Assignment Five

Josh Seligman

joshua.seligman1@marist.edu

November 17, 2022

1 Fractional Knapsack Algorithm

1.1 The Algorithm

The fractional knapsack algorithm solves the problem of maximizing the value of the objects one takes to fill up their knapsack. As displayed in Algorithm 1, the solution to the fractional knapsack problem requires a greedy approach by getting the most valuable spice available at each point the algorithm. To do this, the spices are sorted by their unit price to get the most value for the amount of spice taken. The loop on lines 7-18 will continue until the knapsack is full or until there are no more spices to consider and, as previously mentioned, will take as much of the most valuable spice that is available. Since the spices are sorted ahead of time, the greedy approach of taking the local maximum value and hope it leads to a global maximum value works because the spices are not changing and the unit prices of the spices taken will continue to decrease as the algorithm is run. This ensures that the global maximum value is always achieved for any set of spices and any knapsack capacity.

1.2 Asymptotic Analysis

Listing 1 contains the C++ implementation of the fractional knapsack algorithm. First, line 3 makes a call to a quicksort algorithm for the spices to put them in descending order by their unit prices, which runs in $O(s*log_2 s)$ time, where s is the number of spices available to be taken. Next, the loop defined on line 5 iterates iterates through each of the knapsacks because the implementation takes in many knapsacks for testing. This loop will run k times, where k is the number of knapsacks. The sorting algorithm is executed before the loop because all of the knapsacks are being filled with the same data and, therefor, the sorting operation only has to be done once. Line 6 is also specific to the implementation as it dequeues the next knapsack from the queue, which is a constant time operation. Lines 9, 15, 16, and 19 all define variables to keep track of, which are all constant time operations. The small loop on lines 10-12 initializes the array to use all 0s, which is a C++ specific problem as other programming languages do this automatically, which will cause these lines to be excluded from the asymptotic analysis. Next, the loop defined on line 22 continues until the knapsack is full or until there are no more spices. The worst case scenario is when there is not enough spice to fill the knapsack, which causes the loop to run a total of s times. The entire body of the loop matches what is done in Algorithm 1, which is only the constant time operations of conditions and assignments. Thus, the entire loop on lines 22-43 runs in $O(s)$ time. Lastly, lines 46-62 will be excluded as they are outputting the results.

Algorithm 1 Fractional Knapsack Algorithm.

procedure FRACTIONALKNAPSACK(*spices*, *capacity*)
  sort(*spices*) // *Sort the spices by unit value, descending order*
  $quantityTaken \leftarrow$ *new int*[*spices.length*] // *Store an array to keep track of how much of each spice was taken*
  $capacityLeft \leftarrow capacity$ // *Start with empty knapsack*
  $totalValue \leftarrow 0$ // *Start off with no value*
  $curSpiceIndex \leftarrow 0$ // *Start with most valuable spice per unit*
  while $capacityLeft > 0$ && $curSpiceIndex < spices.length$ **do**
    if $capacityLeft \geq spices[curSpiceIndex].quantity$ **then** // *Enough space to take everything*
      $quantityTaken[curSpiceIndex] \leftarrow spices[curSpiceIndex].quantity$
      $capacityLeft \leftarrow capacityLeft - spices[curSpiceIndex].quantity$
      $totalValue \leftarrow totalValue + spices[curSpiceIndex].value$
    else // *Take what we can*
      $quantityTaken[curSpiceIndex] \leftarrow capacityLeft$
      $totalValue \leftarrow totalValue + capacityLeft * spices[curSpiceIndex].unitPrice$
      $capacityLeft \leftarrow 0$
    end if
    $curSpiceIndex \leftarrow curSpiceIndex + 1$ // *Move on to next spice*
  end while
  return *quantityTaken*, *totalValue*
end procedure

Overall, for each individual knapsack, the runtime complexity is $O(s*log_2 s+s)$, which is $O(s*log_2 s)$ because $s*log_2 s$ is the dominant term in the expression. However, the implementation runs the $O(s)$ loop k times, which will cause the overall runtime complexity of the C++ implementation to become $O(s*log_2 s+k*s)$.

2 Appendix

2.1 Fractional Knapsack Algorithm

```
1  void runAlgo(SpiceArr* spices, Queue<int>* knapsacks) {
2      // Start off by running a sort on the spices array to make them in descending order
3      quickSort(spices);
4
5      while (!knapsacks->isEmpty()) {
6          Node<int>* curKnapsack = knapsacks->dequeue();
7
8          // Create an array that corresponds with the spice array for keeping track of what
               was taken by the knapsack
9          int quantityTaken[spices->length];
10         for (int i = 0; i < spices->length; i++) {
11             quantityTaken[i] = 0;
12         }
13
14         // Start off with an empty knapsack and a value of 0
15         int capacityLeft = curKnapsack->data;
16         double spiceValue = 0;
17
18         // Start considering the first element in the array (most valuable per unit)
19         int spiceIndex = 0;
20
21         // Continue until the knapsack is full or until there is no more spice to take
22         while (capacityLeft > 0 && spiceIndex < spices->length) {
23             // If there is space for the entire pile of spice, take it all
```

```
24              if (capacityLeft >= spices->arr[spiceIndex]->getQuantity()) {
25                  // Update the array of spice taken
26                  quantityTaken[spiceIndex] = spices->arr[spiceIndex]->getQuantity();
27
28                  // Be greedy and take everything available if possible
29                  capacityLeft -= spices->arr[spiceIndex]->getQuantity();
30                  spiceValue += spices->arr[spiceIndex]->getPrice();
31              } else {
32                  // Update the table entry
33                  quantityTaken[spiceIndex] = capacityLeft;
34
35                  // Compute the value of the spice we can take
36                  spiceValue += capacityLeft * spices->arr[spiceIndex]->getUnitPrice();
37
38                  // Update the capacity to be 0
39                  capacityLeft = 0;
40              }
41              // Go on to the next spice
42              spiceIndex++;
43          }
44
45          // Start with this text
46          std::cout << "Knapsack of capacity " << curKnapsack->data << " is worth " <<
                spiceValue << " quatloos and contains";
47
48          // Iterate through all of the spices
49          for (int j = 0; j < spices->length; j++) {
50              // Only print out the spices we take
51              if (quantityTaken[j] > 0) {
52                  // Little formatting logic
53                  if (j > 0) {
54                      std::cout << ", ";
55                  } else {
56                      std::cout << " ";
57                  }
58                  // The amount and name of the spice taken
59                  std::cout << quantityTaken[j] << " scoops of " << spices->arr[j]->getName();
60              }
61          }
62          std::cout << "." << std::endl;
63
64          // Memory management
65          delete curKnapsack;
66      }
67  }
```

Listing 1: Fractional Knapsack Algorithm (C++)