In [277…

```
# setup
from IPython.core.display import display,HTML
display(HTML('<style>.prompt{width: 0px; min-width: 0px; visibility: collapse}</style>'))
display(HTML(open('rise.css').read()))

# imports
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
sns.set(style="whitegrid", font_scale=1.5, rc={'figure.figsize':(12, 6)})
```

CMPS 2200

Introduction to Algorithms

Dynamic Programming

Today's agenda:

- "Dynamic Programming" Paradigm

So far, we've looked at 3 different paradigms of algorithm design:

- Divide and Conquer
- Brute Force
- Greedy

(Randomization can be added to any of these.)

Divide and Conquer and Greedy both had a restriction to them in that we always recurse on one or more solutions to *fixed size* subproblems.

In certain cases, for example using a greedy approach for the Knapsack Problem, this prevented us from reaching an optimal solution.

What if we generalized our approach to consider **all possible** subproblems? This is sort of a hybrid of brute force and Greedy/Divide and Conquer.

0-1 Knapsack

Suppose there are n objects, each with a *value* v_i and *weight* w_i. You have a "knapsack" of capacity W and want to fill it with a set of objects $X \subseteq [n]$ so that $w(X) \leq W$ and $v(X)$ is maximized.

We saw previously that a greedy approach works only if we're allowed to take fractional parts of the objects. The problem was that a greedy approach didn't really take leftover capacity into account.

Recall that for greedy algorithms to be correct, we needed to prove that the greedy choice combined with an optimal solution for a smaller subproblem was also optimal (the **optimal substructure** property).

Fractional Knapsack had this property, but 0-1 Knapsack did not. Why?

We can give a simple counterexample with 2 objects that have weights/values $(10, 5), (9.999, 3)$ with $W = 5$.

index	value	weight
0	10	5
1	9.999	3

The problem is that the greedy choice to maximize value/weight is incorrect because we can have leftover capacity. We really need to look at *all possible* choices of objects and their associated optimal solutions.

Let $OPT(S, W)$ be an optimal solution to the Knapsack problem for a set of objects S and capacity W. We started with n objects and capacity W so we are interested in finding $OPT([n], W)$.

Now, we can make the following simple observation: if object n is in the optimal solution, then $OPT([n], W) = \{n\} \cup OPT([n-1], W - w(n))$. If it isn't, then $OPT([n], W) = OPT([n-1], W)$.

Optimal Substructure for Knapsack: For any set of objects $[n]$ and $W > 0$, we have

$$v(OPT(n, W)) = \max\{v(n) + v(OPT([n-1], W - w(n))), \\ v(OPT(n-1, W))\}.$$

In a way, this really isn't saying much. Put plainly we're just saying that the optimal solution either contains object n or it doesn't.

For the example above, we have that:

$$\begin{aligned} v(OPT(1, 5)) &= \max\{v(1) + v(OPT(0, 2)), v(OPT(0, 5))\} \\ &= \max\{10, 9.999\} \\ &= 10. \end{aligned}$$

Does this give us an algorithm? It is easy to write this in SPARC:

```
knapsack n W =
  if n = 0
    0
  if n = 1
    if w(1) ≤ W
      v(1)
    else 0
  else
    if w(n) ≥ W
      knapsack n − 1 W
    else
      max{v(i) + knapsack n − 1 W − w(n), knapsack n − 1 W}
```

We can see that our optimal substructure recurrence depends both on the number of objects as well as their weights.

The number of recursive calls doubles in every recursion. But suppose all items have weight 1 - is there a glaring inefficiency we can fix? Let's consider the recursion tree:

If we blindly recompute the redundant calls when they are encountered, then we will do $\Omega(2^n)$ work and $O(n)$ span even if we can take all the items.

However, suppose that whenever we need to compute $v(OPT(i, w)$, we compute it once and save the result for later use (e.g., in a suitable data structure) -- this is called *memoization*. Then, we no longer have a binary tree but rather a **directed acyclic graph** or **DAG**.

The number of nodes in this DAG will allow us to determine the work of this algorithm, and the longest path in the DAG will allow us to determine the span.

