ALGORITHMS HOMEWORK 5

VICTOR BANKSTON

Problem 1. Consider the following task: Given an unsorted array of n numbers, find the k smallest numbers and output them in sorted order. Describe three inherently different algorithms to solve this problem. Analyze their runtimes in terms of n and k (so you should have n and k in the big-Oh runtime). Try to find the fastest possible algorithm. Which of your algorithms is fastest?

We can find the k^{th} smallest number using the order statistics algorithm (which is essentially quicksort where you only recurse into one of the partitions). Then, we can step through the array and select the elements which are smaller or equal to than this k^{th} element. Finally, we sort the new array. This takes expected $\Theta(n + n + k\log k) = \Theta(n + k\log k)$ time. If the list of numbers contains redundancies, we may need to remove some of the maximal entries (those equal to the k^{th} smallest element), but this will take linear time.

Alternatively, we could sort the entire array and output the first k elements, which takes $\Theta(n\log n + k)$. Obviously, this is inherently slower than the first method, since it involves sorting the entire array rather than a sub-array. Since $n \geq k$, $\Theta(n + k\log k) \subset \Theta(n\log n + k)$.

Finally, we can apply the order-statistics algorithm k times, first outputting the smallest number in the array, then the $2nd$ smallest, and so forth until the k^{th} smallest for a total runtime of $\Theta(kn)$.

In analyzing these algorithms, we consider two limiting cases. When $k = 1$, and our problem reduces to exactly an order statistics problem. In this case the first and third algorithms have the same asymptotic runtimes, though the third will be faster. When $k = n$, and the problem reduces to sorting. In this case, the first algorithm is optimal.

Problem 2. It is October and you are preparing for Halloween. You have a fixed budget of B dollars to spend, and you found three different types of candy that you are considering to buy: Caramel apples, chocolate bars, and gummy bears. Of course your goal is to maximize the happiness of the children you give the candy to, but at the same time you have to stay within your budget. The three different types of candy have different costs: $c_{apples} > c_{chocolate} > c_{gummy}$. But you know that some of the children prefer the caramel appels and some prefer the chocolate bars. In fact, you know the children that come by your door at Halloween pretty well, and you know all of their candy preferences. For the i-th child, let $H_{apples}[i], H_{chocolate}[i], H_{gummy}[i]$ be the child's happiness for the respective candy type. These are all positive integer values.

Suppose there are n children. Your goal is to buy one piece of candy for each child such that the total happiness of all children is maximized, while thot tal cost of candy is wihin your budget B.

a) Suppose there are $n = 4$ children, and $c_{apples} = \$5$, $c_{chocolate} = \$2$, $c_{gummy} = \$1$. And the happiness values are given in the following arrays: $H_{apples} = [4, 1, 2, 4]$, $H_{chocolate} = [1, 3, 4, 2]$, $H_{gummy} = [2, 1, 3, 3]$. What selection of candies maximizes the total happiness if you can spend at most $B = \$9$? What is the solution for $B = \$8$?

b) Let $h(i, b)$ be the maximum total happiness for the first i children with total cost less than or equal to b. Provide a recusive formula for $h(i, b)$.

c) Give pseudocode for a dynamic programming algorithm to compute $h(n, B)$. What are the runtime and space complexity of your algorithm?

d) Give pseduocode for tracing back the DP table to compute an optimal candy selection. What is the runtime of your algorithm?

a) To answer this question, we should first normalize the children's preferences so that their happiness is expressed per dollar. $H'_{apples} = \left[\frac{4}{5}, \frac{1}{5}, \frac{2}{5}, \frac{4}{5}\right]$, $H'_{chocolate} = \left[\frac{1}{2}, \frac{3}{2}, 2, 1\right]$, $H'_{gummy} = [2, 1, 3, 3]$. By inspection it looks like the maximum should occur when buying an apple for the first child, chocolate for the second, and gummies for the last two. The total cost of this arrangement is \$9, and the total happiness is 13. When $B = 8$, we should second child a gummy for a total cost of \$8 and total happiness of 11.

b) For the i^{th} child, we can buy any of the 3 chocolates. Therefore,

$$h(i, b) = \max\left(H_{apples}[i] + h(i-1, b - c_{apple}), H_{chocolate}[i] + h(i-1, b - c_{chocolate}), H_{gummies}[i] + h(i-1, b -$$

c)

```
for $i = 1$ to $n$:
    for b in -5 to $B$:
        if $b < 0$ :
            $h_{0,b} = -\infty$
        $h_{0,0} = 0$
        else
            $h_{i,b} =$
             $\max\left(H_{apples}[i] + h_{i-1,b-c_{apples}}, H_{chocolate}[i] + h_{i-1,b-c_{chocolate}}, H_{gummies}[i] + h_{i-1,b-c_{gummy}}\right)$
return $h_{n,B}$
```

