Big-O and Sorting

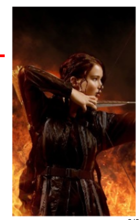
Lecture 17

1/63

1

Outline

- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort



2/63

2

Importance of Algorithm Analysis (1/2)

- **Performance** of algorithm refers to how quickly it executes and how much memory it requires
  - performance matters when amount of data gets large!
  - can analyze and observe performance, then revise algorithm to improve
- Algorithm analysis and sorting/searching data structures are crucial to computing and will be a central topic in CS0200!

3/63

3

Importance of Algorithm Analysis (2/2)

- Factors that affect performance
  - computing resources
  - language
  - implementation
  - size of data, denoted n
    - number of elements to be sorted in alphanumeric order
    - number of elements in `ArrayList` to iterate through
      - much faster to search through list of CS15 students than list of Brown students
- This lecture: a brief introduction to **Algorithm Analysis**!
- Goal: maximize efficiency and conserve resources

4

Outline

- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort

5

Performance of Algorithms

- How fast will recursive Fibonacci(N) run relative to N?
  - Fib(N) = Fib(N-1) + Fib(N-2);
    Fib(N-1) = Fib(N-2) + Fib(N-3), etc.
  - using recurrence relations, proportional to 2^N
- How fast will Towers of Hanoi run relative to the number of disks?
  - also proportional to 2^N for N disks
- How fast will N! run relative to N ?
  - N! will take exponentially longer as N increases, even faster than 2^N
  - not a problem with small N but we care about large inputs
- One algorithm could take 2 seconds while another could take 1 hour to accomplish the same task

6

Runtime (1/2)

- In analyzing an algorithm, runtime is the total number of times "the principal activity" of all steps in that algorithm is performed
  - varies with input and almost always grows with input size N
  - measured as a function of N (N, Nlog(N), N^2, 2^N, etc.)
- In most of computer science, we focus on worst case runtime
  - easier to analyze and important for unforeseen inputs
- Average case is what will typically happen; best case requires least amount of work and is the best situation you could have
  - average case is important; best case is interesting, but not insightful

7/63

7

Runtime (2/2)

- How to determine runtime?
  - inspect **pseudocode** (or actual code if it is small enough) and determine number of elementary operations in all statements executed by algorithm as a function of input size
  - allows us to **evaluate approximate speed** of an algorithm independent of hardware or software environment
  - memory use may be even more important than runtime for embedded devices

8/63

8

Elementary Operations

- Algorithmic "time" is measured in numbers of **elementary operations**
  - math (+, -, *, /, max, min, log, sin, cos, abs, ...)
  - comparisons (==, >, <=, ...)
  - function (method) **calls** and value **returns** (body of the method is separate)
  - variable assignment
  - variable increment or decrement
  - array **allocation** (declaring an array) and array **access** (retrieving an array from memory)
  - creating a new object (careful, object's constructor may have elementary ops too!)
- For purpose of algorithm analysis, assume each of these operations takes same time: **"1 operation"**
  - only interested in "**asymptotic performance**" for large data sets, i.e., as N grows large
    - small differences in performance don't matter when your data sets are billions or even trillions of items – e.g., indexing all the words on the WWW!

9/63

9

Example: Constant Runtime

```
public int addition(int[] a) {
    return a[0] + a[1]; //4 operations
}
```

- 4 operations – 1 addition, 2 array element retrievals, 1 return statement
- How many operations are performed if the input list had 1000 elements? 100,000?
- Runtime is **constant**

10/63

10

Example: Linear Runtime

```
//find max of a set of positive integers
public int maxElement(int[] a) {
   //assignment, 1 op
   int max = 0;
   //2 ops per iteration + 1 initial op to init i
   for (int i=0; i<a.length; i++){
      //2 ops per iteration
      if (a[i] > max) {
          //2 ops per iteration, sometimes
         max = a[i];
      }
   }
   //return, 1 op
   return max;
}
```

- Worst case varies proportional to the size of the input list: **6N + 3**
- How many operations if the array had 1,000 elements?
- We'll run the `for` loop proportionally more times as the input list grows
- Runtime increase is proportional to N, **linear**

