Overview of Big-O Notation
Motivation - Will it fit in memory and finish running?

• Suppose you have written code that takes in data.

```matlab
function findMax(y)
    n = length(y)
    maxValue = -Inf
    for i in 1:n
        if y[i] > maxValue
            end
            maxValue = y[i]
        end
    end
end
```

• We want to get an idea of how the code will perform on large inputs.
  • Is it likely to run out of memory?
  • Is it likely to take a very long time?
• We use big-O notation to approximately answer these questions.
  • A crude measure of how memory or time scale with the data size.
Motivation - Will it fit in memory and finish running?

• Informally:
  • Big-O time complexity is the product of the loop indices for the deepest loops.
  • Big-O memory complexity: product of loop indices to go through all stored data at worst time.

• How can this measure be useful?
  • If the time/memory is $O(2^n)$ for inputs of size $n$, we can only use it for tiny datasets.
  • We can apply $O(n^2)$ algorithms to medium-sized datasets.
  • We can apply $O(n)$ algorithms to huge datasets.
  • Algorithms that take $O(\log n)$ are extremely fast (barely gets slower as data size increases).
  • Some algorithms are even $O(1)$, meaning they do not depend on dataset size.
Big-O: Formal Definition

• Formally, the notation “\( g(n) \text{ is in } O(f(n)) \)” means:
  
  • “for all sufficiently large “n”, \( g(n) \leq c*f(n) \) for some constant \( c > 0 \).”

• I find this concept easiest to learn by examples….
Big-O Arithmetic (Single Variable)

- Some examples for you to work through (assume ‘n’ is a positive integer):
  
  - Any constant number is in $O(1)$, so 10 is in $O(1)$.
    
    - Because $10 \leq 10*1$ for any “n”.
    
    - Here, 10 is a constant that makes the right “O” side bigger for any ‘n’.
  
  - Any constant multiplied by ’n’ is in $O(n)$, so 5n is in $O(n)$.
    
    - Because $5n \leq 5*n$ for any “n”.
    
    - Here, 5 is a constant that makes the right “O” side bigger for any ‘n’.
  
  - Slower-growing terms can be ignored, so $20n + 50 = O(n)$.
    
    - Because $20n + 50 \leq 70n$ for any “n”.
    
    - Here, 70 is a constant that makes the right “O” side bigger for any ‘n’.
Big-O Arithmetic (Single Variable)

- Only large values of “n” matter, and only largest-exponent polynomial matters.
  - So $4n^3 + 50n^2 + 100n + 10000$ is in $O(n^3)$.
    - Because $(4n^3 + 50n^2 + 100n + 10000) \leq 5n^3$ for $n > 100$.

- Slower-growing terms are trivially in the class of faster-growing terms.
  - $O(n^2)$ is in $O(n^3)$, as cubic will be larger for sufficiently large “n”.
  - But usually we try to find smallest $O()$ value, so we would use $O(n^2)$ and not $O(n^3)$ if we can.

- Exponentials grow faster than polynomials which grow faster than logarithms.
  - And $n^{10} + 2^n$ is in $O(2^n)$.
  - So $50n^3 + 5*\log(n)$ is in $O(n^3)$.

- Basically, you drop multiplicative constants and remove terms that do not dominate for large “n”.
  - $40*\log(n) + 100$ is in $O(\log(n))$.
  - $3n*\log(n) + 20n$ is in $O(n*\log(n))$.
  - $3n*\log(n) + 20n^2$ is in $O(n^2)$.
  - $4n^2 + 8n^3 + 10n*\log(n)$ is in $O(n2^n)$.
Example: Finding Maximum of a Vector (Memory)

```
function findMax(y)
    n = length(y)
    maxValue = -Inf
    for i in 1:n
        if y[i] > maxValue
            maxValue = y[i]
        end
    end
end
```

