Question:

Given a problem of a certain size, how long will it take to solve it?

Two Kinds of Answers:

- **Average Case.** How long for a *typical* problem.
- **Worst Case.** How long for the *hardest* problem.

**Average Case.**

- Mathematically difficult (define average!).
- Empirical studies.

**Worst Case.**

- Mathematically tractible (sometimes).
- Limited value.
Measures

Measures of Size

- Number of constraints \( m \) and/or number of variables \( n \).
- Number of data elements, \( mn \).
- Number of nonzero data elements.
- Size, in bytes, of AMPL formulation (model+data).

I recently solved a practical generalized network flow problem (these terms will be defined later) having \( n = 273,798 \) variables and \( m = 79,915 \) constraints. But, the number of nonzero data elements is just \( 339,558 \), which is much less than \( m \times n = 21,880,567,170 \) (that’s 21 BILLION). It solves on my PC in just 2.5 seconds using LOQO and 1.9 seconds using GUROBI.

START HERE ON THURSDAY

Measuring Time

Two factors:

- Number of iterations.
- Time per iteration.
Klee–Minty Problem (1972)

maximize \[ \sum_{j=1}^{n} 2^{n-j} x_j \]
subject to \[ 2 \sum_{j=1}^{i-1} 2^{i-j} x_j + x_i \leq 100^{i-1} \quad i = 1, 2, \ldots, n \]
\[ x_j \geq 0 \quad j = 1, 2, \ldots, n. \]

Example \( n = 3 \):

maximize \[ 4x_1 + 2x_2 + x_3 \]
subj. to \[ x_1 \leq 1 \]
\[ 4x_1 + x_2 \leq 100 \]
\[ 8x_1 + 4x_2 + x_3 \leq 10000 \]
\[ x_1, x_2, x_3 \geq 0. \]
A Distorted Cube

Constraints represent a “minor” distortion to an $n$-dimensional hypercube:

\begin{align*}
0 & \leq x_1 \leq 1 \\
0 & \leq x_2 \leq 100 \\
\vdots \\
0 & \leq x_n \leq 100^{n-1}.
\end{align*}
Replace

\[ 1, 100, 10000, \ldots, \]

with

\[ 1 = \beta_1 \ll \beta_2 \ll \beta_3 \ll \ldots. \]

Then, make following replacements to rhs:

\[
\begin{align*}
\beta_1 & \rightarrow \beta_1 \\
\beta_2 & \rightarrow 2\beta_1 + \beta_2 \\
\beta_3 & \rightarrow 4\beta_1 + 2\beta_2 + \beta_3 \\
\beta_4 & \rightarrow 8\beta_1 + 4\beta_2 + 2\beta_3 + \beta_4 \\
\vdots
\end{align*}
\]

Hardly a change!

Make a similar constant adjustment to objective function.

Look at the pivot tool version…
Case \( n = 3 \):  

**Simplex Challenge -- Klee-Minty**

Your Name: Blob

constraints: 3, variables: 3, Go Pivoting

| Undp | Number format: Decimal ♦ |

**Current Dictionary:**

maximize \( \zeta = -2 \beta_1 + -1 \beta_2 + 0 \beta_3 + 4 x_1 + 2 x_2 + 1 x_3 \)

<table>
<thead>
<tr>
<th></th>
<th>( w_1 )</th>
<th>( w_2 )</th>
<th>( w_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>1 ( \beta_1 ) + 0 ( \beta_2 ) + 0 ( \beta_3 ) - 1 ( x_1 ) - 0 ( x_2 ) - 0 ( x_3 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_2 )</td>
<td>2 ( \beta_1 ) + 1 ( \beta_2 ) + 0 ( \beta_3 ) - 4 ( x_1 ) - 1 ( x_2 ) - 0 ( x_3 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( x_3 )</td>
<td>4 ( \beta_1 ) + 2 ( \beta_2 ) + 1 ( \beta_3 ) - 8 ( x_1 ) - 4 ( x_2 ) - 1 ( x_3 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( x_1, x_2, x_3, w_1, w_2, w_3 \geq 0 \)

Optimal

Now, watch the pivots...
Klee–Minty problem shows that:

Largest-coefficient rule can take $2^n - 1$ pivots to solve a problem in $n$ variables and constraints.

For $n = 70$,

$$2^n = 1.2 \times 10^{21}.$$  

At 1000 iterations per second, this problem will take 40 billion years to solve. The age of the universe is estimated to be 13.7 billion years.

Yet, problems with 10,000 to 100,000 variables/constraints are solved routinely every day.

Worst case analysis is just that: worst case.
## Complexity

<table>
<thead>
<tr>
<th>n</th>
<th>n ^2</th>
<th>n ^3</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>27</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>125</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>216</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>49</td>
<td>343</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>81</td>
<td>729</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>1000</td>
<td>1024</td>
</tr>
<tr>
<td>12</td>
<td>144</td>
<td>1728</td>
<td>4096</td>
</tr>
<tr>
<td>14</td>
<td>196</td>
<td>2744</td>
<td>16384</td>
</tr>
<tr>
<td>16</td>
<td>256</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>18</td>
<td>324</td>
<td>5832</td>
<td>262144</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>8000</td>
<td>1048576</td>
</tr>
<tr>
<td>22</td>
<td>484</td>
<td>10648</td>
<td>4194304</td>
</tr>
<tr>
<td>24</td>
<td>576</td>
<td>13824</td>
<td>16777216</td>
</tr>
<tr>
<td>26</td>
<td>676</td>
<td>17576</td>
<td>67108864</td>
</tr>
<tr>
<td>28</td>
<td>784</td>
<td>21952</td>
<td>268435456</td>
</tr>
<tr>
<td>30</td>
<td>900</td>
<td>27000</td>
<td>1073741824</td>
</tr>
</tbody>
</table>

Sorting: fast algorithm = $n \log n$, slow algorithm = $n^2$

Matrix times vector: $n^2$

Matrix times matrix: $n^3$

Matrix inversion: $n^3$

Simplex Method:

- Worst case: $n^2 2^n$ operations.
- Average case: $n^3$ operations.
- Open question:

  Does there exist a variant of the simplex method whose worst case performance is polynomial?

