Algorithms

An Introduction

Outline

- **Motivational Example: Steam Sale**
- **What is an algorithm?**
- **How do we assess algorithms?**
- **Sorting Problems**
  - Bubble Sort
  - Merge Sort

Motivational Example

Steam Sale

- **When summer is over,** as usual, online game platform Steam launches its seasonal sales. It's time to **give your money to Gabe** (founder of Valve Corp., the chubby dude you saw in video during Week 1)

Steam Sale

- The catch: You can only access the sale once, and your laptop only has 20GB (I know this limit seems ridiculous) of free space to install the games.

	Value ($)	Size(GB)
Civilization V	190	10
CS:GO	300	15
The Elder Scrolls V	180	10
Torchlight II	20	2
Age of Empires II	35	5
Banished	10	1

Most Valuable First

- What if you just purchase the most valuable games into your inventory?

	Value ($)	Size(GB)
Civilization V	190	10
CS:GO	300	15
The Elder Scrolls V	180	10
Torchlight II	20	2
Age of Empires II	35	5
Banished	10	1

What would you end up with? (Keep in mind the space limit)

Maximum number of Games

- What if you just purchase as many games as possible?

	Value ($)	Size(GB)
Civilization V	190	10
CS:GO	300	15
The Elder Scrolls V	180	10
Torchlight II	20	2
Age of Empires II	35	5
Banished	10	1

What would you end up with? (Keep in mind the space limit)

Best Value/Space Ratio

- What if you purchase games with best value for size?

	Value ($)	Size(GB)
Civilization V	190	10
CS:GO	300	15
The Elder Scrolls V	180	10
Torchlight II	20	2
Age of Empires II	35	5
Banished	10	1

What would you end up with? (Keep in mind the space limit)

What's In Common?

- Though the metric we care about is different each time, the three approaches share a number of steps:

1. **Decide upon a metric** that is valuable to you (value, count of games, ratio of value to size)
2. **Sort the items** by your **metric**
3. Add items to your result until you **hit your constraint**

- They're all **algorithms**
- They're all **greedy**
- They share a common method of **approach**

What Are Our Results?

- Interestingly, they are all different

```
Filling your 20GB space with the most valuable games:
Total value of items purchased = 335.0
- $ 300.0,        15.0,  CS:GO
- $ 35.0,         5.0,   Age of Empires II

Filling your 20GB space with as many games as possible:
Total value of items purchased = 255.0
- $ 10.0,         1.0,   Banished
- $ 20.0,         2.0,   Torchlight II
- $ 35.0,         5.0,   Age of Empires II
- $ 190.0,        10.0,  Civilization V

Filling your 20GB space with high value/size games:
Total value of items purchased = 330.0
- $ 300.0,        15.0,  CS:GO
- $ 20.0,         2.0,   Torchlight II
- $ 10.0,         1.0,   Banished
```

- **Can we do better?**

What Are Our Results?

- Interestingly, they are all different

```
Filling your 20GB space with the most valuable games:
Total value of items purchased = 335.0
- $ 300.0,        15.0,  CS:GO
- $ 35.0,         5.0,   Age of Empires II

Filling your 20GB space with as many games as possible:
Total value of items purchased = 255.0
- $ 10.0,         1.0,   Banished
- $ 20.0,         2.0,   Torchlight II
- $ 35.0,         5.0,   Age of Empires II
- $ 190.0,        10.0,  Civilization V

Filling your 20GB space with high value/size games:
Total value of items purchased = 330.0
- $ 300.0,        15.0,  CS:GO
- $ 20.0,         2.0,   Torchlight II
- $ 10.0,         1.0,   Banished
```

- **Can we do better?** Try **exhaustive search** to find the **highest total value** under our 20G space constraint

The Result: Exhaustive Search

- Is this surprising to you?

Algorithm	Result
Buy the most valuable games	$335.0
Buy as many games as possible	$255.0
Buy games w/ best value/size ratio	$330.0
Buy for highest total value using exhaustive search	$370.0

```
Using exhaustive search, total value of games purchased = 370.0
- $ 190.0,        10.0,  Civilization V
- $ 180.0,        10.0,  The Elder Scrolls V
```

- Our different methods of approach yielded different results
  1. **Decide upon a metric** that is valuable to you
  2. **Sort the items** by your metric
  3. Add items to your result until you **hit your constraint**
- What are some of the trade-offs for each method?
- **Can we formalize our approach to problems like this?**

What is an algorithm?

