducing the capacity of the knapsack) and the exclusion recursive call will have no addition of profit or reduction of capacity. The base cases are also very intuitive as if the weight of the current item is above capacity, we must exclude this current item naturally as it just won't fit. The other cases of if we are out of items to look at or the capacity is 0 and us returning a profit of 0 (as we there's no more money to be added with no more room to add items or no more items to add) also makes inherent sense when discussing adding items to a knapsack. The addition of the bin instances may even be a little confusing for people knowing recursion though, as we are making multiple copies of what our bin with or without items looks like and bringing up the recursive chain the optimal one which can be hard to grasp as the list grows incrementally.

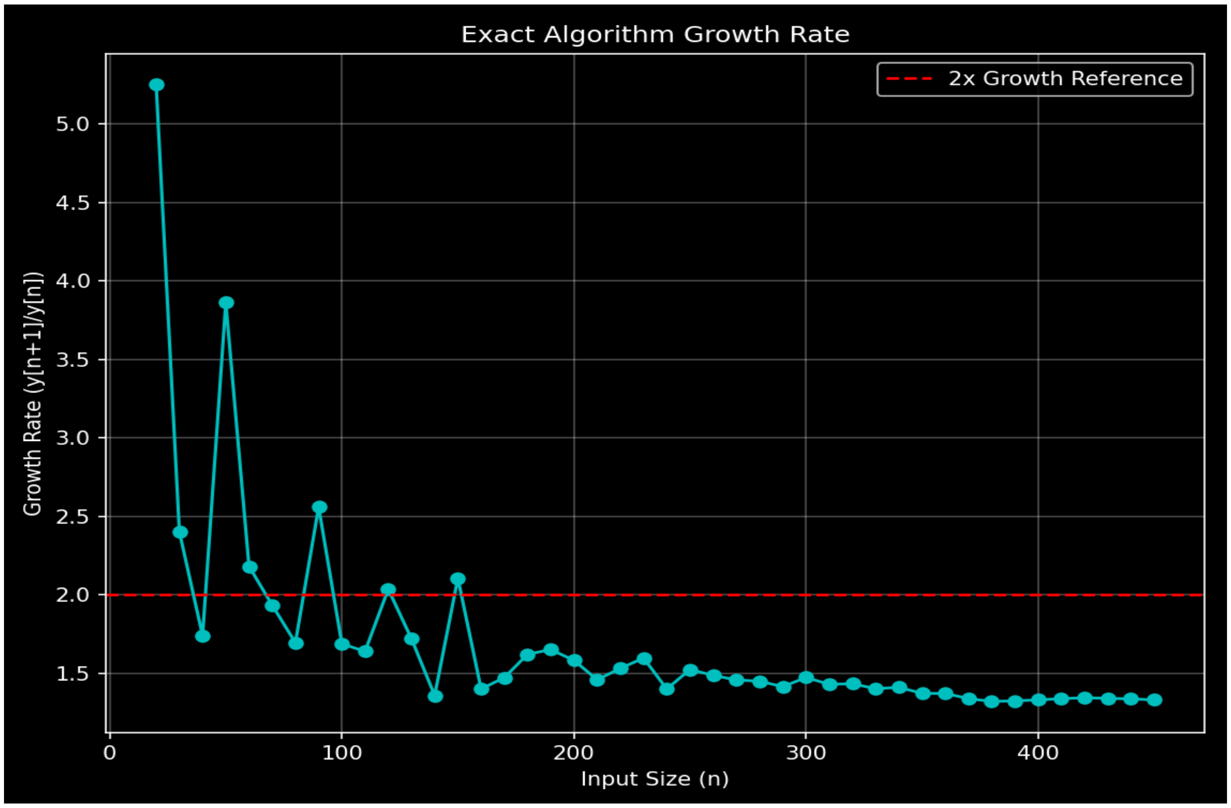
If you don't know recursion though, this could become a little more of a headache as we use a lot of it in a intuitive way by making 2 recursive calls for inclusion-exclusion for each item. The reduction of capacity and addition of profit and how this works with our base case can inherently also just cause confusion for the novice to recursion which is something to amply consider in our solution which is aided in our heuristic solution.