- The input in this example is a list of “n” numbers.
- The size of this input is thus “n” times the cost to store one number.
- We assume that the cost to store one number is $O(1)$.
- So the cost of storing the input is $O(n)$.
- The algorithm itself only stores the extra variables “maxVal” and “i”.
  - Plus maybe some bookkeeping.
  - So the additional storage required by the algorithm is $O(1)$.
- Combining the input storage and algorithm storage gives $O(n) + O(1)$ or $O(n)$ memory complexity.
Example: Finding Maximum of a Vector (Time)

```plaintext
function findMax(y)
    n = length(y)
    maxValue = -Inf
    for i in 1:n
        if y[i] > maxValue
            maxValue = y[i]
        end
    end
end
```

- First two lines do not depend on “n”, so we say it costs O(1).
  - We assume that “y” knows its own length.
- Runtime of lines inside “for” loop do not depend “n”, so they also cost O(1).
  - We will assume that comparing and assigning numbers takes O(1).
- But the number of times we go through the “for” loop is “n”.
  - So the cost is O(1) for the first lines, and then n*O(1) for the “for” loop.
  - So time complexity is O(1) + n*O(1) or O(n) time complexity.
Watch out for sub-functions!

• An alternative way to compute the maximum is with this line:

```
maximum(y)
```

• Does this cost $O(1)$ because there is no “for” loop?

• No! The “maximum” function still needs to loop through all elements of “y”.
  
  • The “for” loop is just hidden inside the function.
  
  • The time complexity of the above line of code is $O(n)$.

• What does having $O(n)$ time complexity mean?
  
  • If it takes ~1 seconds with $n=10,000$, it takes ~10 seconds with $n=100,000$.
  
  • For large inputs, time will grow linearly with input size.
    
    • You would expect this code to finish in a reasonable amount of time even if “n” is 1 billion.
Example: Finding Maximum and Minimum

- Consider finding the maximum and the minimum:

- There are 2 “for” loops, but they are not nested.
  - Each one costs O(n).

- So total cost of this code is in O(n)+O(n) which is O(n).
Example: Showing all Pairs of Products

• Consider showing all products between numbers:

```
function showAllPairs(y)
    n = length(y)
    for i in 1:n
        for j in 1:n
            @show y[i]*y[j]
        end
    end
end
```

• In this case the “for” loops are nested.
  • Inner “for” loops costs $O(n)$.
  • But outer “for” loop makes us call the “inner” loop “n” times.

• So the time complexity of this code is in $O(n) \times O(n)$ which is in $O(n^2)$.
  • Though the memory complexity is still $O(n)$.

• This code will get slower at a faster than linear rate as “n” gets big.
  • You would not expect this code to finish in a reasonable amount of time if “n” is 1 billion.
Example: Returning all Pairs of Products

Consider returning all products between numbers:

```python
function allPairs(y)
    n = length(y)
    yy = zeros(n,2)
    for i in 1:n
        for j in 1:n
            yy[i,j] = y[i]*y[j]
        end
    end
    return yy
end
```

The time complexity is still $O(n^2)$.

But the memory complexity is now $O(n^2)$ too.

You would need nested “for” loops that go through all “$n^2$” elements of “yy”.

This code’s memory grows at a faster than linear rate as “n” gets big.

You would expect this code to run out of memory if “n” is 1 billion.

An alternate way to implement this would be with an outer product: $yy = y*y'$

Note that this 1 line would also cost $O(n^2)$ time and require $O(n^2)$ memory.
Standard Sorting and Searching Time Costs

- Some well-known big-O results:
  - Given a list of “n” numbers, sorting costs $O(n \cdot \log(n))$ time.
    - Sometimes just written as $O(n \log n)$.
  - Given a list of “n” numbers, finding kth largest costs $O(n)$.
    - Faster than sorting: you can avoid sorting using a “select” algorithm.
  - Given a sorted list of “n” numbers, finding kth largest costs $O(1)$.
    - You can just return the kth element.
  - Given a list of “n” numbers, finding smallest greater than “α” costs $O(n)$.
  - Given a sorted list of “n” numbers, finding smallest greater than “α” costs $O(\log n)$.
    - Using a binary search where each step throws away half the remaining possible answers.
  - Standard operations on hash data structures cost $O(1)$.
    - Looking up key, inserting new element, deleting element.
Big-O with Multiple Variables
Motivation for Multiple Variables

• Our input size often depends on more than one variable.
  • Our data matrix ‘X’ typically has ‘n’ rows and ‘d’ columns.