Algorithms

- Typical definition: an ordered set of executable steps to accomplish a task

- Algorithms are present everywhere

  Examples:

  - ✓cooking
  - ✓daily schedule
  - ✓walking
  - ✓etc.

Algorithms

- There are a variety of representations
  - Pseudocode
  - Flow-charts
  - Programs
  - Functions

- **Remember this?**

1. **Decide upon a metric** that is valuable to you (value, less weight, ratio of value to weight)
2. **Sort the items** by your metric
3. Add items to your result until you **hit your constraint**

Algorithms

- Building blocks of algorithms
  - Statements
  - Decision structures (if-else)
  - Repetition structures (for)
  - **Other algorithms (like sort)!**

Algorithms

- **Computer Science is the science of algorithms**

- 9 Algorithms book by John MacCormick- a set of algorithms you use every day, including:
  - Encryption
  - Page rank – web search (typical case: Google search engine)
  - Error correction

- But for now, we'll focus on the problem of **sorting lists**
  - Why? Think of its role in the Steam Sale problem
  - There are algorithms doing exactly the same thing (*Which one is better? What does "better" mean?*)

Sorting

Bubble Sort

- Input: a list of values

  6 5 3 1 8 7 2 4

- Output: a sorted list of values

  1 2 3 4 5 6 7 8

Bubble Sort: Animation

6 5 3 1 8 7 2 4

How does this work?

- Basic Idea:
  - in a sorted list neighbours are sorted (obvious)
  - Parse the list and swap elements that are not sorted.
  - Repeat until the list is sorted.

Bubble Sort: Pseudocode

```
func bubblesort ( var a as array)
    for i from 2 to N
        for j from 0 to N-2
            if a[j] > a[j+1]
                swap ( a[j], a[j+1] )
end func
```

<u>Exercise:</u> Implement a function that performs bubble sort on a list of numbers (15-20 minutes)

Translate pseudocode into a function!

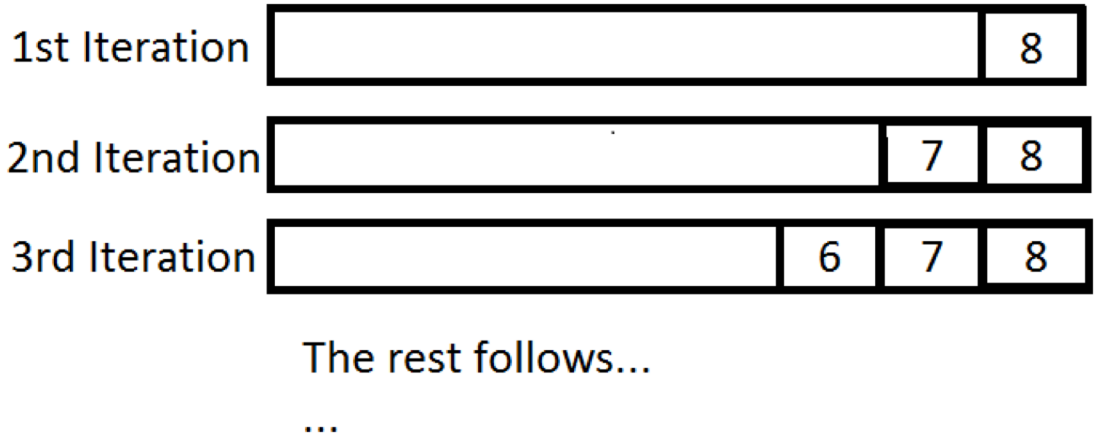
Bubble Sort: Optimizations?

Did we do it “efficiently”?

- Do we always have to make **n** -1 passes?
  - If no swap has not been done at a certain pass, we can assume the list is already sorted.

Bubble Sort: Optimizations?

- Illustrations



- In each pass you keep track of whether or not any pair of elements was swapped; If there's no swap, you can safely assume the list is sorted.