11/63

11

Example: Quadratic Runtime

```
public void printPossibleSums(int[] a) {
    for (int i = 0; i < a.length; i++) { //2 ops per iteration
                                          //ignore op to init i
        for (int j = 0; j < a.length; j++) { //2 ops per iteration
            System.out.println(a[i] + a[j]); // 4 ops per iteration
        }
    }
}
```

- Requires about **$8N^2$** operations (it is okay to approximate!)
- Number of operations executed **grows quadratically!**
- If one element added to list, element must be added with every other element in list
- Notice that linear runtime algorithm on previous slide had only one `for` loop, while this quadratic one has two **nested** `for` loops, a typical **N^2** pattern

12/63

12

Big-O Notation

- Used to abstract from implementation by ignoring constants!
- **O(N)** implies runtime is linearly proportional to number of elements/inputs in the algorithm (constant operations per element)
  - (N elements) * (constant operations/element) = cN operations => O(N)
- **O(N^2)** implies each element is operated on N times
  - (N elements) * (N operations/element) = N^2 operations; cN^2 => O(N^2)
- **O(1)** implies that runtime does not depend on number of inputs
  - runtime is the same regardless of how large/small input size is
- Only consider "**asymptotic behavior**" i.e., when N >> 1
  - N is tiny when compared to N^2 for N >> 1

13/63

13

Big-O Constants

- **Important**: Only the largest N expression ***without constants*** matters
- We are not concerned about runtime with small numbers of data – we care about running operations on large amounts of inputs
  - $3N^2$ and $500N^2$ are both **O(N^2)** because the larger the input, the less the "500" and the "3" will affect the total runtime
  - N/2 is **O(N)**
  - $4N^2 + 2N$ is **O(N^2)**
- Useful sum for analysis:

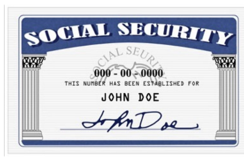
$$1 + 2 + 3 + \cdots + N = \sum_{k=1}^{N} k = N(N+1)/2, \text{ which is } \mathbf{O(N^2)}$$

14/63

14

Social Security Database Example (1/3)

- Hundreds of millions of people in the US have a number associated to them
- If 100,000 people are named John Doe, each has an individual SSN
- If the government wants to look up information they have on John Doe, they use his SSN



15/63

15

Social Security Database Example (2/3)

- Say it takes 10^{-4} seconds to perform a constant set of operations on one SSN
  - running an algorithm on 5 SSNs will take 5×10^{-4} seconds, and running an algorithm on 50 will only take 5×10^{-3} seconds
  - both are incredibly fast, difference in runtime might not be noticeable by an interactive user
  - this changes with large amounts of data, i.e., the actual SS Database

16/63

16

Social Security Database Example (3/3)

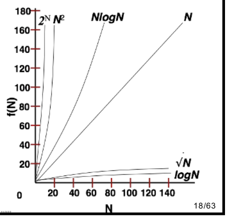
- Say we want to scale this algorithm to every SSN (300+ million)
  - to perform algorithm with **O(N)** on 300 million people will take **8.3 hours**
  - **$O(N^2)$** takes **285,000 years**
- With large amounts of data, **differences between O(N) and $O(N^2)$ are HUGE!**

17/63

17

Graphical Perspective (1/2) – Linear Plot

- f(N) on a small scale →

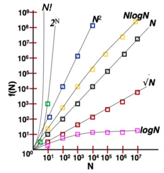


18/63

18

Graphical Perspective (2/2) – Log Plot

- f(N) on a larger scale →
- For 10 million items ($N = 10^7$)…
  - and $O(\log_{10} N)$ runtime, perform roughly 7 operations
  - and O(N) runtime, perform roughly 10 million operations
  - and $O(N^2)$ runtime, perform roughly 100 trillion operations
- really try to stay sub-quadratic!!



19/63

19

TopHat Question (1/3)

What is the big-O **runtime** of this algorithm?

```
public int sumArray(int[] array){
    int sum = 0;
    for (int i = 0; i < array.length; i++){
        sum = sum + array[i];
    }
    return sum;
}
```

A) O(N)   B) $O(N^2)$ C) O(1)   D) $O(2^N)$

20/63

20

TopHat Question (2/3)

What is the big-O **runtime** of this algorithm?