• We can still use big-O in this setting.
  • To consider time/memory in terms of both variables.

• Example:
  • It costs $O(nd)$ memory to store ‘X’.
  • It takes $O(nd)$ time to find the maximum element of ‘X’.
    • This is ok for large values of ‘n’ and ‘d’.
    • Although if both ‘n’ and ‘d’ are large, this may be prohibitive.
Example: Computing Sum of Matrix

- Consider code for computing sum of all elements of a matrix:

```python
function matrixSum(X)
    (n, d) = size(X)
    sm = 0
    for i in 1:n
        for j in 1:nd
            sm += X[i, j]
        end
    end
    return sm
end
```

- The “sm +=“ line costs O(1).
- The O(1) cost of the inner loop is repeated ‘d’ times, giving O(d).
- The O(d) cost of the outer loop is repeated ’n’ times, giving O(nd).
Big-O Arithmetic (Multiple Variables)

• Additional rule for multiple variables:
  • Include terms that could be dominant for any combination of variables.

• Examples:
  • $O(n) + O(d)$ is in $O(n + d)$.
  • $O(n^3) + O(\log(d)) + O(n)$ is in $O(n^3 + \log(d))$.
  • $O(n^2) + O(nd) + O(d) + O(d^3)$ is in $O(n^2 + nd + d^3)$.
  • $O(n^2d^3) + O(d^3n^2) + O(n^2d^2) + O(n^3)$ is in $O(n^3 + n^2d^3 + d^3n^2)$.
  • $O(n\sqrt{m}) + O(n^2) + O(\sqrt{m}) + O(n^*\log(m))$ is in $O(n^2 + n\sqrt{m})$. 
Standard Linear Algebra Time Costs

• Some well-known linear algebra costs:
  
  • Multiplying ‘d’ by 1 vector ‘w’ by scalar $\alpha$, $\alpha w$ costs $O(d)$.
    • For loop over elements of ‘w’.
  
  • Adding two ‘d’ by 1 vectors ‘w’ and ‘v’, $w+v$ costs $O(d)$.
    • For loop over elements of ‘w’.
  
  • Dot products between vectors, $w^T v$, and norms of vector $||w||$ cost $O(d)$.

  • Scalar multiplication and addition of ’n’ times ‘d’ matrices is $O(nd)$.
    • Double for loop over elements of matrix.
Standard Linear Algebra Time Costs

• Some well-known linear algebra costs:
  
  • Matrix-vector product with 'n’ times ‘d’ matrix ‘X’, \( Xw \) costs \( O(nd) \).
    
    • Double loop over all elements of ‘X’.
    
    • Same cost for matrix-vector product with transpose, \( X^Ty \).
  
  • Matrix-matrix product of 'n’ times ‘d’ matrix X with ‘d’ times ‘k’ matrix W, \( XW \) costs \( O(ndk) \).
    
    • Double loop over all elements of the 'n’ times ‘k’ resulting matrix.
      
      • Each element of the matrix requires computing an \( O(d) \) dot product.
    
    • There exist faster ways to implement this, but we will use the \( O(ndk) \) cost for this course.
  
  • Inverting an ‘n’ times ‘n’ matrix or solving an ‘n’ times ‘n’ linear system costs \( O(n^3) \).
    
    • Perform up to ‘n’ stages of Gaussian elimination, each costing \( O(n^2) \).
    
