CS 170 HW 8

Due **2021-03-16, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, write "none".

DP solution writing guidelines:

Try to follow the following 3-part template when writing your solutions.

- Define a function $f(\cdot)$ in words, including how many parameters are and what they mean, and tell us what inputs you feed into f to get the answer to your problem.
- Write the "base cases" along with a recurrence relation for f.
- Prove that the recurrence correctly solves the problem.
- Analyze the runtime and space complexity of your final DP algorithm.

2 DP Coding

This semester, we are trying something new: questions which involve coding.

(a) In this question, we consider the maximum subarray sum problem. This is a classic dynamic programming problem and also happens to be asked somewhat frequently during coding interviews.

The setup of the problem is as follows: You are given an array A of numbers. You want to find the subarray sum of the subarray, i.e. slice of the array, with the largest sum. In other words, you want to find

$$\max_{i,j} \sum_{k=i}^{j} A[k].$$

For example, the array $[-1, 2, 6, -3, 5, -6, 3]$ has a maximum subarray sum of 10, which is achieved by the subarray $[2, 6, -3, 5]$.

Describe and analyze a dynamic programming algorithm to solve the Maximum Subarray Sum problem.

Solution: We will have n subproblems, which we call f; subproblem $f(j)$ will be the maximum subarray sum for subarrays ending with exactly index j; that is, $f(j) = \max_i \sum_{k=i}^{j} A[k]$.

Base case: There is only one subarray ending with exactly index 0, which contains only $A[0]$ and so has sum $A[0]$.

Recurrence relation: The subarray ending with index j contains $A[j]$ and potentially some other earlier elements; that is, $\max_i \sum_{k=i}^{j} A[k] = \max(\max_i \sum_{k=i}^{j-1} A[k], 0) + A[j]$. We can replace the sum in the second expression with $f(j-1)$, by definition of f. Therefore, our recurrence relation will be $f(j) = max(f(j-1), 0) + A[j]$.

Then, since we would like to recover the maximum subarray sum, and we know every subarray must end at some index, we can return our final answer as $\max_i f(i)$.

Note that above we have both described and proven the correctness of our algorithm. We have n subproblems, and the recurrence relation takes $O(1)$ time for each, so our algorithm has runtime $O(n)$. As described, it takes $O(n)$ space, but this can be reduced to $O(1)$.

(b) This link will take you to a python notebook, hosted on the Berkeley datahub, in which you will implement your dynamic programming algorithm for the problem above, so you can see what it looks like in actual code. Once you have finished, download a PDF of your completed notebook via File → Download as → PDF via HTML, and append the downloaded pdf to the rest of your homework submission. Be careful when selecting pages on gradescope.

Note: Datahub does not guarantee 100% reliability when you save your notebook, and recommends downloading a local copy occasionally to backup progress (via File → Download as → Notebook (.ipynb)).

3 Spaceship

A spaceship uses some *oxidizer* units that produce oxygen for three different compartments. However, these units have some failure probabilities.

Because of differing requirements for the three compartments, the units needed for each have somewhat different characteristics.

A decision must now be made on just *how many* units to provide for each compartment, taking into account design limitations on the *total* amount of *space*, *weight* and *cost* that can be allocated to these units for the entire ship. Specifically, the total space for all units in the spaceship should not exceed 500 cubic inches, the total weight should not exceed 200 lbs and the total cost should not exceed 400,000 dollars.

The following table summarizes the characteristics of units for each compartment and also the total limitation:

	Space (cu in.)	Weight (lb)	Cost ($)	Probability of failure
Units for compartment 1	40	15	30,000	0.30
Units for compartment 2	50	20	35,000	0.40
Units for compartment 3	30	10	25,000	0.20
Limitation	500	200	400,000	

The objective is to *minimize the probability* of all units failing in all three compartments, subject to the above limitations and the further restriction that each compartment have a probability of no more than 0.05 that all its units fail.

Formulate the *integer programming model* for this problem. An integer programming model is the same as a linear programming model with the added functionality that variables can be forced to be integers.

Side note: Integer programming is often intractable, so we use a linear program as a heuristic. We can take out the integrality constraints and change our model into a linear program. We then round the solution and make sure none of constraints have been violated (you should think about why this won't always give us the optimum)