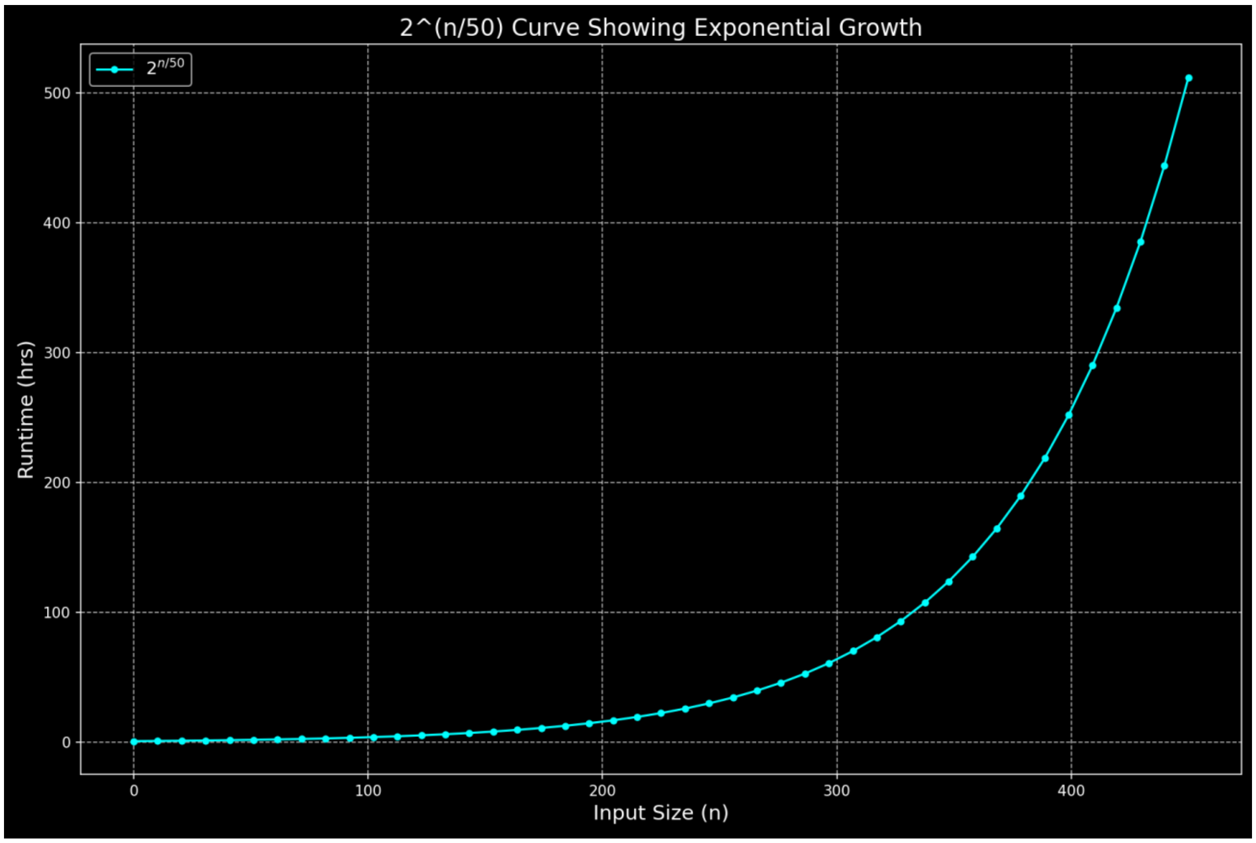
Data analysis performed within the code:

We do not have any means of efficient early dropout and because we have no ordering assumptions of use, this means we don't know if our last item should be considered in any of the previous knapsacks unless we check it. This means for our solution we consider including or excluding every item with every given intermediate valid knapsack of items causing us to check all subset combinations. Unless the capacity is filled fully—or our item's current weight will not fit in this current knapsack—we will not have an early dropout of a recursive chain, and even then, this is only one assortment of our knapsack so isn't actually of runtime benefit for our **O(2^n)** runtime complexity as is only a handful of best-cases where this would be triggered so will normally be doing around **2^n** cases.

Observed (actual) performance on each data set:

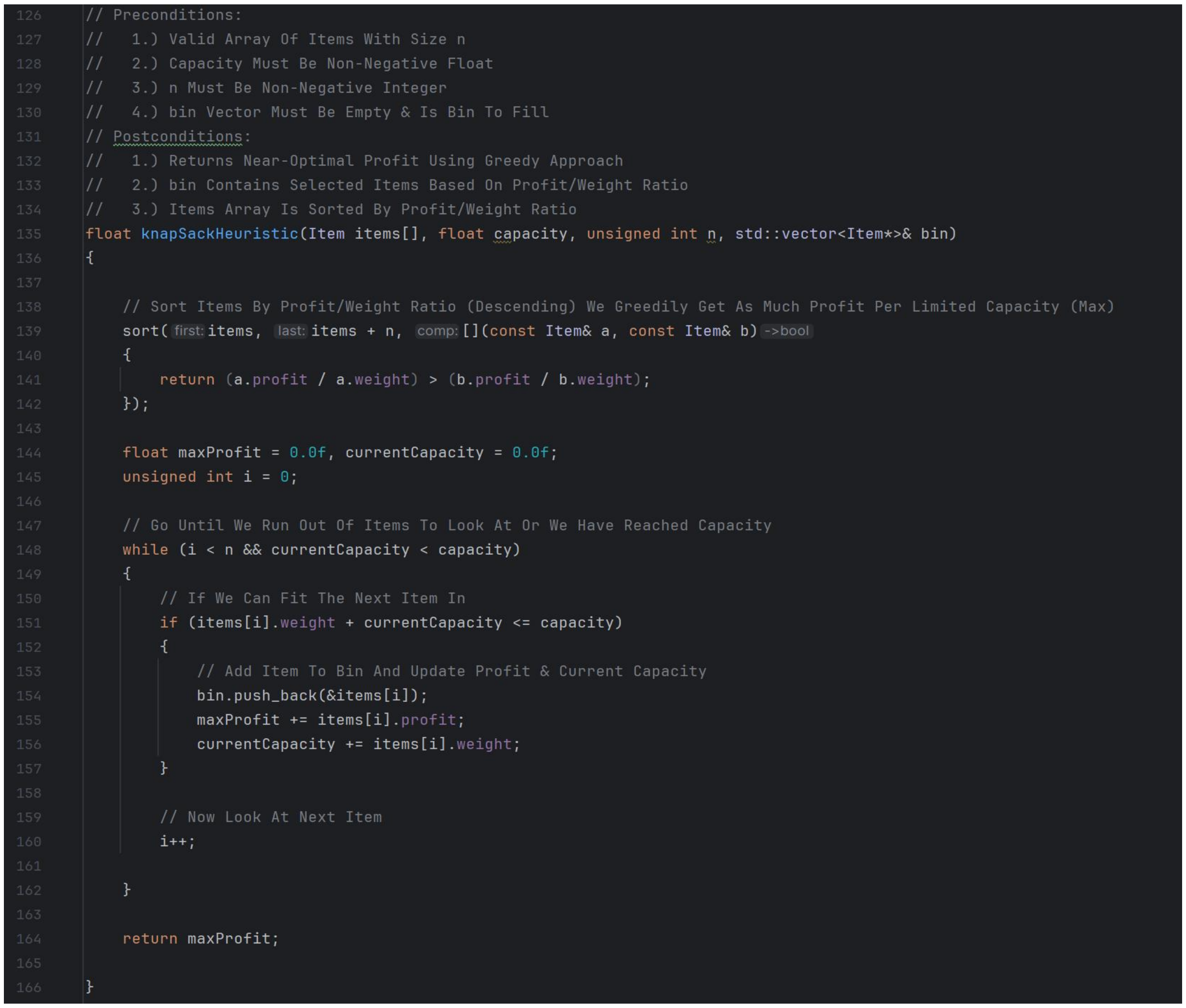






When looking at our data provided through the datasheet as well as from the graphing the data for the exhaustive implementation, we see how visually our data actually follows a very similar curve to an exponential curve like the one of 2^n. If we look at the growth rate graph though, we see interesting results as while it seems to be at a rate of change of around 2x during each increment, its actually extremely varying and generally settles ~0.75x. This is very interesting as it doesn't show a consistent trend. When we graph a scalar of 2^n we can see that our trendline looks very similar to the one we found in our dataset. Even though we are having n divided by 50, it still falls in the class of **O(2^n)** as constants do not effect the general complexity class of an implementation and only mean we have our doubling happening in smaller increments but still is dependent on n. The spikes and dips in growth rate could also be chalked up to caching optimizations and other CPU optimizations going on when dealing with these reproducible recursive chains.

Heuristic Implementation:



```cpp
// Preconditions:
//   1.) Valid Array Of Items With Size n
//   2.) Capacity Must Be Non-Negative Float
//   3.) n Must Be Non-Negative Integer
//   4.) bin Vector Must Be Empty & Is Bin To Fill
// Postconditions:
//   1.) Returns Near-Optimal Profit Using Greedy Approach
//   2.) bin Contains Selected Items Based On Profit/Weight Ratio
//   3.) Items Array Is Sorted By Profit/Weight Ratio
float knapSackHeuristic(Item items[], float capacity, unsigned int n, std::vector<Item*>& bin)
{

    // Sort Items By Profit/Weight Ratio (Descending) We Greedily Get As Much Profit Per Limited Capacity (Max)
    sort( first: items,  last: items + n,  comp: [](const Item& a, const Item& b) ->bool
    {
        return (a.profit / a.weight) > (b.profit / b.weight);
    });

    float maxProfit = 0.0f, currentCapacity = 0.0f;
    unsigned int i = 0;

    // Go Until We Run Out Of Items To Look At Or We Have Reached Capacity
    while (i < n && currentCapacity < capacity)
    {
        // If We Can Fit The Next Item In
        if (items[i].weight + currentCapacity <= capacity)
        {
            // Add Item To Bin And Update Profit & Current Capacity
            bin.push_back(&items[i]);
            maxProfit += items[i].profit;
            currentCapacity += items[i].weight;
        }

        // Now Look At Next Item
        i++;

    }

    return maxProfit;

}
```

Algorithm Complexity:

In the heuristic implementation for the 0-1 knapsack, it works quite straight forward. We are provided 4 arguments:

1. The Items to be considered for the knapsack
2. The current capacity of the knapsack
3. The amount of items to be considered
4. The current knapsack (more optional as not specifically required by problem)

In our implementation it mainly relies on the "Standard Greedy" heuristic to maximize the knapsack's profitability. The "Standard Greedy" approach works by finding the best profit-per-weight and sorting based on this as this ensures for our limited space, we are making a good decision to maximize the profit we generate per each individual piece of weight. Utilizing this heuristic makes us employ a pseudo greedy paradigm into our implementation (as we don't get an optimal answer from our subproblems but follow the same logic) which is good as it allows us to work through our scan as if we are doing the exhaustive inclusion-exclusion approach of looking at an item and never again after (albeit this being much less computationally and comprehensively insignificant).

After we sort our items based on descending order of profit-per-weight (which incurs us a **O(n*log(n))** runtime complexity) we can simply tally our profit (and add the item into the return bin) through a simple scan (**O(n)**) through our list of items going until we include all our items, or we run out of space to accommodate more items. While we do have some constant time operations like the variable initialization and conditionals, these are negligible in our analysis. This means of solving the problem sub optimally greatly simplifies the computations done, making our runtime complexity **O(n*log(n))** at the expense of consistently optimal answers.

Code Complexity:

The code complexity of this heuristic implementation is very easily understood. This can firstly be seen in it being iterative-based as iteration usually is seen as much easier to follow trace compared to recursion which can have many "invisible" actions going on that aren't easily able to be mentally imagined.

