MSSC 6000 - Spring 2023 - Homework 3 Solutions

March 16, 2023

1 Homework 3 Solutions

Please do not distribute these solutions to anyone at any point in the future. If this becomes a problem, I will have to stop providing solutions.

Note that I have not included the full details of the *statement* of each problem. Please refer back to the original assignment for that information.

---

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.

   (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

```
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: `py brute_force_two_knapsacks(items, 150, 150)` and get output `py ([[(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.

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

```
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.

(a) The search space can be naturally defined to be all ways of assigning each item to be either in Knapsack 1, Knapsack 2, or neither. Therefore, there are 3^n elements in the search space. Note that many of these solutions will be invalid because they exceed the capacity, but *this is perfectly fine*. It's very standard for the search space to contain solutions that violate the constraints.

[7]:
```
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))

# item = (weight, value)
def brute_force_two_knapsacks(items, capacity1, capacity2):
    best_score = 0
    best_sol = None

    # loop over all subsets for knapsack 1 possibilities
    for knapsack1 in powerset(items):
        remaining_items = [thing for thing in items if thing not in knapsack1]

        # if knapsack 1 is over capacity, don't bother trying all possibilities
        # for knapsack 2. That would waste a LOT of time!
        if sum(item[0] for item in knapsack1) > capacity1:
            continue

        # loop over all subsets of remaining items for
        # knapsack 2 possibilities
        for knapsack2 in powerset(remaining_items):
            if sum(item[0] for item in knapsack1) <= capacity1:
                if sum(item[0] for item in knapsack2) <= capacity2:
                    score = sum(item[1] for item in knapsack1 + knapsack2)
                    if score > best_score:
                        best_score = round(score)
                        best_sol = (knapsack1, knapsack2)

    return best_sol
```

[8]:
```
test_items_1 = [(33, 6), (81, 17), (89, 24), (16, 9), (81, 4), (91, 36), (72,␣
↪44), (74, 18), (91, 49), (87, 67), (37, 3), (48, 6)]

knapsack1, knapsack2 = brute_force_two_knapsacks(test_items_1, 150, 150)
BF_score = sum(item[1] for item in knapsack1 + knapsack2)

print(f"Knapsack 1: {knapsack1}")
print(f"Knapsack 2: {knapsack2}")
print(f"Score: {BF_score}")
```

```
Knapsack 1: ((72, 44), (74, 18))
Knapsack 2: ((33, 6), (16, 9), (87, 67))
Score: 144
```

(c)

[10]:
```
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)]
```

[14]:
```
from time import time
tt = time()
brute_force_two_knapsacks(random_items(14), 150, 150)
print(f"{round(time()-tt, 2)} seconds.")
```

```
0.87 seconds.
```

The runtime seems quite variable depending on the random items, so in order to observe an average runtime, we will run 50 times for each length from 10 to 14, and use this to extrapolate to 100 items.

[18]:
```
def run_N_times_n_items(N, n):
    runtimes = []
    for _ in range(N):
        tt = time()
        brute_force_two_knapsacks(random_items(n), 150, 150)
        runtimes.append(time()-tt)
    # return (min, mean, max)
    return min(runtimes), sum(runtimes)/len(runtimes), max(runtimes)
```

[19]:
```
run_N_times_n_items(50,10)
```

[19]:
```
(0.012115001678466797, 0.027329540252685545, 0.05758213996887207)
```

[20]:
```
run_N_times_n_items(50,11)
```

[20]:
```
(0.02947402000427246, 0.06707979679107666, 0.11410093307495117)
```

[21]:
```
run_N_times_n_items(50,12)
```

[21]:
```
(0.08384013175964355, 0.16714455604553222, 0.314129114151001)
```

[22]:
```
run_N_times_n_items(50,13)
```

[22]:
```
(0.14128327369689941, 0.3854285192489624, 0.6959939002990723)
```

[23]:
```
run_N_times_n_items(50,14)
```

[23]:
```
(0.4412569999694824, 0.9688764953613281, 3.0447990894317627)
```

[24]:
```
averages = [0.027329540252685545, 0.06707979679107666, 0.16714455604553222, 0.
↪3854285192489624, 0.9688764953613281]
```

[26]:
```
print([averages[i]/averages[i-1] for i in range(1, len(averages))])
```

```
[2.454479518164783, 2.491727226993728, 2.3059591551636713, 2.5137644127872525]
```

It appears (for these small cases at least), like the average runtime increases by a factor of about 2.5 for each new item added. (Why is not 3? Because we are technically not checking everything in the search space—when knapsack 1 is over capacity, we don't check every combination of knapsack 2.