Bubble Sort: optimized

```
func bubblesort2( var a as array )
    for i from 2 to N
        swaps = 0
        for j from 0 to N - 2
           if a[j] > a[j + 1]
              swap( a[j], a[j + 1] )
              swaps = swaps + 1
        if swaps = 0
            break
end func
```

<u>Exercise:</u> Implement a function that performs bubble sort on a list of numbers
Translate pseudocode into a function!

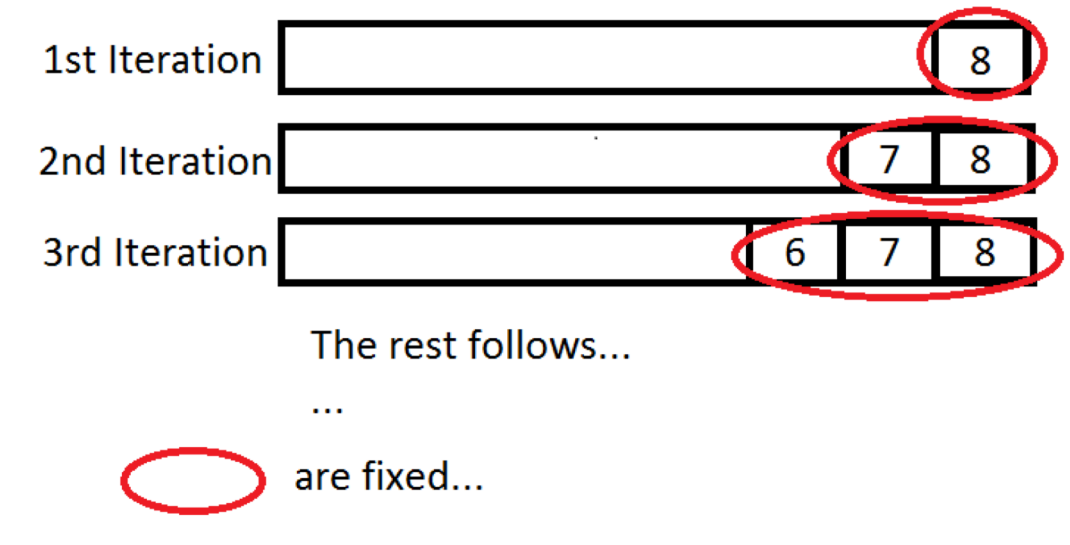
Bubble Sort: Optimizations?

Did we do it “efficiently”?

- Do we always have to make **n** -1 comparisons per pass?

Bubble Sort: optimized(cont.)

- Can we do better?
  - Once we have placed the largest number at the end of the list, this number won't be moved around anymore.



- We will reveal a further optimized version of bubble sort in the end.

Bubble Sort: optimized(cont.)

- Bubble Sort
  - Further optimized version

```
func bubblesort3( var a as array )
    for i from 1 to N
        swaps = 0
        for j from 0 to N - i
           if a[j] > a[j + 1]
              swap( a[j], a[j + 1] )
              swaps = swaps + 1
        if swaps = 0
            break
end func
```

  - After every current iteration, we cut down the next round of passes by 1, i.e. n passes for the initial iteration, and n-1 for the following iteration, until the last one with just 1 pass.
  - Overall, the total passes should be n*(n+1)/2

- We have a few lists listed below:
- 1 3 7 9 2 4 6 8
- 11 13 15 17 12 14 16 18
- What's in common among these lists?

- We have a few lists listed below:
- 1 3 7 9 2 4 6 8
- 11 13 15 17 12 14 16 18
- What's in common among these lists?
- 1 3 7 9 2 4 6 8
- 11 13 15 17 12 14 16 18

- We have a few lists listed below:
- 1 3 7 9 2 4 6 8
- 11 13 15 17 12 14 16 18
- What's in common among these lists?
  - These lists are composed by **sorted** sublists.
- We can do better than bubble sort them. *How?*

Merge Sort: Animation

6 5 3 1 8 7 2 4

Merge Sort

- Basic Idea:
  - Merging two sorted lists is trivial
  - Divide list into lists of single elements – which are by nature sorted (divide and conquer, *divide-et-impera*)
  - Merge adjacent lists to form sorted lists
  - Continue with adjacent lists until one list is obtained