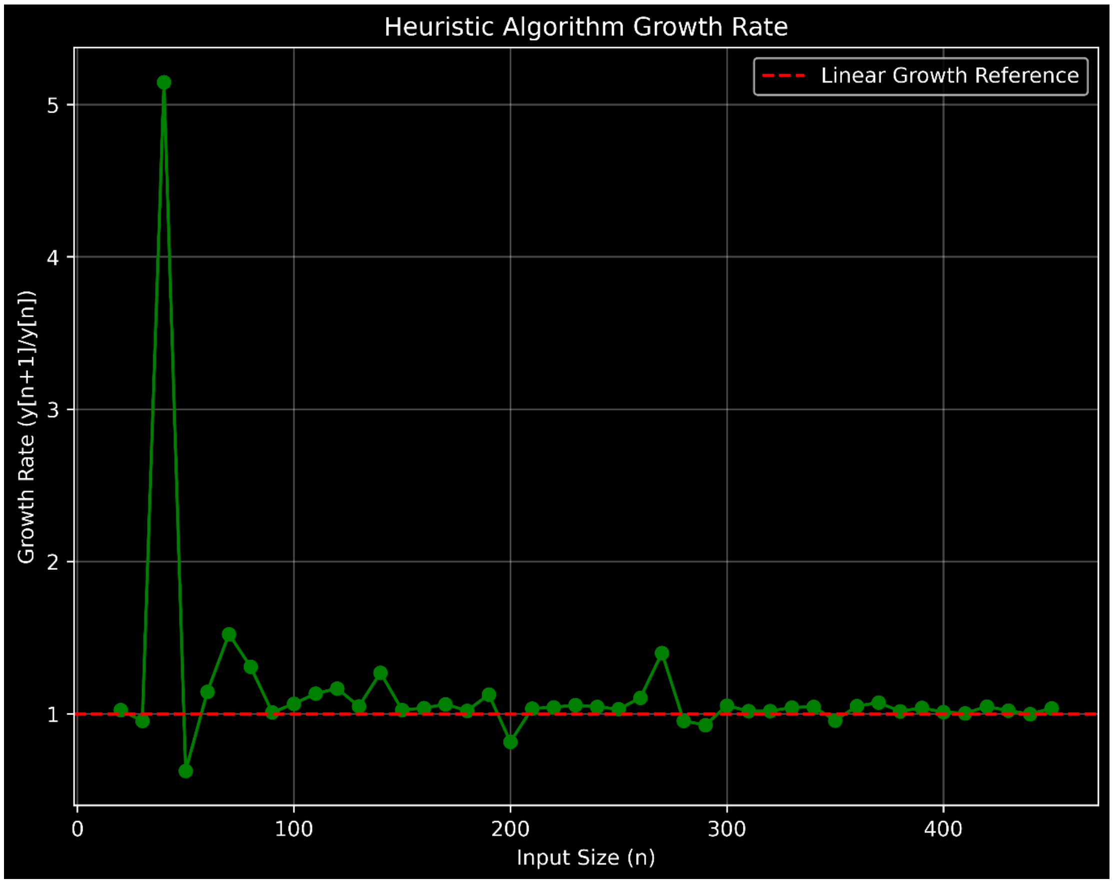
Other than the iteration, the intuition provided in this codebase is great as if you read the problem statement of the 0-1 knapsack you can easily pick up on sorting based on profit-per-weight as a very strong means of solving this problem. It may be a little difficult without examples though for people to understand the suboptimality of this heuristic based solely on the code provided though, as intuitively you expect packing based on profit-per-weight as being the optimal way but after going through examples you start seeing its hidden flaws.

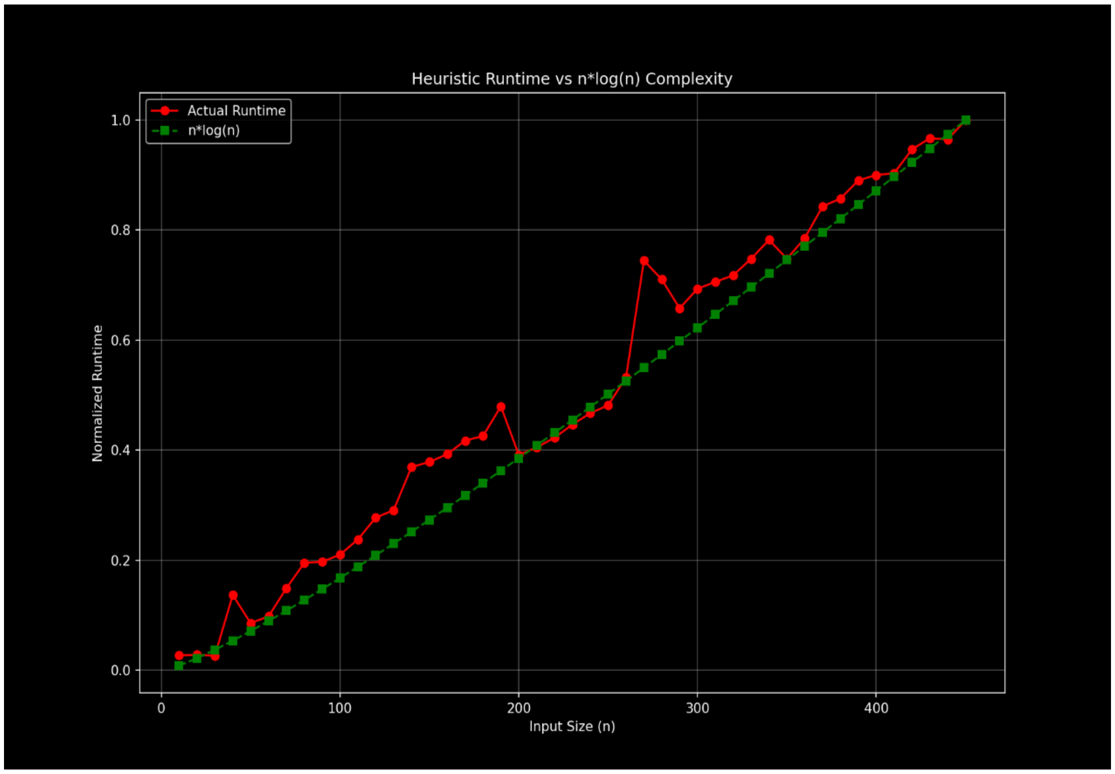
Walking through the sorted item list and tallying the max profit also can be something easily understood from the provided codebase and even without knowledge of the greedy paradigm you understand why/how we are scanning and adding items based on the means of sorting as if one item cant be added, adding another item won't change this fact so we can simply iterate forward without ever backtracking. This makes our conditionals for our while loop easy to pick up and understand the context of. It may be hard for them to understand that because we aren't sorting based on weight, we don't terminate our scan after one item cannot be added as another item can have a lower profit-per-weight and still have a weight that could be added.