    • Faster methods exist, but this course will use \( O(n^3) \) cost of basic implementation.
Example: Least Squares with Normal Equations

- Cost to solve normal equations for least squares: $X^TXw = X^Ty$.
  - Cost of $O(nd)$ for matrix-vector product $b=X^Ty$.
  - Cost of $O(n^2d)$ for matrix-matrix product $A=X^TX$.
  - Cost of $O(n^3)$ to solve ’$n$’ times ’$n$’ linear system $Aw=b$.
- Total cost of $O(nd + n^2d + n^3) = O(n^2d + n^3)$. 
Decision Trees and Stumps
**Decision Stumps**

- **$O(n^2d)$ decision stump pseudocode:**
  
  ```
  Input: feature matrix $X$ and label vector $y$
  
  $(n, d) = \text{size}(X)$
  
  minError = $\sum(y \neq \text{mode}(y))$  # compute error if you don't split (user-defined function `mode`)
  
  minRule = []  # initialize with empty list

  for $j = 1:d$
      for $i = 1:n$
          $t = X[i,j]$
          
          y_above = $\text{mode}(y[X[:,j] > t])$
          
          y_below = $\text{mode}(y[X[:,j] \leq t])$
          
          $yhat[t] = f(||y_{above} - y||)$
          
          $yhat[t] = y_{below}$
          
          error = $\sum(yhat \neq y)$
          
          if error < minError
              minError = error
              minRule = $t$
          
  # set threshold to feature $j$ in example $i$:
  # find mode of label vector when feature $j$ is above threshold
  # find mode of label vector when feature $j$ is below threshold
  # classify all examples based on threshold
  # count the number of errors
  # store this rule if it has the lowest error so far.
  ```

- Number of outer loop iterations is ‘$d$’.

- Number of inner loop iterations is ‘$n$’.

  - Cost of operations in the inner loop is $O(n)$
    
    - Finding mode among ‘$n$’ objects is $O(n)$ (may need to use dictionary if very sparse).
    
    - Assigning labels and computing error also costs $O(n)$.

  - But runtime can be reduced to $O(nd \log n)$.

    - At start of each outer loop, sort the $X[:,j]$ values for cost of $O(n \log n)$.
    
    - Each inner loop updates mode/assignments/error for example ‘$i$’, for cost of $O(1)$.
Decision Trees - Naive Analysis

- Using greedy decision tree learning:
  - With depth of 1, we need to fit 1 decision stump.
  - With depth of 2, we need to fit up to 3 decision stumps.
  - With depth of 3, we need to fit up to 7 decisions stumps.
  - With depth of 4, we need to fit up to 15 decision stumps.
  - With depth of ‘m’, we need to fit up $2^{m-1}$ decision stumps.

- Since fitting one stump costs $O(nd \log n)$, cost of fitting tree is $O(2^{mnd} \log n)$.
  - But this is too pessimistic: it can be improved to $O(nd(m + \log n))$. 
Decision Trees - $O(nd(m + \log n))$ implementation

• Instead of having each stump sort, you could sort all features once.
  • One-time sorting cost of $O(nd \log n)$.
  • But with sorted features fitting stumps only costs $O(nd)$.
• Now use the fact that each example is only assigned to one stump per depth:
  • If all training examples are in one leaf node to be split:
    • We fit one decision stump, at a cost $O(nd)$.
  • If we have $n_1$ examples in one leaf and $n_2$ examples in another ($n_1 + n_2 = n$):
    • Fit one decision stump at cost $O(n_1d)$ and the other with cost $O(n_2d)$.
    • So total cost is $O((n_1 + n_2)d)$ which is in $O(nd)$.
  • No matter how examples are distributed, total cost for one depth is $O(nd)$.
• Get result by combining one time sort cost of $O(nd \log n)$, and depth cost of $O(nd)$ for each depth ‘$m$’.
  • In practice, most implementations do not pre-sort which is similar in practice but slower theoretically.