Consider the `getLetter()` *(or equivalent) method from TicTacToe:*

```
public String getLetter(){
    return this.letter;
}
```

A) O(N)   B) $O(N^2)$ C) O(1)   D) $O(2^N)$

21/63

21

TopHat Question (3/3)

What is the big-O **runtime** of this algorithm?

```
public int sumSquareArray(int[][] a){
    int sum = 0;
    for (int i = 0; i < a.length; i++){
        for (int j = 0; j < a[0].length; j++){
            sum = sum + a[j][i];
        }
    }
    return sum;
}
```

A) O(N) B) O(N^2) C) O(1) D) O(2^N)

22/63

22

Outline

- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort

23/63

23

Sorting

- We use runtime analysis to help choose the best algorithm to solve a problem
- Two common problems: **sorting** and **searching** through a list of objects
- We will analyze different **sorting** algorithms to find out which is fastest

24/63

24

Sorting – Social Security Numbers

- Consider an example where run-time influences your approach
- How would you **sort** every SSN in the Social Security Database in **increasing order?**
- Multiple known algorithms for sorting a list
  - these algorithms vary in their Big-O runtime

25/63

25

Bubble Sort (1/2)

- Iterate through sequence, comparing each element to its right neighbor
- Exchange/swap adjacent elements if necessary; largest element "bubbles" to the right
- End up with sorted sub-array on the right. Each time we go through the list, need to switch at least one item fewer than before

49 2 36 55 4 72 23
Before a pass

2	36	49	55	4	72	23

Middle of first pass

2	36	49	4	55	23	72

After one pass

26/63

26

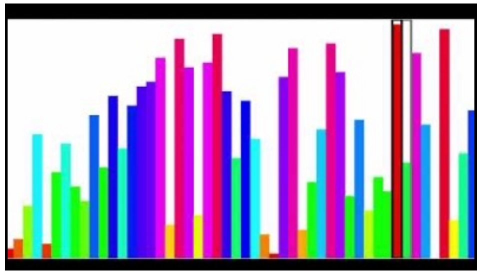
Bubble Sort (2/2)

- Iterate through sequence, comparing each element to its right neighbor
- Exchange adjacent elements if necessary; largest element "bubbles" to the right
- End up with sorted sub-array on the left. Each time we go through the list, need to switch at least one item fewer than before
- More efficient version: keep track of last largest element inserted so we don't have to go all the way over the right

```
int i = array.length;
boolean sorted = false;
while ((i > 1) && (!sorted)) {
    sorted = true;
    for(int j = 1; j < i; j++) {
        if (a[j-1] > a[j]) {
            int temp = a[j-1];
            a[j-1] = a[j];
            a[j] = temp;
            sorted = false;
        }
    }
    i--;
}
```

27/63

27



28

Bubble Sort - Runtime

Worst-case analysis:

- `while` loop iterates N-1 times
  - `while i > 1` and `i = array.length`
- Inner for loop iterates N-1 times
  - `j < i` and `i = array.length`

Total:

- $(N-1)(N-1) \approx O(N^2)$

Remember!
Small operations and constants don't majorly affect runtime, so we can ignore them when calculating big-O!

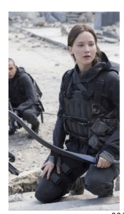
```
int i = array.length;
boolean sorted = false;
while ((i > 1) && (!sorted)) {
    sorted = true;
    for(int j = 1; j < i; j++) {
        if (a[j-1] > a[j]) {
            int temp = a[j-1];
            a[j-1] = a[j];
            a[j] = temp;
            sorted = false;
        }
    }
    i--;
}
```

29/63

29

Outline

- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort

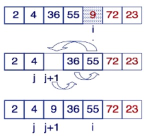


30/63

30

Insertion Sort (1/2)

- Like inserting a new card into a partially sorted hand by bubbling to the left in a sorted subarray
  - close to bubble sort but less brute force because we don't start always from the rightmost entry
- Add one element `a[i]` at a time
- Find proper position, `j + 1`, to the left by swapping with neighbors on the left (`a[i-1]`, `a[i-2]`, …, `a[j+1]`) to the right, until `a[j] < a[i]`
- Move `a[i]` into vacated `a[j+1]`
- After iteration `i < a.length`, original `a[0]` … `a[i]` are in sorted order, but not necessarily in final position, depending on what comes after `a[i]`