Under this assumption, the average runtime for 100 items would be approximately

$$0.969 \cdot (2.5)^{100-14} \approx 1.6 \cdot 10^{34} \text{ seconds} \approx 5.1 \cdot 10^{26} \text{ years.}$$

For reference, our sun is expected to burn out in about $8 \cdot 10^9$ years.

---

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.

- For part (a), submit your backtracking code.
- Did testing your results against your brute force solution help you catch any bugs?
- In what situations can you be guaranteed that your backtracking solution will be just as bad as brute force, or possibly even worse?
- What is the most number of items you can run with where you usually finish within a minute?
- 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.)

(a)

[34]:
```
# When writing recursive algorithms, always plan out your inputs
#   and outputs very explicitly!
# Input:
#   items: list of *remaining* items to consider putting in
#   capacity1_left: remaining capacity of knapsack 1
#   capacity2_left: remaining capacity of knapsack 2
# Output:
#   best solution using these items, within the remaining capacities

def backtracking_two_knapsacks(items, capacity1_left, capacity2_left):
    # Base case
    if len(items) == 0:
        return (tuple(), tuple())

    # We will consider what to do with items[0]
    weight = items[0][0]
    possible_sols = []

    # can we put it in knapsack 1?
    if weight <= capacity1_left:
        # if it fits in the capacity, then we get a solution with it (item[0])
        #   followed by whatever the best solution is using items[1:] with
        #   the appropriate new capacities
        get_sol = backtracking_two_knapsacks(items[1:], capacity1_left-weight,␣
 ↪capacity2_left)
        # at this point, get_sol does not include items[0] in knapsack 1
        #   so we put it in and record this as a possible solution
        possible_sols.append(((items[0],) + get_sol[0], get_sol[1]))
    if weight <= capacity2_left:
        # similar to previous case
        get_sol = backtracking_two_knapsacks(items[1:], capacity1_left,␣
 ↪capacity2_left-weight)
        possible_sols.append((get_sol[0], (items[0],)+get_sol[1]))

    # third possibility: we don't include the item at all
    get_sol = backtracking_two_knapsacks(items[1:], capacity1_left,␣
 ↪capacity2_left)
    possible_sols.append(get_sol)

    # out of the possibilites in possible_sols, return the one for which
    #  the sum of the values is maximal
    return max(possible_sols, key=lambda sol : sum(item[1] for item in␣
 ↪sol[0]+sol[1]))
```

[38]:
```
# Check solutions match for test_items_1
BF_sol = brute_force_two_knapsacks(test_items_1, 150, 150)
BF_score = sum(item[1] for item in BF_sol[0] + BF_sol[1])
BT_sol = backtracking_two_knapsacks(test_items_1, 150, 150)
BT_score = sum(item[1] for item in BT_sol[0] + BT_sol[1])

print(f"BF sol: {BF_sol}")
print(f"BT sol: {BT_sol}")
print(f"BF score: {BF_score}")
print(f"BT score: {BT_score}")
```

```
BF sol: (((72, 44), (74, 18)), ((33, 6), (16, 9), (87, 67)))
BT sol: (((33, 6), (16, 9), (87, 67)), ((72, 44), (74, 18)))
BF score: 144
BT score: 144
```

[41]:
```
test_items_2 = [(32, 7), (62, 33), (11, 2), (98, 19), (66, 8), (58, 38), (57,␣
 ↪25), (59, 23), (65, 12), (35, 28), (94, 59), (16, 5), (51, 9), (34, 17),␣
 ↪(75, 51), (15, 2), (21, 21), (29, 6), (23, 12), (40, 18)]
BT_sol = backtracking_two_knapsacks(test_items_2, 150, 150)
BT_score = sum(item[1] for item in BT_sol[0] + BT_sol[1])
print(f"BT sol: {BT_sol}")
print(f"BT score: {BT_score}")
```

```
BT sol: (((58, 38), (16, 5), (75, 51)), ((35, 28), (94, 59), (21, 21)))
BT score: 202
```

(b) Yes! Every time I write an algorithm, I write a brute force version to compare against for small inputs. It's always very helpful.

