Homework 3

MSSC 6000 — Scientific Computing — Spring 2023

Due Wednesday, March 8, on D2L, by 11:59pm

Read this first! Some of these questions instruct you to devise an algorithm to solve a problem and explain it to me. In this assignment, approach describing your algorithm like writing a report to a boss or an advisor. Your goal is for me to read your answer and have a full understanding of how it works. That doesn't mean it should be long! It just means you should think carefully about how to write a clear explanation. If you think examples or pictures help, use them! If I don't understand your algorithm, I can't give full credit. (But let me repeat once more: this does not imply that your answers need to be really long.)

The emphasis of this assignment is not just on devising algorithms, but also describing them, testing them, validating them, possibly visualizing them, etc.

Please do all of the written parts in one standalone document for all questions, not as comments in the code.

1. In class we have used the Knapsack problem as a running example. We are given a list of items, each of which has a weight and a value, as well as the capacity of the knapsack, and our goal is to choose the best combination of items whose combined weight is at most the capacity and whose combined value is maximized.

   In this problem we will consider a variant, the *Two Knapsack Problem*. You have two knapsacks, each of which has its own capacity—the two capacities could be the same, or could be different. Your goal is to choose some items to put in the first knapsack, and some items to put in the second knapsack, such that neither knapsack's capacity is exceeded, and the total value of the items in both knapsacks is maximized.

   For this problem, you may wish to use the following function (defined here: `https://docs.python.org/3/library/itertools.html`) that uses the powerful `itertools` package to compute all of the subsets of a set:

```python
import itertools

def powerset(iterable):
    "powerset([1,2,3]) --> () (1,) (2,) (3,) (1,2) (1,3) (2,3) (1,2,3)"
    s = list(iterable)
    return itertools.chain.from_iterable(itertools.combinations(s, r) for r in range(len(s)+1))
```

   (a) Define the search space for this problem. (Note that there could be multiple valid answers.) If there are a total of n items to consider, what is the size of your search space?

   (b) Devise and implement a brute force algorithm to solve this problem. Your code should have a function called `brute_force_two_knapsacks` whose input is: (1) a list of items, each item being a (weight, value) pair, (2) the capacity of the first knapsack, (3) the capacity of the second knapsack. It should return a pair (best_sol, best_score), where best_sol is a list whose first entry is the list of items in the first knapsack and whose second entry is the list of items in the second knapsack, and best_score is the sum of the values of the items in both knapsacks. For example, with

```python
items = [(94, 33), (10, 7), (62, 22), (87, 37), (94, 42), (45, 1), (13, 3), (58, 21),
            (73, 20), (73, 25)]
```

you should be able to run the following:

```
brute_force_two_knapsacks(items, 150, 150)
```

and get output

```
([[(10, 7), (58, 21), (73, 25)], [(62, 22), (87, 37)]], 112)
```

Note that the order of the items in each knapsack and the order of the two knapsacks are irrelevant, so your answer may not look exactly like mine.

For part (b) submit your python code.

(c) Here are functions to generate a list of random items.

```python
import random

def random_item():
    weight = random.randint(10,100)
    value = round(random.random()*weight)
    return (weight, value)

def random_items(n):
    return [random_item() for _ in range(n)]
```

Use these with both capacities set to 150 to test out the speed of your brute force function for various sized inputs. You should probably be able to run 14 items in less than a minute, possibly more depending on your computer and how you implemented it. Use the results of your testing and your answer to part (a) to estimate how long it would take your code to run on 100 items.

2. For this problem, you will implement a backtracking algorithm for the Two Knapsacks Problem, as described in the previous question. Your function should be called `backtracking_two_knapsacks` and it should accept the same arguments and return the same information as in question 1.

   Use your brute force solution from question 1 to test your backtracking algorithm and make sure it's correct.

   (a) For part (a), submit your backtracking code.

   (b) Did testing your results against your brute force solution help you catch any bugs?

   (c) In what situations can you be guaranteed that your backtracking solution will be just as bad as brute force, or possibly even worse?

   (d) What is the most number of items you can run with where you usually finish within a minute?

   (e) Collect some data on how long your runs take for various numbers of items, and form a prediction for how long it would take to run with 100 items. (This is less clear cut than problem 1c, so I am mostly evaluating your ability to gather evidence and make a reasoned estimate.)

3. We saw in class that if you are given a list that you know ahead of time is sorted, then you can search the list for any particular element in $O(\log(n))$ time using a divide-and-conquer approach. Suppose instead that you are given a list L of length n with **distinct entries** that you know has the property that the entries increase for a while, then maybe start to decrease, but then never increase again. For example, the list $[3, 6, 7, 12, 17, 5, 2, 1]$ is such a list, because it increases up to 17, then decreases again. The lists $[1, 5, 8]$ and $[8, 5, 1]$ also count — the first one just doesn't have the decreasing second half and the second one doesn't have the increasing first half.

You goal is to find the largest element of the list. It would be easy to just check all the elements one-by-one, which would take $O(n)$ time. Devise a divide-and-conquer style algorithm to do that has a runtime recurrence of something like

$$T(n) \leqslant T(n/2) + c$$

for a constant c, which implies a runtime of $O(\log(n))$.

(a) Describe your algorithm in writing.

(b) Implement it in a function called `find_max`, which I will call with a list, as in

```
find_max([3, 6, 7, 12, 17, 5, 2, 1])
```

and it should return the value of the largest entry, in this case 17.

(c) How did you test your algorithm to validate its results?

(d) How could you test your algorithm to validate that the runtime is faster than $O(n)$ as claimed?

4. Suppose that there is a stock with ticker symbol MSSC whose price stays constant all day, and only changes between days. Suppose you are given ahead of time $p_0, p_1, \ldots, p_{n-1}$, the price on days 0 through $n-1$. You are going to buy the stock on one day i, and then sell it on a later day j, and your goal is to maximize the amount of money you make. In other words, you want to find the pair of days (i, j) with $i < j$ such that $p_j - p_i$ is maximized.

By simply checking all pairs, you could do this in $O(n^2)$ time. Your goal is to devise a divide-and-conquer algorithm that has a runtime recurrence of something like

$$T(n) \leqslant 2T(n/2) + O(n),$$

which implies a runtime of $O(n \log(n))$.

Describe your algorithm in writing, and then implement it in a function called `mssc`, which I will call with a list of the daily stock prices, as in

```
mssc([4, 10, 1, 3, 7, 2, 5, 8])
```

and it should return the pair of days such that if you buy on the first and sell on the second, your profits are as large as possible (the first day is day 0). In this case, it would return

```
[2, 7]
```