When performing memoization, we can either proceed **top-down** or **bottom-up**:

- **top-down**: use recursion solution as usual, but maintain a hashmap or related data structure to quickly lookup solutions previously computed
- **bottom-up**: start with solutions to smallest problem instances, then proceed to larger instances. This is typically implemented by filling a table.

index	value	weight
0	10	5
1	6	3
2	6	2

Optimal solution is 12 (second and third items)

Consider this table: number of items to include (rows) by weight (cols)

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	10
2	0	0	0	6	6	10
3	0	0	6	6	6	**12**

In [5]:

```
import random

def recursive_knapsack(objects, i, W):
    v, w = objects[i]
    if (i == 0):
        if (w <= W):
            return(v)
        else:
            return(0)
    else:
        if (w <= W):
            take = v + recursive_knapsack(objects, i-1, max(W-w, 0))
```

```
            dont_take = recursive_knapsack(objects, i-1, W)
            return(max(take, dont_take))
        elif (W == 0):
            return(0)
        else:
            return(recursive_knapsack(objects, i-1, W))

def tabular_knapsack(objects, W):
    n = len(objects)
    # we'll rely on indices to also represent weights, so we'll index from 1...W
    # in the weight dimension of the table
    OPT = [[0]*(W+1)]

    #print(objects[0][1])
    # initialize the first row of the table
    for w in range(W+1):
        if (objects[0][1] <= w):
            OPT[0][w] = objects[0][0]
        else:
            OPT[0][w] = 0

    # use the optimal substructure property to compute increasingly larger solutions
    for i in range(1,n):
        OPT.append([0]*(W+1))
        v_i, w_i = objects[i]
        for w in range(W+1):
            if (w_i <= w):
                OPT[i][w] = max(v_i + OPT[i-1][w-w_i], OPT[i-1][w])
            else:
                OPT[i][w] = OPT[i-1][w]

    print(OPT)
    return(OPT[n-1][W])

W = 3
objects = [(10,1), (6,3), (6,2)]
print(tabular_knapsack(objects, W))


#W = 5
#objects = [(10, 5), (9.999, 3)]
#n = len(objects)-1
#print(recursive_knapsack(objects.copy(), n, W))
#print(tabular_knapsack(objects.copy(), W))


#W = 1000
#objects = [(i, i) for i in range(1, 500000)]
#n = len(objects)-1
#print(recursive_knapsack(objects, n, W))
#print(tabular_knapsack(objects, W))
```

```
[[0, 10, 10, 10], [0, 10, 10, 10], [0, 10, 10, 16]]
16
```

There are at most $O(nW)$ nodes in this DAG, and the longest path is $O(n)$. Each node requires $O(1)$ work/span, so the work is $O(nW)$ and the span is $O(n)$. Is this efficient?

Well, not really. We need exponential time in the input size of W. We get only $\log_2 W$ bits as input, but take time proportional to W.

This isn't completely satisfying, but if it makes you feel better we do not know of any efficient algorithm for the 0-1 Knapsack problem (more on this at the end of the semester).

Elements of Dynamic Programming

This is what we call **dynamic programming**. The elements of a dynamic programming algorithm are:

- Optimal Substructure
- Recursion DAG

The correctness of the dynamic programming approach follows from the optimal substructure property (i.e., induction). If we can prove that the optimal substructure property holds, and that we compute a solution by correctly implementing this property then our solution is optimal.

As with divide and conquer algorithms, achieving a good work/span can be tricky. We can minimize redundant computation by memoizing solutions to all subproblems. This can be done *top-down* by saving the result of a recursive call the first time we encounter it. Or, we can compute the optimal substructure property *bottom-up* by starting with the base case(s) and working our way up.

Can we derive the number of nodes in the DAG using the optimal substructure property?

Work and Span in Dynamic Programming

Since we memoize the solution to every distinct subproblem, the number of nodes in the DAG is equal to the number of distinct subproblems considered.

The longest path in the DAG represents the span of our dynamic programming algorithm.

Why "Dynamic Programming"?

The mathematician Richard Bellman coined the term "dynamic programming" to describe the recursive approach we just showed. The optimal substructure property is sometimes referred to as a "Bellman equation." But why did he call it dynamic programming?

There is some folklore around the exact reason. But it could possibly be because "dynamic" is a really dramatic way to describe the search weaving through the DAG. The term "programming" was used in the field of optimization in the 1950s to describe an optimization approach (e.g., linear programming, quadratic programming, mathematical programming).