(c) If the capacities are so large every solution in the search space is valid, then backtracking has no benefit. It might even be slower than brute force because all of the recursion adds some degree of overhead.

(d) We want to figure out how many items we can run with to usually finish in around a minute.

[50]:
```
trials = 10
num_items = 30
for num in range(trials):
    items = random_items(num_items)
    tt = time()
    backtracking_two_knapsacks(items, 150, 150)
    print(f"Trial {num+1}: {round(time()-tt,2)} seconds.")
```

```
Trial 1: 170.33 seconds.
Trial 2: 196.62 seconds.
Trial 3: 47.02 seconds.
Trial 4: 26.04 seconds.
Trial 5: 6.16 seconds.
Trial 6: 234.25 seconds.
Trial 7: 28.49 seconds.
Trial 8: 15.01 seconds.
Trial 9: 13.92 seconds.
Trial 10: 40.34 seconds.
```

[51]:
```
L = [170,196,47,26,6,234,28,15,13,40]
sum(L)/len(L)
```

[51]:
```
77.5
```

Clearly there is a LOT of variability, but the average of these times is a little over a minute. To estimate how quickly runtime is increasing, we'll run some trials like in the first question.

[56]:
```
def run_N_times_n_items_backtracking(N, n):
    runtimes = []
    for _ in range(N):
        tt = time()
        backtracking_two_knapsacks(random_items(n), 150, 150)
        runtimes.append(time()-tt)
    return sum(runtimes)/len(runtimes)
```

[68]:
```
runtimes = [run_N_times_n_items_backtracking(50,i) for i in range(15,25)]
```

[77]:
```
# ratios
avgs = [runtimes[i]/runtimes[i-1] for i in range(len(runtimes))]
avgs
```

[77]:
```
[0.01304406325118098,
 1.5984133619682588,
 2.115468032546407,
 1.1478324638400197,
 2.3445834798824015,
 1.9792903671078632,
 2.0384342911006548,
 1.1889763077792834,
 1.0686177540941504,
 1.6434084580694477]
```

[78]:
```
sum(avgs)/len(avgs)
```

[78]:
```
1.5138068579639667
```

We would need more testing to get more reliable data, but let's assume the average runtime is increasing by about 1.5 times with each increase in the number of items. How accurately does this predict the runtime with 30 items that we tested above? Using a ratio of 2 and the calculated average for 24 items, we would predict the average runtime for 30 items to be:

[79]:
```
runtimes[-1] * (1.5)**6
```

[79]:
```
179.6997668802738
```

which overestimates the true answer by about 2.5x. Extrapolating this (somewhat dodgy) formula, we'd get a runtime for 100 items of:

[81]:
```
seconds = runtimes[-1] * (1.5)**(100-24)
years = seconds / 60 / 60 / 24 / 365.25
print(seconds)
print(years)
```

```
381009362440688.2
12073458.13498771
```

about 12 million years.

---

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

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

   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?

(a) Suppose we have a list of n elements with the given property. Since we are asked to come up with a divide-and-conquer algorithm, let's start by splitting into a left half and a right half. We are aiming for a recurrence of the form $T(n) \leq T(n/2) + c$, which tells us that we are only allowed to recursively call our function on one half, not both halves.

Let's compare the last element of the left list with the first element of the right list, which we'll call a and b, respectively. If $a < b$, then the largest element of our list must be in the right half. If $a > b$, then the largest element of our list must be in the left half. So, if we know which half the max is in, then we only need to call the function recursively on that half.

The base case is when we have a list with only one element, which cannot be split in half. In that case, the single element is the max, so we return that.

(b)

[193]:
```
import math

def find_max(L):
    if len(L) == 1:
        return L[0]

    middle_index = math.floor(len(L) / 2)
    end_of_left = L[middle_index - 1]
    start_of_right = L[middle_index]

    if end_of_left > start_of_right:
        return find_max(L[:middle_index])
    else:
        return find_max(L[middle_index:])
```

[147]:
```

```

(c) The way that I tested my code ***and the way that I graded your assignments*** was by writing a function to generate random lists with the given property, and testing the divide-and-conquer algorithm against python's built-in `max` function.

