Intractable Problems

Part Two

Announcements

- Problem Set Five graded; will be returned at the end of lecture.
- Extra office hours today after lecture from 4PM – 6PM in Clark S250.
- Reminder: Final project goes out on Monday; we recommend *not* using a late day on Problem Set Six unless necessary.

Please evaluate this course on Axess.

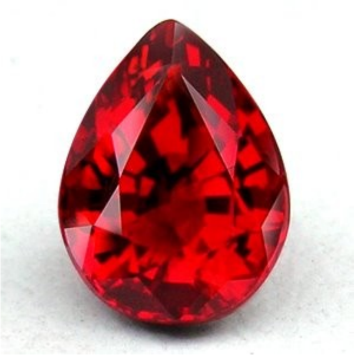
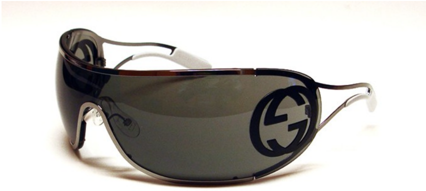
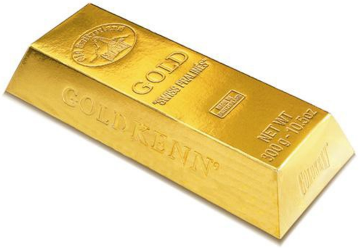
Your feedback really makes a difference.

Outline for Today

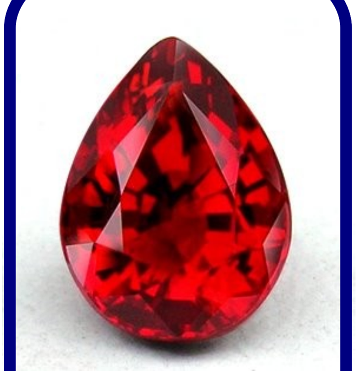
- **0/1 Knapsack**
  - An **NP**-hard problem that isn't as hard as it might seem.
- **Fixed-Parameter Tractability**
  - What's the *real* source of hardness in an **NP**-hard problem?
- **Finding Long Paths**
  - A use case for fixed-parameter tractability.

