Traveling Example

You have 8 gallons of gas, and need to travel as far as you can.

You can change speed every hour if you want, but when your 8 gallons is gone, you’re done.

<table>
<thead>
<tr>
<th>mph</th>
<th>8</th>
<th>50</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>170</th>
<th>170</th>
<th>196</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg</td>
<td>8</td>
<td>25</td>
<td>27</td>
<td>22.5</td>
<td>20</td>
<td>28</td>
<td>24</td>
<td>24.5</td>
</tr>
</tbody>
</table>
**First** question – what are we trying to “optimize”?

Maximize the distance we go.

**Second** question – what data are we given?
Starting number of gallons, miles/hour, and miles/gallon

**Third** question – what data do we need?
How many gallons we use in an hour, at each speed
We essentially have the data in “miles” and need it in “gallons”

<table>
<thead>
<tr>
<th>mph→</th>
<th>8</th>
<th>50</th>
<th>80</th>
<th>90</th>
<th>100</th>
<th>170</th>
<th>170</th>
<th>196</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpg→</td>
<td>8</td>
<td>25</td>
<td>27</td>
<td>22.5</td>
<td>20</td>
<td>28</td>
<td>24</td>
<td>24.5</td>
</tr>
</tbody>
</table>

We can change speed every hour, so use mph to get how far in 1 hr and then divide by mpg to get number of gallons used. (Round to closest integer.)

<table>
<thead>
<tr>
<th>gals→</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>miles→</td>
<td>8</td>
<td>50</td>
<td>80</td>
<td>90</td>
<td>100</td>
<td>170</td>
<td>170</td>
<td>196</td>
</tr>
</tbody>
</table>
What problem can we think about that can be divided into smaller instances of the same problem?

How about choosing the number of gallons to use?

We could decide how many gallons to use, then decide how many gallons to use of what is left, ....

We can do this till there is 1 gallon left, and that’s the smallest we can choose so we’re done.
choose x

10 gal

10 - x gal

... → 1 gal

and we go 8 miles

<table>
<thead>
<tr>
<th>gals →</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>d[()] →</td>
<td>8</td>
<td>50</td>
<td>80</td>
<td>90</td>
<td>100</td>
<td>170</td>
<td>170</td>
<td>196</td>
</tr>
</tbody>
</table>
getDist(d, n)
for i = 1..n
    distSoFar = max(distSoFar, d[i] + getDist(d, n-i))

\[ gD(d,n) = \begin{cases} 
8 & n = 1 \\
\sum_{i=1}^{n-1} \max(\text{distSoFar}, d[i] + gD(n-i)) & n > 1 
\end{cases} \]
Recursive calls: 1

Recursive calls: 2

Recursive calls: 4

See a pattern? Every time we add a node, we double the work: $2^{(n-1)}$
If it takes $T(n)$ amount of work to figure out this problem for $n$ nodes, and we know that we double the amount of work for every node we add, then we get this series:

$$
1T(n) = 2T(n-1) = 4T(n-2) = 8T(n-3) = 2^{i-1}T(n-i) = 2^{n-1}T(n-n)
$$

From 0 up to $n-1$ in this $n-1..n-n$ series we get a summation term which we know will be order $n^2$. However, the exponential term completely dominates the complexity.
Clearly this isn’t really acceptable for complexity.

So now we’ll try a dynamic programming version. You can try a memorizing top-down version for practice if you like.

The next series of slides uses a bottom-up dynamic programming algorithm.
Bottom-up Dynamic Prog

Calculate 2 arrays:

• Maximum distance: call this md[]
• How we got the max distance: call this path[]

Input: the total number of gallons (8) and the distance array, d[]

Two nested loops:

• Nodes from 1..n: call this iterator j
• Subnodes from 1..j: call this iterator i
for j = 1 .. n
    maxD = -1
    for i = 1 .. j
        newMax = max (maxD, d[i] + md[j - i])
        if newMax > maxD
            path[j] = i
            maxD = newMax
    md[j] = maxD

md[n] is the maximum distance we can go.
If (path[n] ≠ n), then we didn’t choose n.
We chose (n − path[n]), call it choice1. Now check path[choice1]. If it isn’t choice1, then we also took (choice1 − path[choice1]), and so on till we get to 0.
Work for the DP algo

We go through the outer loop $n$ times, and we go through the inner loop once, then twice, then 3 times, and so on up to $n$ times.

Iterate from 1 up to the current value of $j$ in $j=1..n$, so more work gets done the closer $j$ gets to $n$, and the total work is approximately $n^2/2$. 

![Graph showing work as a function of $n$.]
Try coding this up in Python 3. Hint: Python arrays are 0-based, and arrays in this solution are 1-based, but you need a value for 0 since the algorithm accesses \( md[j - i] \) which will evaluate to 0 when \( i=j \).

Input: \( n = 8, d = [0, 8, 50, 80, 90, 100, 170, 170, 196] \)
Calculate: \( md = [0, 8, 50, 80, 100, 130, 170, 180, 220], \)
\[ \text{path} = [0, 1, 2, 3, 2, 2, 6, 2, 2] \]
How far can we go? What are the choices that give us this maximum?