Solution: Let x_1 be the number of oxidizer units provided for compartment 1, and define similarly x_2 for compartment 2 and x_3 for compartment 3. The probability that all units fail in compartment 1 is $(0.3)^{x_1}$, for compartment 2 is $(0.4)^{x_2}$ and for compartment 3 is $(0.2)^{x_3}$. The probability that all units fail in all compartments is $(0.3)^{x_1}(0.4)^{x_2}(0.2)^{x_3}$. The exponential function is non-linear, so we take logarithm (which preserves ordering of numbers) to get the linear programme

$$\text{Minimize } \log(0.3)x_1 + \log(0.4)x_2 + \log(0.2)x_3$$

$$\text{subject to } \begin{cases} 40x_1 + 50x_2 + 30x_3 \leq 500 & \text{space constraint (cu in.)} \\ 15x_1 + 20x_2 + 10x_3 \leq 200 & \text{weight constraint (lb)} \\ 30x_1 + 35x_2 + 25x_3 \leq 400 & \text{cost constraint (\$1000)} \\ \log(0.3)x_1, \log(0.4)x_2, \log(0.2)x_3 \leq \log(0.05) & \text{reliability constraints (log scale)} \end{cases}.$$

(Note that the reliability constraints imply the non-negativity constraints that $x_1, x_2, x_3 \geq 0$.)

4 Motel Choosing (optional)

You may submit your solution to this problem if you wish it to be graded, but it will be worth no points.

You are traveling along a long road, and you start at location $r_0 = 0$. Along this road, there are n motels at location $\{r_i\}_{i=1}^n$ with $0 < r_1 < r_2 < \cdots < r_n$. The only places you may stop are these motels, but you can choose which to stop at. You must stop at the final motel (at distance r_n), which is your destination.

Ideally, you want to travel exactly T miles a day and stop at a motel at the end of the day, but this may not be possible (depending on the spacing of the motels). Instead, you receive a *penalty* of $(T - x)^8$ each day, if you travel x miles during the day. The goal is to plan your stops to minimize the total penalty (over all travel days).

Describe and analyze an algorithm that outputs the minimum penalty, given the locations $\{r_i\}$ of the motels and the value of T.

Solution: Let $P[i]$ be the optimal (*i.e.* minimum) penalty for getting to hotel i; it holds that

$$P[i] = \min_{j<i} \left\{ P[j] + (T - (r_i - r_j))^8 \right\}.$$

The base case is given by $P[0] = 0$. We can compute the recurrence as follows.

Input: $\{r_i\}, T$
$P[0] := 0$
for $i = 1$ to n:
    $P[i] = 0$
    for $j = 1$ to $i - 1$:
        if $P[i] > P[j] + (T - (r_i - r_j))^8$
            $P[i] = P[j] + (T - (r_i - r_j))^8$
Output: $P[n]$

The running time is $O(n^2)$. To prove correctness, note that the base case is trivial. Assume as an inductive hypothesis that $P[j]$ are computed correctly for all $j < n$. Then to finally get to the destination at location r_n, there has to be a penultimate stop is i along our route. Then each choice of the stop i lead to a penalty of $P[i] + \left(T - (r_i - r_n)^8\right)$. We assumed that the first term is the minimum penalty to get to i, and the second term is the necessary penalty from i to n. Therefore, minimizing this quantity over all i leads to the minimum penalty of the entire trip.

DP Coding Assignment: Maximum Subarray Sum

In [1]:
```
import random
import time
import matplotlib.pyplot as plt
```

Part (b): Implementation

In the following cell, implement your dynamic programming algorithm from part (a).

In [2]:
```
def max_subarray_sum(A):
    # Initializing f. Recall f[i] contains the maximum sum of all subarrays ending with item i.
    f = [None for _ in range(len(A))]

    # Base Case
    f[0] = A[0]

    for i in range(1, len(f)):
        f[i] = max(f[i - 1], 0) + A[i]

    return max(f)
```

Now, let's test your implementation. Run the two cells below and check that your algorithm's output matches the naive algorithm's output on some random inputs.

In [3]:
```
def max_subarray_sum_naive(A):
    maxSum = 0
    n = len(A)
    for i in range(n):
        for j in range(i, n):
            maxSum = max(maxSum, sum(A[i:j+1]))
    return maxSum
```

In [4]:
```
for i in range(10):
    A = [random.uniform(-1000, 1000) for i in range(500)]
    print('Optimized Answer: {0}, Naive Answer: {1}'.format(max_subarray_sum(A), max_subarray_sum_naive(A)))
    assert max_subarray_sum(A) == max_subarray_sum_naive(A)

print('Test passed')
```

```
Optimized Answer: 7102.062586363136, Naive Answer: 7102.062586363136
Optimized Answer: 18802.530069811884, Naive Answer: 18802.530069811884
Optimized Answer: 18261.376661851777, Naive Answer: 18261.376661851777
Optimized Answer: 14495.821646541244, Naive Answer: 14495.821646541244
Optimized Answer: 33658.868348374715, Naive Answer: 33658.868348374715
Optimized Answer: 21119.904490421333, Naive Answer: 21119.904490421333
Optimized Answer: 8280.948495644903, Naive Answer: 8280.948495644903
Optimized Answer: 11636.444275877351, Naive Answer: 11636.444275877351
Optimized Answer: 10525.326768102477, Naive Answer: 10525.326768102477
Optimized Answer: 17214.193339852132, Naive Answer: 17214.193339852132
Test passed
```

Compare your runtime from part (a) to the runtime of the naive algorithm.