1 3 7 9 | 2 4 6 8

Merge two lists

- 1 3 7 9 | 2 4 6 8

1. Make a new empty list Q = []
2. Compare the heads
3. Pop the smaller to the Q
4. Go back to 2 until both lists are empty

- Or: Q=[], during iteration i, min(h1,h2)→ Q[i]

Merge Sort: Pseudocode

```
func mergesort( var a as array )
     if ( n == 1 ) return a
     var l1 as arrary = a[0] ... a[n/2]
     var l2 as arrary = a[n/2+1] ... a[n]

     l1 = bubblesort( l1 )
     l2 = bubblesort( l2 )

     return merge( l1, l2 )
end func
```

```
func merge( var a as array, var b as array )
     var c as array

     while ( a and b have elements )
          if ( a[0] > b[0] )
               add b[0] to the end of c
               remove b[0] from b
          else
               add a[0] to the end of c
               remove a[0] from a
     while ( a has elements )
          add a[0] to the end of c
          remove a[0] from a
     while ( b has elements )
          add b[0] to the end of c
          remove b[0] from b
     return c
end func
```

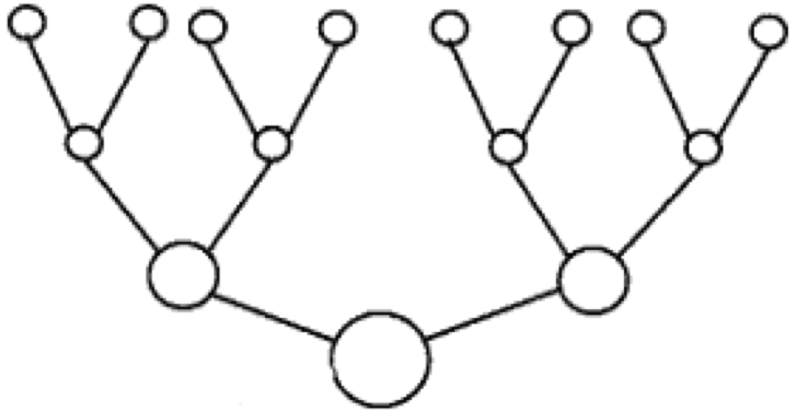
Merge Sort: Pseudocode

```
func mergesort( var a as array )
     if ( n == 1 ) return a
     var l1 as arrary = a[0] ... a[n/2]
     var l2 as arrary = a[n/2+1] ... a[n]

     l1 = bubblesort( l1 )
     l2 = bubblesort( l2 )

     return merge( l1, l2 )
end func
```

```
l1 = mergesort( l1 )
l2 = mergesort( l2 )
```

```
func merge( var a as array, var b as array )
     var c as array

     while ( a and b have elements )
          if ( a[0] > b[0] )
               add b[0] to the end of c
               remove b[0] from b
          else
               add a[0] to the end of c
               remove a[0] from a
     while ( a has elements )
          add a[0] to the end of c
          remove a[0] from a
     while ( b has elements )
          add b[0] to the end of c
          remove b[0] from b
     return c
end func
```

How can we visualize the steps?



For layer 1, you have 8 elements.

For layer 2, you cut down the numbers by half and acquire 4 elements.

For layer 3, similarly, 2 elements.

For layer 4, or the last layer, 1 element.

8*(1/2)^3=1→2^3=8→log(8)=3(for binary searches, 2 is the natural log base)

General case: **log(n)=x**, x refers to the steps you need to get to the designed element

Analysis: Merge vs. Bubble

- Which sort is better?
  - What does “better” mean?
- Some algorithms are “better” than others
- But in what way?
  - Usability
  - Simplicity / Readability
  - **Time**
  - **Space**
  - ‘Beauty’/elegance

Appendix B

- Bucket sort
  - Divide values into buckets, sort buckets then concatenate them
- Insertion sort
  - Build the sorted list by inserting one element at a time
- Heap sort
  - Uses trees (a type of data structure)
- Quick sort
  - Based upon pivots