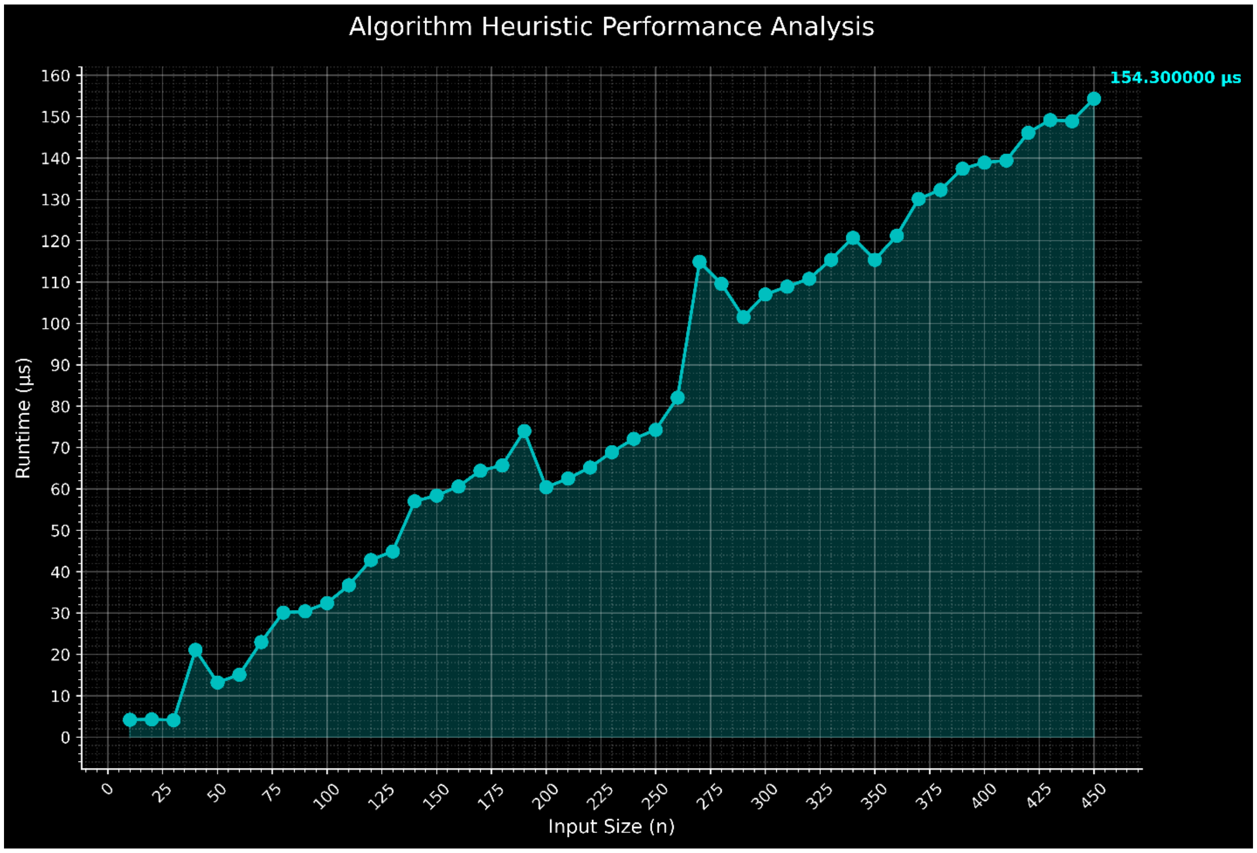
Data analysis performed within the code:

For our heuristic implementation we have a quite dynamic means of data analysis going on as after we sort we at worst case could be scanning all the items we have in our sorted item set if our knapsack is never fully filled as just because one item cannot fit—based on our sorting of profit-per-weight—we really have no assumptions on the weight amongst list entries so we must exhaustively scan our sorted list to ensure that if the last item in the sorted list with the lowest profit-to-weight is the only item that could be added to the knapsack it is without early termination. If we do get to our maximum capacity though exactly, we could then do an early dropout as we wouldn't be able to include any more items that would increase our profit as our profit-per-weight will ensure that element **i** in the sorted list will always give us more profit than element **i-1**. Traditionally though, we will be doing around **n** cases as instead of looking at all subset combinations like the exponential solution, we are only looking at each item once for only one knapsack instance.

Observed (actual) performance on each data set:



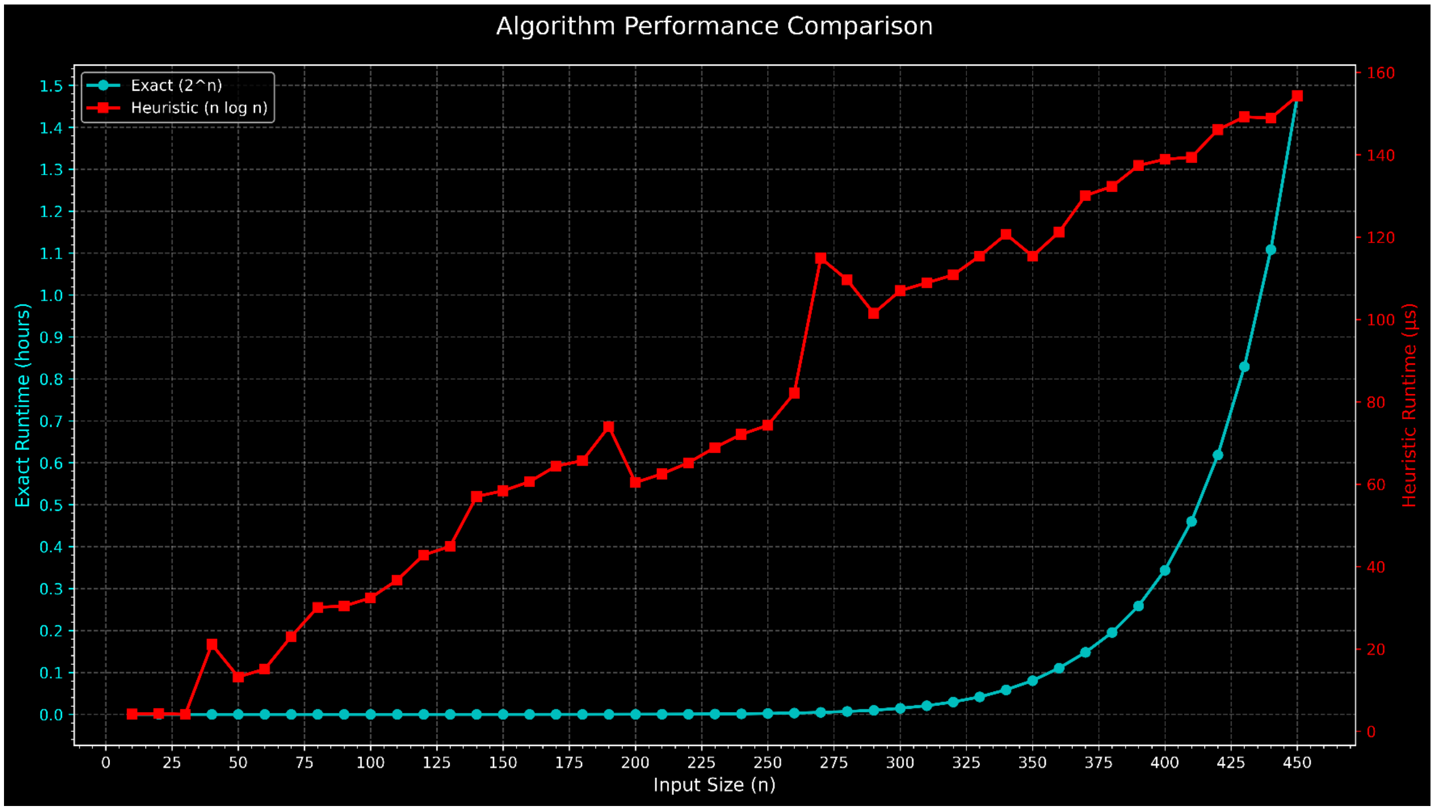




When looking at our data provided through the datasheet as well as from the graphing the data for the Standard Greedy implementation, we see how visually our data actually follows an extremely similar curve to a logarithmic pseudo-linear curve like the one of n*log(n) when normalized to [0.0, 1.0]. When looking at the growth rate, we see it also follows our expected pseudo-linear nature of log curves very closely helping support that our implementation is **O(n*log(n))**. Our rises and dips in runtime complexity could be from cache optimizations, early terminations on smaller datasets, and varying input properties, which could practically affect our runtime. We also see how our runtime is in microseconds compared to the exhaustive implementation which is in hours, showing just how much runtime benefits a heuristic can give us at the cost of suboptimality.

6. Implementations Comparison

Algorithmic complexity:



When looking at the two implementations, there are obvious differences between their algorithmic complexities but with their tradeoffs based on their optimal and suboptimal nature. Our optimal exhaustive algorithm runs with a runtime complexity of **O(2^n)** based on its recursive means of inclusion-exclusion to consider all subset combinations to find our optimal packing, making us roughly double our runtime when our input size of n increases. This becomes impractical very quickly as even though we are getting the most optimal answer the runtime when n is around 300 starts gradually increasing so that even further when we get to an n of 440 it is noticeably increasing in time.

With the Standard Greedy Heuristic implementation, because we are not finding an optimal answer, we can start using approximations, which use some relationship about our data to help give an educated guess as to what item should be added next to the knapsack. Our educated guess for this knapsack problem is that the profit-to-weight ratio will help maximize the profit we fit into our knapsack for our limited space. This sorting for this ratio allows us to run in **O(n*log(n))** as we simply sort and scan through the ratios by best profit-to-weight margin to worst, adding the best that fits at our current capacity. While this is suboptimal and won't always be right, it is much faster and scales better with n compared to our exhaustive implementation, it also is very close to optimal a good portion of the time making it good for large datasets.

This helps show how each implementation has its own use-case in pros-and-cons for runtime complexity and their applicability in the specific domain as if you know n is on the low end, maybe a optimal exhaustive search is what you want but this is unrealistic for big n.

Code complexity:

The two code complexities severely contrast; with the optimal exhaustive implementation utilizing a recursive structure for a quite conceptually straightforward idea of including an item or excluding an item and can be very trivial if having a solid idea of the problem statement and recursion. Recursion though—while trivial if you have worked with it—has a very aggressive learning curve to feel comfortable in and mentally trace through stack frames. Mainly with a problem like inclusion-exclusion which has many off-shooting states that are all being aggregated into one optimal solution as tracing through this can be a nightmare if not confident on just how we reach our leaf nodes and work up the chain. Understanding how the base cases are utilized and how profit and our containers are being passed up the call stack is a little complex and is something to greatly consider when implementing this algorithm in the real world as we also have to consider call stack overhead from the usage of recursion.

With the Standard Greedy Heuristic, it's much more digestible as it is iterative, making it much easier to process and trace through. Being able to sort on profit-per-weight ratio is a very intuitive and understandable means of trying to find the maximum knapsack profit with our limited space, and the scan through after our sorting, adding the next highest profit ratio based on our capacity, is very simple to understand and efficient. This heuristic though can be deceptively easy to digest but hard to understand it's negatives and why we actually wouldn't get optimal packings of our knapsack which is interesting to consider when applying it to the real world as people may need to see examples to see the suboptimality of this implementation. This isn't as much a problem with the optimal implementation which is much more brute force in approach.

Data analysis performed with code:

The exhaustive algorithm exhaustively has to check all **2^n** possible subsets of items. This can come from its inherent brute-force nature, as for every inclusion and exclusion of items, we must compute and check to ensure if this is either the optimal packing of our knapsack or not. Because our algorithm also has no assumptions on ordering, we have no means of effective early dropout of our recursive chains to do. We do, though, have one if the knapsack is currently at full capacity, as then we know we cannot fit any more items into this current knapsack, so we can drop out early, but this realistically won't do anything visible to our runtime as this only terminates one branch, causing us to only be reduced to a branch count of **2^n – 1** which is very negligible. This is likewise for our item's weight exceeding the current capacity as we still need to look through our exclusion recursive chain just not our inclusion chain. This causes our exhaustive algorithm to check basically all our subsets, with early terminations being a rarity.