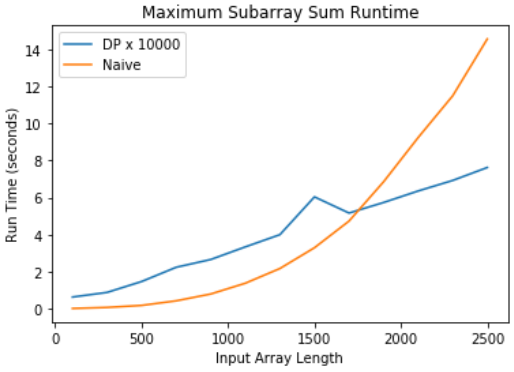
The dynamic programming algorithm runs in $O(N)$, whereas the naive algorithm runs in $O(N^3)$ (here N is the length of A).

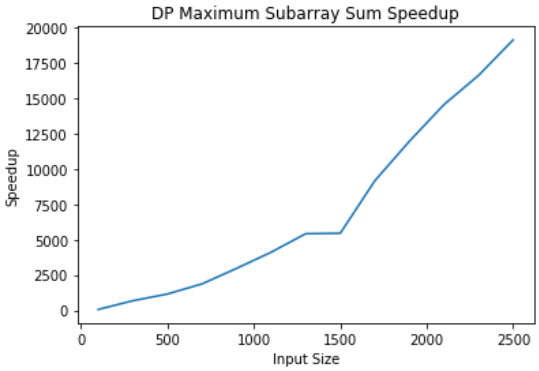
Run the following cell to compare the runtimes of your DP algorithm and the naive algorithm. Check that the runtime and speedup graphs have the asymptotic behavior that you expected in your answer above (don't be worried if the graphs are a bit unsmooth as long as the overall trend is OK).

In [5]:
```
def record(array, value, name):
    array.append(value)
    print("%s%f" % (name, value))

dp_times = []
naive_times = []
speed_ups = []

input_lengths = range(100, 2700, 200)

for n in input_lengths:
    print("\narray length: %d" % n)
    A = [random.uniform(-1000, 1000) for i in range(n)]
    time1 = time.time()
    dp_res = max_subarray_sum(A)
    time2 = time.time()
    dp_time = time2 - time1
    record(dp_times, dp_time, "DP time:   ")
    naive_res = max_subarray_sum_naive(A)
    time3 = time.time()
    naive_time = time3 - time2
    record(naive_times, naive_time, "naive time: ")
    assert dp_res == naive_res
    speed_up = naive_time / dp_time
    record(speed_ups, speed_up, "speed up: ")

plt.plot(input_lengths, [t * 10000 for t in dp_times], label="DP x 10000")
plt.plot(input_lengths, naive_times, label="Naive")
plt.xlabel("Input Array Length")
plt.ylabel("Run Time (seconds)")
plt.legend(loc="upper left")
plt.title("Maximum Subarray Sum Runtime")

plt.figure()
plt.plot(input_lengths, speed_ups)
plt.xlabel("Input Size")
plt.ylabel("Speedup")
plt.title("DP Maximum Subarray Sum Speedup")
```

```
array length: 100
DP time:   0.000062
naive time: 0.003676
speed up: 59.300000

array length: 300
DP time:   0.000087
naive time: 0.058725
speed up: 674.821918

array length: 500
DP time:   0.000146
naive time: 0.167998
speed up: 1151.364379

array length: 700
DP time:   0.000223
naive time: 0.415720
speed up: 1864.872727

array length: 900
DP time:   0.000265
naive time: 0.784456
speed up: 2961.518452

array length: 1100
DP time:   0.000333
naive time: 1.366928
speed up: 4104.016464

array length: 1300
DP time:   0.000399
naive time: 2.161401
speed up: 5415.514934

array length: 1500
DP time:   0.000603
naive time: 3.284719
speed up: 5445.497628

array length: 1700
DP time:   0.000516
naive time: 4.725008
speed up: 9158.096118

array length: 1900
DP time:   0.000573
naive time: 6.848706
speed up: 11954.039118

array length: 2100
DP time:   0.000635
naive time: 9.238458
speed up: 14545.383258

array length: 2300
DP time:   0.000692
naive time: 11.495904
speed up: 16609.478471

array length: 2500
DP time:   0.000762
naive time: 14.566455
speed up: 19116.439299
```

Out[5]: Text(0.5,1,'DP Maximum Subarray Sum Speedup')




DP Maximum Subarray Sum Speedup
Speedup
Input Size
20000
17500
15000
12500
10000
7500
5000
2500
0
0
500
1000
1500
2000
2500