The 0/1 Knapsack Problem

The 0/1 Knapsack Problem

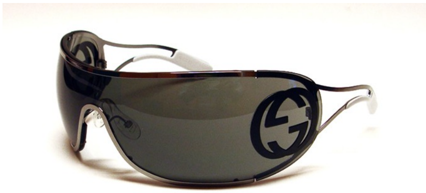
				
$500	$200	$900	$1,500	$400
45g	20g	53g	25g	35g

The 0/1 Knapsack Problem



$500

45g

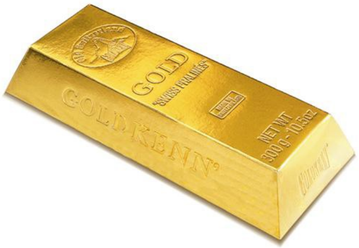


$200

20g

$900

53g



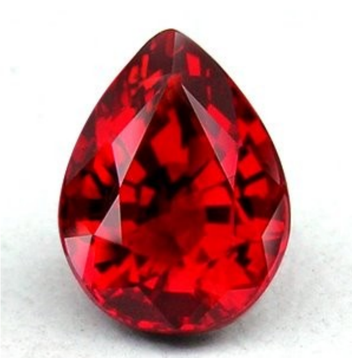
$1,500

25g

$400

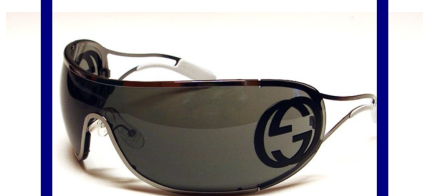
35g

The 0/1 Knapsack Problem



$500

45g

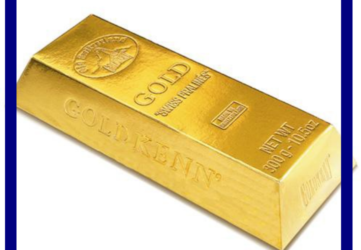


$200

20g

$900

53g



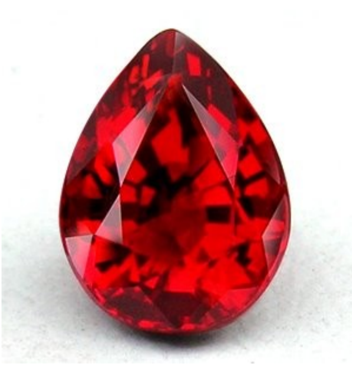
$1,500

25g

$400

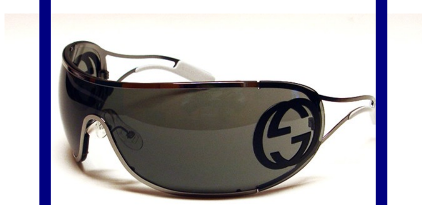
35g

The 0/1 Knapsack Problem



$500

45g

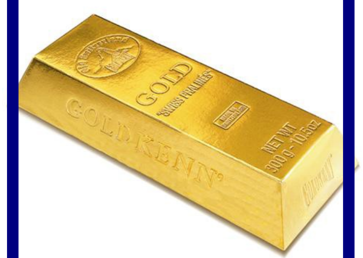


$200

20g

$900

53g



$1,500

25g

$400

35g

The 0/1 Knapsack Problem

- You are given a list of n items with weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$.
- You have a bag (knapsack) that can carry W total weight.
- Weights are assumed to be integers.
- **Question:** What is the maximum value of items that you can fit into the knapsack?
- This problem is known to be **NP**-hard.

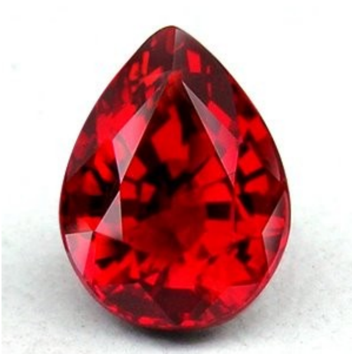
A Naïve Solution

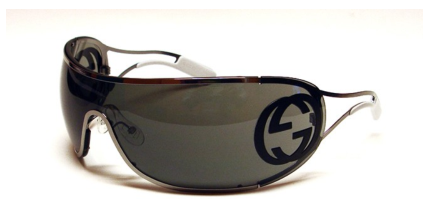
- One option: Try all possible subsets of the items and find the feasible set with the largest total value.
- How many subsets are there?
  - Answer: $\mathbf{2^n}$.
- Subsets can be generated in O(n) time each.
- Total runtime is $\mathbf{O(2^n\ n)}$.
- Slightly better than TSP, but still not particularly good!

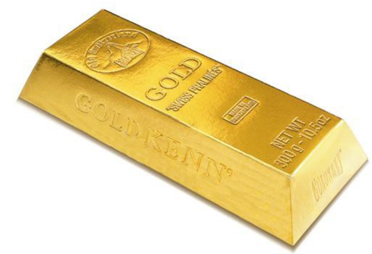
A Greedy Solution

- Sort items by their "unit value:" v_k / w_k.
- For all items in descending unit value:
  - If that item will fit in the knapsack, add it to the knapsack.
- Does this algorithm always return an optimal solution?
- Unfortunately, **no**; in fact, this algorithm can be *arbitrarily bad!*

The 0/1 Knapsack Problem

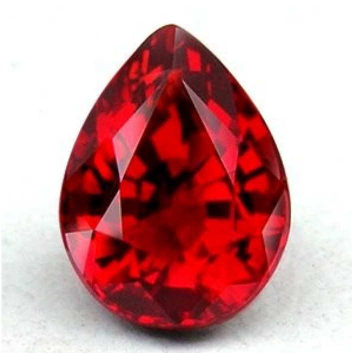






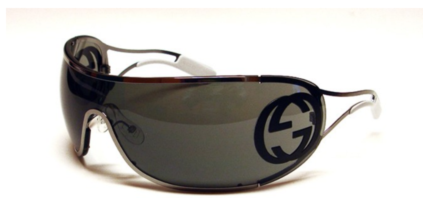
$500	$200	$900	$1,500	$400
45g	20g	53g	25g	35g