Algorithm 1: candy

My algorithm runs in time and space $\Theta(nB)$, since the table $h_{i,b}$ has this size, and it takes constant time to fill out each entry. However, it would be possible use only $\Theta(B)$ space. After computing $h_{i,B}$, we can forget all entries $h_{i-1,b}$, because when we compute $h_{i+1,b}$, we only need entries of the form $h_{i,b}$. If we do this, we will no longer be able to back-track. If we wanted to be able to back-track, we would need to store in $h_{i,b}$ the list of i candies which achieves that cost. Such a list require i bits per table entry, so our total space cost is still $\Theta(nB)$.

d) We start with $h_{i,B}$ and determine the candy for which $h_{i-1,b-c_{candy}} + H_{candy}[i] == h_{i,b}$. This will determine the candy we should buy for the i^{th} child. It takes constant time, because we just need to check the equality condition for each of the 3 candies. Having found the candy for the i^{th} child, we can recurse and perform the same backtracking on $h_{i-1,b-c_{candy}}$. The whole process will take linear time.

Problem 3. Let $A[1..n]$ be an array of n integers (which can be positive, negative, or zero). An interval with start-point i and end-point j, $i \leq j$, consists of the

```
$b = B$
for $i = n$ to 1:
    if $h_{i,b} == H_{apples}[i] + h_{i-1,b-c_{apple}}$:
        output.append("apple")
        $b- = 5$
    else if $h_{i,b} == H_{chocolate}[i] + h_{i-1,b-c_{chocolate}}$:
        output.append("chocolate")
        $b- = 2$
    else if $h_{i,b} == H_{gummies}[i] + h_{i-1,b-c_{gummy}}$
        output.append("gummies")
        $b- = 1$
return reverse(output)
```

Algorithm 2: CandyList

numbers $A[i], \ldots, A[j]$ and the weight of this interval is the sum $A[i] + \cdots + A[j]$. The problem is: Find the interval in A with maximum weight.

a) Describe an algorithm for this problem that is based on the following idea: Try out all combinations of i, j with $1 \leq i < j \leq n$. What is the runtime of this algorithm?

b) Describe a dynamic programming algorithm for this problem. Proceed in the following steps:

i) Develop a recusive formula for the following entity: $S(j)$ = maximum of the weights of all intervals with endpoint j.

ii) Describe an algorithm that computes all $S(j)$ in a dynamic programming fashion based on the recursive formula, and afterwards determines the end-point $j^\star$ of an optimal interval. You can describe your algorithm in words.

iii) Given the end-point $j^\star$ describe how to find the start point $i^\star$ of an optimal interval by backtracking.

iv) What are the runtime and space complexity of the overall algorithm?

a) If we test all combinations of i and j, it will take $\Omega(n^2)$ time, since there are $\binom{n}{2} \in \Theta(n^2)$ combinations for i and j. However, it will actually take more time than this because there is some time associate with performing the sum $\sum_{k=i}^{j} A[k]$. Let $d = j - i$, and observe we have $n - d$ combinations for which $j - i = d$. Such a combination will require $d - 1$ additions to compute the sum. Thus, the total runtime will be $\sum_{d=1}^{n} (n-d)(d-1) = \sum_{d=1}^{n} nd - d^2 + d - n \in \Theta(n^3)$.

b) i) There are two options: either the maximal interval contains j alone, or it contains $j - 1$. If it contains $j - 1$, then it had better contain the entire maximal interval ending at $j - 1$. Otherwise, we could get a larger interval by adding in the elements of the maximal interval ending at $j - 1$. Our recursive formula is then $S(j) = \max(S(j-1) + A[j], A[j])$.

ii) The recursive formula, computed with memoization, constitutes a dynamic programming algorithm. We need to supply the base case: $S[1] = A[1]$. Set $j^\star = 1$. Then, for $i = 1$ to n, we compute $S(j)$ in constant time using our recursive formula. If $S(j) > S(j^\star)$, set $j^\star = j$. Finally, we return $j^\star$.

iii) To recover the starting point of the maximal interval for $j^\star$, we set $i^\star = j^\star$ and $Sum = 0$, and check whether $Sum + A[i^\star] == S(j^\star)$. If not, we set $Sum+ = A[i^\star]$ and decrease $i^\star$ by 1 and repeat .

iv) Our table takes linear space and each entry requires constant time to compute, so our algorithm takes $\Theta(n)$ space and time. Backtracking will also require linear time.

Actually, we could do this with only constant space, since our recursive formula only involves the previous element in our table. If we do this, we need save the value $S(j^{\star})$ along with $j^{\star}$ to perform the backtracking.