[91]:
```
# Generate a random list of length n with the property that
#  entries increase for a while, then decrease
def test_list(n):
    # choose the index in the list where the decrease will start
    decrease_index = random.randint(0, n-1)
    start = random.randint(1,10)
    unimodal = [start]
    while len(unimodal) < n:
        if len(unimodal) < decrease_index:
            # add an element a bit bigger than the last one
            unimodal.append(random.randint(unimodal[-1]+1,unimodal[-1]+10))
        else:
            # add an element a bit smaller than the last one
            unimodal.append(random.randint(unimodal[-1]-10,unimodal[-1]-1))
    return unimodal
```

```
[146]: # test 1000 times for lists of each length from 5 to 100
       # takes a few seconds
       for list_length in range(5, 101):
           for _ in range(1000):
               random_list = test_list(list_length)
               assert find_max_non_recursive(random_list) == max(random_list)
```

(d) A good way to see if this runs faster than $O(n)$ is to run it for different large lengths and check if the growth appears linear.

```
[189]: # run N times with a list of length n and return average runtime
       from time import time
       def run_time_n_items_N_runs(N, n):
           total_time = 0
           for _ in range(N):
               L = test_list(n)
               tt = time()
               find_max(L)
               total_time += time() - tt
           return total_time
```

If $f(n)$ is a logarithmic function, then as n grows exponentially, $f(n)$ should grow linearly. So, we will test the values $f(n)$, $f(2n)$, $f(4n)$, $f(8n)$ (doubling every time) and check that those values *do not double each time*, but instead grow linearly.

```
[201]: run_time_n_items_N_runs(100, 10000)
```

```
[201]: 0.003577709197998047
```

```
[202]: run_time_n_items_N_runs(100, 20000)
```

```
[202]: 0.006951332092285156
```

```
[205]: run_time_n_items_N_runs(100, 40000)
```

```
[205]: 0.014272689819335938
```

```
[206]: run_time_n_items_N_runs(100, 80000)
```

```
[206]: 0.03482389450073242
```

```
[207]: run_time_n_items_N_runs(100, 160000)
```

```
[207]: 0.0707857608795166
```

Uh oh!!! When we double the length of the lists, the time it takes to run also doubles! This tells us that our algorithm is NOT $O(\log n)$, but probably $O(n)$. What went wrong? If you read our `find_max` function again, it certainly looks like its runtime is roughly $T(n) = T(n/2) + c$, which guarantees $O(\log n)$, so... what happened? Spend a minute looking at the function to see if you can figure it out.

The problem is that once we decide which half of the list to look in, we call the recursive function again, *making a full copy of half the list in memory*:

```
if end_of_left > start_of_right:
    return find_max(L[:middle_index])
else:
    return find_max(L[middle_index:])
```

Copying a list of length L requires L operations, so doing this to the half of the list of length n means the time is more like $T(n) = T(n/2) + O(n)$, which gives a total runtime of $O(n)$. Alas, recursion bites us.

The way to accomplish this without recursion is by keeping the list as it is, and just repeatedly changing the `start_index` and `end_index` and pretending the list is just numbers within that range.

[208]:
```
import math

def find_max_non_recursive(L):

    start_index = 0
    end_index = len(L)-1

    while start_index != end_index:

        middle_index = start_index + math.ceil((end_index - start_index)/2)

        end_of_left = L[middle_index - 1]
        start_of_right = L[middle_index]

        if end_of_left > start_of_right:
            end_index = middle_index - 1
        else:
            start_index = middle_index

    return L[start_index]
```

[216]:
```
# run N times with a list of length n and return average runtime
from time import time
def run_time_n_items_N_runs_non_recursive(N, n):
    total_time = 0
    for _ in range(N):
        L = test_list(n)
        tt = time()
        find_max_non_recursive(L)
        total_time += time() - tt
    return total_time
```

[217]:
```
run_time_n_items_N_runs_non_recursive(100, 10000)
```

[217]:
```
0.0006022453308105469
```

[221]:
```
run_time_n_items_N_runs_non_recursive(100, 20000)
```

[221]:
```
0.0008115768432617188
```

[220]:
```
run_time_n_items_N_runs_non_recursive(100, 40000)
```

[220]:
```
0.001132014135742188
```

[222]:
```
run_time_n_items_N_runs_non_recursive(100, 80000)
```

[222]:
```
0.0012569427490234375
```

[223]:
```
run_time_n_items_N_runs_non_recursive(100, 160000)
```

[223]:
```
0.0013203620910644531
```

This time, we are doubling the input and the times are not doubling! They seem to be increasing linear (a constant different each time). We did it!