The 0/1 Knapsack Problem



$500

45g

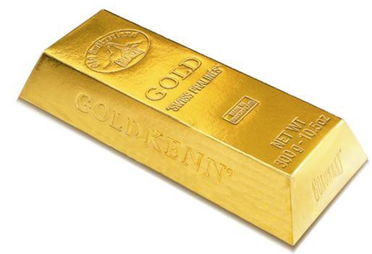


$200

20g

$900

53g



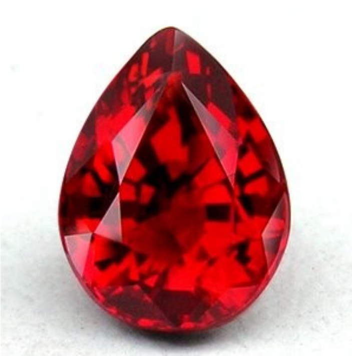
$1,500

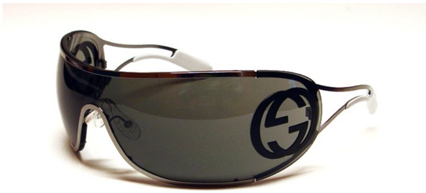
25g

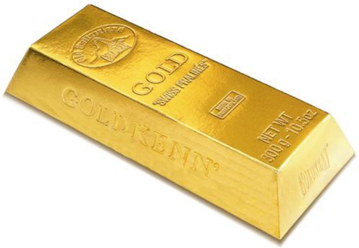
$400

35g

The 0/1 Knapsack Problem







$500	$200	$900	$1,500	$400
45g	20g	53g	25g	35g

A Recurrence Relation

- Let OPT(k, X) denote the maximum value that can be made from the first k items without exceeding weight X.
  - Note: OPT(n, W) is the overall answer.
- **Claim:** OPT(k, X) satisfies this recurrence:

$$\mathrm{OPT}(k, X) = \begin{cases} 0 & \text{if } k = 0 \\ OPT(k-1, X) & \text{if } w_k > X \\ max\left\{\begin{matrix} OPT(k-1, X), \\ v_k + OPT(k-1, X - w_k) \end{matrix}\right\} & \text{otherwise} \end{cases}$$

$$\mathrm{OPT}(k, X) = \begin{cases} 0 & \text{if } k = 0 \\ OPT(k-1, X) & \text{if } w_k > X \\ max\left\{ \begin{array}{c} OPT(k-1, X), \\ v_k + OPT(k-1, X - w_k) \end{array} \right\} & \text{otherwise} \end{cases}$$

Let DP be a table of size $(n + 1) \times (W + 1)$.

For $X = 0$ to $W + 1$:

  Set DP[0][X] = 0

For $k = 1$ to n:

  For $X = 0$ to W:

    If $w_k > W$, set DP[k][X] = DP[k – 1][X].

    Else, set DP[k][X] = max{
        DP[k – 1][X],
        v_k + DP[k – 1][X – w_k]
    }

Return DP[n][W].

Um... Wait...

- Runtime of this algorithm is $O(nW)$ and space complexity is $O(nW)$.
- This is a polynomial in n and W.
- This problem is **NP**-hard.

Did we just prove P = NP?

A Note About Input Sizes

- A polynomial-time algorithm is one that runs in time polynomial *in the total number of bits required to write out the input to the problem*.
- How many bits are required to write out the value W?
  - Answer: **O(log *W*)**.
- Therefore, O(nW) is **exponential** in the number of bits required to write out the input.
  - Example: Adding one more bit to the end of the representation of W doubles its size and doubles the runtime.
- This algorithm is called a ***pseudopolynomial time algorithm***, since it is a polynomial in the *numeric value* of the input, not the number of bits in the input.