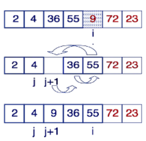


31/63

31

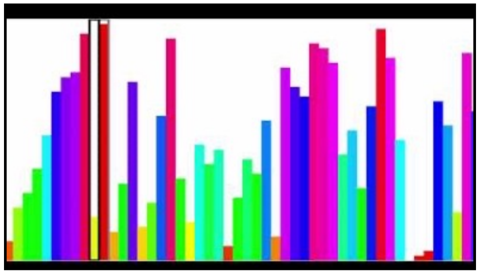
Insertion Sort (2/2)

```
for (int i = 1; i < a.length; i++) {
    int toInsert = a[i];
    int j = i-1;
    while ((j >= 0) && (a[j] > toInsert)){
       a[j+1] = a[j];
       j--;
    }
    a[j+1] = toInsert;
  }
```



32/63

32



33

Insertion Sort - Runtime

```
for (int i = 1; i < a.length; i++) {
    int toInsert = a[i];
    int j = i-1;
    while ((j >= 0) && (a[j] > toInsert)){
       a[j+1] = a[j];
       j--;
    }
    a[j+1] = toInsert;
 }
```

- `while` loop inside our `for` loop
  - `while` loop calls N-1 operations
  - `for` loop calls the `while` loop N times
- **O(N²)** because we have to call on a `while` loop with ~N operations N different times
- Reminder: **constants do NOT matter with Big-O!**

34/63

34

Outline

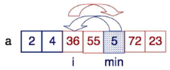
- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort

35/63

35

Selection Sort (1/2)

- Find smallest element and put it in a[0]
- Find 2nd smallest element and put it in a[1], etc.
- Less data movement (no bubbling movement)



36/63

36

Selection Sort (2/2)

What we want to happen:

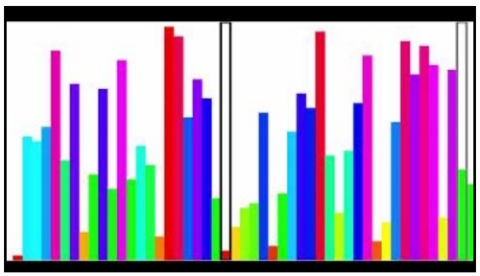
```
int n = a.length;
for (int i = 0; i < n; i++) {
  find minimum element a[min]
  in subsequence a[i...n-1]
  swap a[min] and a[i]
}
```

a	2	4	36	55	5	72	23
			i		min		

```
int n = a.length;
for (int i = 0; i < n-1; i++) {
  int min = i;
  for (int j = i + 1; j < n; j++) {
     if (a[j] < a[min]) {
        min = j;
     }
  }
  temp = a[min];
  a[min] = a[i];
  a[i] = temp;
}
```

37/63

37



38

Selection Sort - Runtime

- Most executed instructions are in if statement in inner for loop
- Each instruction is executed (N-1) + (N-2) + ... + 2 + 1 times
- Time Complexity: **$O(N^2)$**
  - nested loops!

```
for (int i = 0; i < n-1; i++) {
  int min = i;
  for (int j = i + 1; j < n; j++) {
     if (a[j] < a[min]) {
        min = j;
     }
  }
  temp = a[min];
  a[min] = a[i];
  a[i] = temp;
}
```

39/63

39

Comparison of Basic Sorting Algorithms

- Differences in **Best-** and **Worst-case** performance are based on current order of input before sorting
- Selection Sort wins on data movement
- For small data, even the worst sort – Bubble (based on comparisons and movements) – is fine!