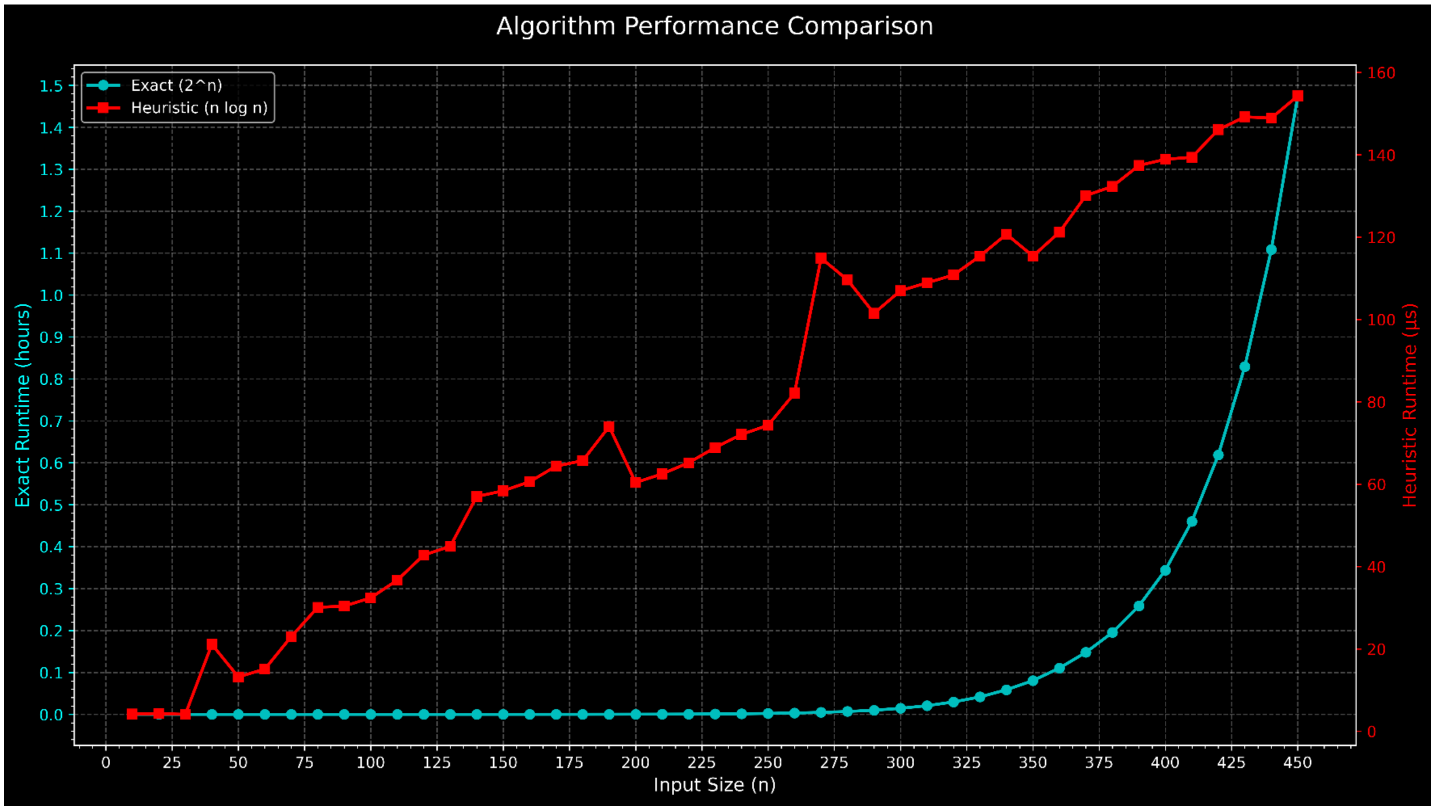
To avoid this computationally heavy subset check, the heuristic implementation focuses on sorting and scanning—basically organizing our data based on some relation—and looking at the entries left-to-right. This very simplistic way of sorting the data and checking it is a process naturally proportional to our item count, **n**. This process of sorting (**O(n*log(n)**) followed by a single scan (**n**) comes to:

n*log(n) + n

Showing how runtime is proportional to our input, n. We benefit much more in this heuristic than in the exhaustive algorithm for early dropouts, from our sorted nature, we know when our knapsack is full, we don't need to look at any more items to the right of this item as they will all provide us a poorer profit-per-weight. This means that if we have 100 items and sort them based on profit-per-weight and we then put in the highest profit-per-weight item, and it fills our capacity, we don't have to look at the other 99 items as we maximize our profit-per-weight, which our heuristic wishes to achieve.