---

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 $p_0, p_1, \dots, p_{n-1}$, 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

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

   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]

Answer: We start by splitting our input list into a left half and a right half, since we are asked to find a divide-and-conquer algorithm. As we're aiming for a recurrence with a $2T(n/2)$ term, we are allowed to recursively call our function on both halves.

We're looking for the best two days, one to buy and a later one to sell. There are three possibilities for the best combination: both days could be in the left half, both days could be in the right half, or the days could be split. So, the "divide" step finds the best pair in the left and the best pair in the right, and then in the "conquer" step we need to find the best split pair. We can do this by just finding the smallest day in the left and the largest day in the right with the `min` and `max` functions (which add $O(n)$ time to the "conquer" step, which is allowed).

First, a simplified version that returns the best dollar amount instead of the indices of the best days:

[224]:
```python
def mssc_simplified(L):

    if len(L) == 1:
        return 0

    middle_index = math.floor(len(L) / 2)
    left_half = L[:middle_index]
    right_half = L[middle_index:]

    best_left_score = mssc(left_half)
    best_right_score = mssc(right_half)
    best_mix_score = max(right_half) - min(left_half)

    return max(best_left_score, best_right_score, best_mix_score)
```

Now the full version:

[230]:
```python
def mssc(L):

    assert len(L) >= 2

    if len(L) == 2:
        return [0, 1]
    elif len(L) == 3:
        # return which of the three possible buy/sell combos has the
        #   biggest difference
        return max([[0, 1], [0, 2], [1, 2]], key=lambda x: L[x[1]] - L[x[0]])

    middle_index = (len(L) - 1) // 2
    left_half = L[:middle_index + 1]
    right_half = L[middle_index + 1:]

    best_left_index = mssc(left_half)
    best_right_index = mssc(right_half)
    best_right_index[0] += middle_index + 1
    best_right_index[1] += middle_index + 1

    # Another slick use of lambda functions to get the *index* of
    # the max and min values. This is a good trick to remember!
    best_mix_index = [
        min(range(middle_index+1), key=lambda j : L[j]),
        max(range(middle_index+1,len(L)), key=lambda j : L[j]),
    ]

    return max([best_left_index, best_right_index, best_mix_index], key=lambda␣
↪x: L[x[1]]-L[x[0]])
```

As with other parts, I tested this by writing a brute force function and comparing answers for randomly generated problems. This is also how I tested your answers and found most errors.

[231]:
```
import itertools

def brute_force(L):
    indices = itertools.combinations(range(len(L)), 2)
    return list(max(indices, key=lambda x: L[x[1]]-L[x[0]]))

def test_prices(n):
    return [random.randint(1,10*n) for i in range(n)]
```

[235]:
```
for n in range(5, 10):
    for _ in range(1000):
        L = test_prices(n)
        assert brute_force(L) == mssc(L), (L, brute_force(L), mssc(L))
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Input In [235], in <module>
      2 for _ in range(1000):
      3     L = test_prices(n)
----> 4     assert brute_force(L) == mssc(L), (L, brute_force(L), mssc(L))

AssertionError: ([30, 4, 9, 4, 11], [1, 4], [3, 4])
```

Uh oh, did we mess up? No. The brute force function says to buy on day 1 and sell on day 4 for a profit of 7. The divide-and-conquer function says to buy on day 3 and sell on day 4, also for a profit of 7. These are two different solutions, but both optimal. So when checking correctness, we should only check that they have the same value.

[237]:
```
for n in range(5, 10):
    for _ in range(1000):
        L = test_prices(n)
        bf_sol = brute_force(L)
        dc_sol = mssc(L)
        assert L[bf_sol[1]] - L[bf_sol[0]] == L[dc_sol[1]] - L[dc_sol[0]]
```

[ ]:
```

```