That Said…

- The runtime of O(nW) is better than our old runtime of O(2^n n) assuming that $W = o(2^n)$.
  - That's *little-o*, not big-O.
- In fact – for *any* fixed W, this algorithm runs in linear time!
- Although there are exponentially many subsets to test, we can get away with just linear work if W is fixed!

Parameterized Complexity

- **Parameterized complexity** is a branch of complexity theory that studies the hardness of problems with respect to different “parameters” of the input.
- Often, **NP**-hard problems are not entirely infeasible as long as some “parameter” of the problem is fixed.
- In our case, $O(nW)$ has two parameters – the number of elements (n) and weight (W).

Fixed-Parameter Tractability

- Suppose that the input to a problem P can be characterized by two parameters n and k.
- P is called **fixed-parameter tractable** iff there is some algorithm that solves P in time $O(f(k) \cdot p(n))$, where
  - $f(k)$ is an arbitrary function.
  - $p(n)$ is a polynomial in n.
- Intuitively, for any fixed k, the algorithm runs in a polynomial in n.
  - That polynomial is always the same polynomial regardless of the choice of k.

Example: **Finding Long Paths**

The Long Path Problem

- Given a graph $G = (V, E)$ and a number k, we want to determine whether there is a simple path of length k exists in G.
- Known to be **NP**-hard by a reduction from finding Hamiltonian paths: a graph has a Hamiltonian path iff it has a simple path of length $n - 1$.
- Applications in biology to finding protein signaling cascades.

A Naïve Approach

- To find all simple paths of length k, enumerate all $(k + 1)$-permutations of nodes in V and check if each is a path.
- How many such permutations are there?
  - Answer: $\boldsymbol{n! \,/\, (n - k - 1)!}$
- Time to process each is $O(k)$ when using an adjacency matrix.
- Total runtime is $\boldsymbol{O(k \cdot n! \,/\, (n - k - 1)!)}$
- Decent for small k, unbelievably slow for larger k.

A Better Approach

- We can use a *randomized* technique called **color-coding** to speed this up.
- Suppose every node in the graph is colored one of $k + 1$ different colors. A **colorful path** is a simple path of length k where each node has a different color.
- **Idea:** Show how to find colorful paths efficiently, then build a randomized algorithm for finding long paths that uses the colorful path finder as a subroutine.

Finding Colorful Paths: Seem Familiar?

Finding Colorful Paths

- Using a dynamic programming approach similar to TSP, can find all colorful paths originating at a node s in time $O(2^k n^2)$.
- Can find colorful paths between any pair of nodes in time $O(2^k n^3)$ by iterating this process for all possible start nodes.
- This is fixed-parameter tractable!

Random Colorings

- Suppose you want to find a simple path of length k in a graph.
- Randomly color all nodes in the graph one of $(k + 1)$ different colors.
- If P is a simple path of length k in G, what is the probability that it is a colorful path?
  - Answer: $\boldsymbol{(k + 1)! \,/\, (k + 1)^{k+1}}$

Stirling's Approximation

- **Stirling's approximation** states that

$$n! \geq \frac{n^n}{e^n}\sqrt{2\pi n}$$

- Therefore, $(k + 1)! \,/\, (k + 1)^{k+1} \geq 1 \,/\, e^{k+1}$.
- If we randomly color the nodes in G e^{k+1} times, the probability that any simple path of length k never becomes colorful is at most $1 / e$.
- Doing $e^{k+1} \ln n$ random colorings means we find a simple path of length k with high probability.
- Total runtime: $O(e^k 2^k n^3 \log n)$ = $\mathbf{O((2}\boldsymbol{e}\mathbf{)}^{\boldsymbol{k}}\ \boldsymbol{n}^{\mathbf{3}}\ \mathbf{log}\ \boldsymbol{n}\mathbf{)}$.
- Better than naïve solution in many cases!

Why All This Matters

- Last lecture: Brute-force search is not necessarily optimal for **NP**-hard problems.
- Today: Can often factor out the complexity into a “tractable” part and “intractable” part that depend on different parameters.
- Plus, we got to see DP combined with randomized algorithms!

Next Time

- Approximation Algorithms
- FPTAS's and Other Acronyms