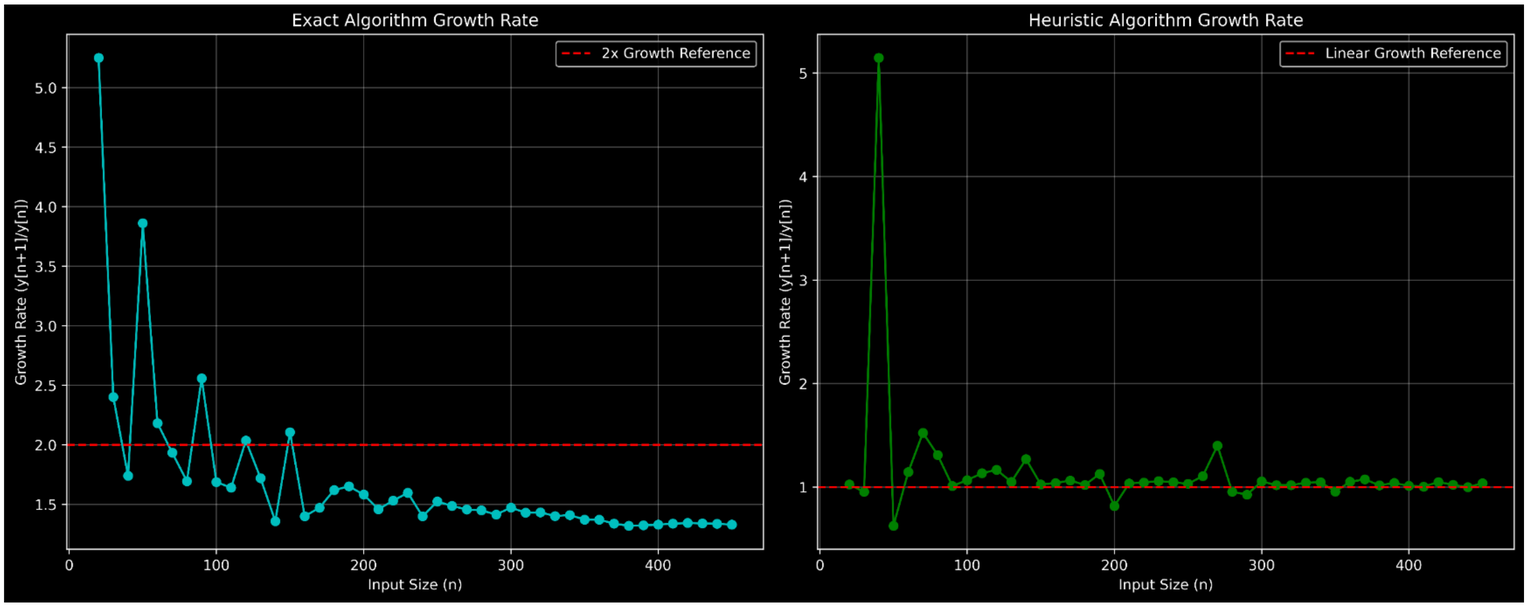
So, while we get an optimal answer by looking at all **2^n** subsets and incur this computational overhead, the heuristic only needs to check **n** items (after our sorting) at their worst cases. While the exhaustive algorithm has an early dropout, it isn't ever used compared to the greedy, which utilizes its sorted relationship between items to help drop out of scans.

Observed (actual) performance of each implementation:



When running our exhaustive algorithm we see how when our n increases, the runtime follows the **O(2^n)** complexity curve, slowly growing speed in the rate of change as n gets bigger. For the small n, like n = 10 and n = 20 we have a difference of less than **20** milliseconds then going up to **~1.5 hours** at **n = 450**. This gives us a general range in milliseconds of our runtimes for **n = 10** to **n = 450** of **[0.0028, 1.5 hours).** Our datasets don't really affect too much the exhaustive algorithm as it will a large majority of the time be checking **2^n** cases even with redundant subsets.

For our heuristic, on the other hand, while it does increase with n, the runtime follows the **O(n*log(n))** curve from its most computationally intensive logic being sorting, which dominates the trends of the curve compared to the linear scan of **O(n)** afterward. With our runtimes tracked in milliseconds as well for **n** = **10** and **n** = **450** we get the range of **[0.0042, 0.1543],** which is much more manageable than the exhaustive solution. Even though for **n** = **10**, we see the exhaustive implementation having an overall faster runtime in milliseconds, this is from a very small n which is understandable as sorting can be expensive when the dataset isn't verbose enough. The runtime could also be seen being this low from the early dropouts the heuristic can make in comparison with the exhaustive implementation and it being much more thorough.



We can tell also through both growth rates that they both suffered from some spikes and divots perhaps from means of CPU optimization from the more redundant actions like recursive chains or sorting and scanning logic.

While exhaustive is optimal this optimality comes at a huge cost as it makes our implementation very difficult to scale as at **n** = **300** we start seeing gradual changes in our runtime which further is compounded the higher we go beyond **n** = **300**. This is different than the heuristic which even throughout all the entries of n, had a very consistent rate of change and overall runtime overhead **[0.0042, 0.1543]** with it also being in the time units of microseconds with the exhaustive being in terms of hours which even further emphasizes how major these two implementations are and just how expensive optimality can be mostly in a scalable setting.

We see this increase impact of input size on our implementations quite easily as with n in our exhaustive solution when we get to high values of n, it is visually obvious this doubling process that is going on (albeit not exactly 2^n) it is easier to see between smaller entries with dramatic jumps like **n** = **40** to **n** = **50** with **~0.0614 milliseconds** to **~0.2371 milliseconds**. While the Standard Greedy Heuristic isn't exactly fully linear, the logarithm works on a quasi-linear manner with n, meaning our n generally trends in a linear manner mainly for the range of n we have studied in this report as even between n of 10 and an n of 450 (x45) we only get a solutions in microseconds.

The knapsack size doesn't really effect the exhaustive solution that much mostly if no subset can actually provide a full filling with zero capacity left over as even if the weight is exceeding for a specific item it will still look at all other items in the recursive chain to see if any of these items could be considered as the computer wont know until it checks so if no item ever fits in the knapsack we could only do **O(n)** recursive calls with all them being excluded but this is not that realistic in practice as at least one item usually can fit in the knapsack which we may not know the location of in the item list unless exhaustively checking.

The capacity of the knapsack plays a significant role in the heuristic's performance though. Once the knapsack is full, the algorithm terminates early, skipping the remaining items in the sorted list. This makes the runtime dependent on how quickly the knapsack reaches its maximum capacity. Larger capacities may delay early termination, slightly increasing runtime. However, this effect is negligible compared to the overall **O(n*log(n))** complexity.