Linear Programming:

- **Theorem:** There exists an algorithm whose worst case performance is $n^{3.5}$ operations.
Define a random problem:

\[
\begin{align*}
m &= \text{int}(\text{ceil}(10 \cdot \exp(\log(\text{maxsize}/10) \cdot \text{random.rand()}))) \\
n &= \text{int}(\text{ceil}(10 \cdot \exp(\log(\text{maxsize}/10) \cdot \text{random.rand()}))) \\
A &= \text{around}(\sigma \cdot \text{random.randn}(m, n), 0) \\
b &= \text{array}(\text{around}(\sigma \cdot \text{abs}(\text{random.randn}(m, 1)), 0)) \\
c &= \text{array}(\text{around}(\sigma \cdot \text{random.randn}(n, 1), 0)) \\
A &= -A
\end{align*}
\]

Initialize a few things:

\[
\begin{align*}
\text{nonbasics} &= \text{arange}(n) \\
\text{basics} &= n + \text{arange}(m) \\
\text{iter} &= 0 \\
\text{opt} &= 0
\end{align*}
\]
The Main Loop:

while ( (max(c) > eps) ):
    col = argmax(c)
    Acol = A[:,col].reshape(m,1)
    tmp = -Acol/(b+eps)
    row = argmax(tmp)
    if ( sum( Acol<-eps ) == 0 ):
        opt = -1
        break
    j = nonbasics[col]
    i = basics[row]

    Arow = A[row,:]
    a = A[row,col]

    iter = iter+1

The code for a pivot:

A = A - Acol*Arow/a
A[row,:] = -Arow/a
A[:,col] = Acol.reshape(1,m)/a
A[row,col] = 1/a

# update the right-hand side
brow = b[row,0]
b = b - brow*Acol/a
b[row] = -brow/a

# update the objective function
ccol = c[col,0]
c = c - ccol*(Arow.reshape(n,1))/a
c[col] = ccol/a

# swap variables $x_j$ and $x_i$ in the dictionary
basics[row] = j
nonbasics[col] = i
Empirical Performance of the Simplex Method

- Problems with an optimal solution
- Problems that are unbounded
Empirical Performance of the Simplex Method

- Problems with an optimal solution
- Problems that are unbounded

Number of pivots vs. $n$
\[ \text{iters} = 0.122 \min(m, n)^{1.77} \]

\[ \text{iters} = 0.153 \min(m, n)^{1.43} \]
Declare parameters:

```plaintext
param eps := 1e-9;
param sigma := 30;
param niters := 1000;
param size := 400;

param m;
param n;
param AA {1..size, 1..size};
param bb {1..size};
param cc {1..size};
param A {1..size, 1..size};
param b {1..size};
param c {1..size};
param x {1..size};
param z {1..size};
param w {1..size};

param Arow {1..size};
param Acol {1..size};
param a;
param brow;
param ccol;
param ii;
param jj;
```

Define a random problem:

```plaintext
let m := ceil(exp(log(size)*Uniform01()));
let n := ceil(exp(log(size)*Uniform01()));
let {i in 1..m, j in 1..n} A[i,j] := round(sigma*Normal01());
let {i in 1..m} b[i] := round(sigma*abs(Normal01()));
let {j in 1..n} c[j] := round(sigma*Normal01());
let {i in 1..m, j in 1..n} AA[i,j] := A[i,j];
let {i in 1..m} bb[i] := b[i];
let {j in 1..n} cc[j] := c[j];
```
The Simplex Method (Phase 2)

repeat while (max {j in 1..n} c[j]) > eps {
  let maxc := 0;
  for {j in 1..n} {
    if (c[j] > maxc) then {
      let maxc := c[j];
      let col := j;
    }
  }
  let minbovera := 1/eps;
  for {i in 1..m} {
    if (A[i,col] < -eps) then {
      if (-b[i]/A[i,col] < minbovera) then {
        let minbovera := -b[i]/A[i,col];
        let row := i;
      }
    }
  }
  if minbovera >= 1/eps then {
    let opt := -1; # unbounded
    display "unbounded";
    break;
  }
  .
  .
  .
}

The code for a pivot:

let {j in 1..n} Arow[j] := A[row,j];
let {i in 1..m} Acol[i] := A[i,col];
let a := A[row,col];
let {i in 1..m, j in 1..n}
let {j in 1..n} A[row,j] := -Arow[j]/a;
let {i in 1..m} A[i,col] := Acol[i]/a;
let A[row,col] := 1/a;
let brow := b[row];
let {i in 1..m}
  b[i] := b[i] - brow*Acol[i]/a;
let b[row] := -brow/a;
let ccol := c[col];
let {j in 1..n}
  c[j] := c[j] - ccol*Arow[j]/a;
let c[col] := ccol/a;
The Python notebook can be found here:

https://vanderbei.princeton.edu/307/python/PrimalSimplexComplexity.ipynb

The AMPL code can be found here:

https://vanderbei.princeton.edu/307/lectures/primalsimplex.txt