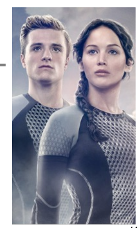
		Selection	Insertion	Bubble
Comparisons of data	Best	$n^2/2$	n	n
	Average	$n^2/2$	$n^2/4$	$n^2/4$
	Worst	$n^2/2$	$n^2/2$	$n^2/2$
Movements of data	Best	0	0	0
	Average	n	$n^2/4$	$n^2/2$
	Worst	n	$n^2/2$	$n^2/2$

40/63

40

Outline

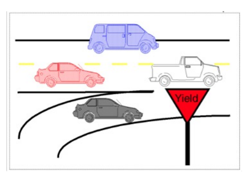
- Importance of Algorithm Analysis
- Runtime
- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort



41/63

41

Merge Sort



42/63

42

Recap: Recursion (1/2)

- Recursion is a way of solving problems by breaking them down into smaller sub-problems, and using results of sub-problems to find the answer
- Example: You want to determine what row number you're sitting in, but you can only get information by asking the people in front of you
  - they also don't know what row they're in, must ask people in front of them
  - people in first row know that they're row 1, since there is no row in front (base case)
  - they tell people behind them, who know that they're 1 behind row 1, so they are row 2, etc.
  - this "unwinds" the recursion

43/63

43

Recap: Recursion (2/2)

```
public int findRowNumber(Row myRow) {

    if (myRow.getRowAhead() == null) { // base case!
        return 1;
    } else {
        // recursive case - ask the row in front
        int rowAheadNum = this.findRowNumber(myRow.getRowAhead());

        // my row number is one more than the row ahead's number
        return rowAheadNum + 1;
    }
}
```

44/63

44

Recursion (Top Down) Merge Sort (1/7)

- Let's say you don't know how to sort n elements, but you have a friend who can sort any number less than n. How can you use the results to do your work? (like auditorium row number problem)
  - one answer is to sort n-1, then just slot the last element into the sorted order (insertion sort)
  - another answer is to pick the smallest single entry, then give remaining elements to your friend to sort and add your element to the beginning of her results (selection sort)
  - what if your friend can only sort things of size n/2 or smaller? She can sort the two pieces... can we quickly make a sorted list from what's left? (merge sort!)

45/63

45

Recursion (Top Down) Merge Sort (2/7)

- **Partition** sequence into two sub-sequences of N/2 elements

- Recursively **partition** and **sort** each sub-array

- **Merge** the sorted sub-arrays

46/63

46

Recursion (Top Down) Merge Sort (3/7)

- **Partition** sequence into two sub-sequences of N/2 number of elements
- Recursively **partition** and **sort** each sub-array
- **Merge** the sorted sub-arrays

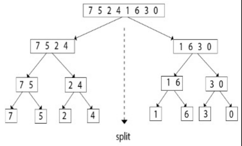


Figure: Merge sort divide phase

47/63

47

Recursion (Top Down) Merge Sort (4/7)

```
public class Sorts {
  public ArrayList<Integer> mergeSort(ArrayList<Integer> list) {
     if (list.size() == 1) {
       return list;
     }
     int middle = list.size() / 2;
     ArrayList<Integer> left =
          this.mergeSort(list.subList(0, middle));
     ArrayList<Integer> right =
          this.mergeSort(list.subList(middle, list.size()));
     return this.merge(left, right);
  }
  //code for merge() coming next!
}
```

ArrayList list is the sequence to sort, a sequence of ints

Base case: return the list when you get to its last element

Else, recur on both halves of the list and merge the sorted lists

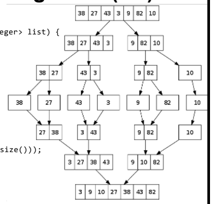
48/63

48

Recursion (Top Down) Merge Sort (5/7)

```
public class Sorts {

  public ArrayList<Integer> mergeSort(ArrayList<Integer> list) {
     if (list.size() == 1) {
       return list;
     }
     int middle = list.size() / 2;
     ArrayList<Integer> left =
          this.mergeSort(list.subList(0, middle));
     ArrayList<Integer> right =
          this.mergeSort(list.subList(middle, list.size()));
     return this.merge(left, right);
  }
  //code for merge() coming next!
}
```



49/63

49

Recursive (Top Down) Merge Sort (6/7)

```
public ArrayList merge(ArrayList<Integer> A, ArrayList<Integer> B) {
   ArrayList<Integer> result = new ArrayList<Integer>();
   int aIndex = 0;
   int bIndex = 0;
   while (aIndex < A.size() && bIndex < B.size()) {
      if (A.get(aIndex) <= B.get(bIndex)) {
         result.add(A.get(aIndex));
         aIndex++;
      } else {
         result.add(B.get(bIndex));
         bIndex++;
      }
   }
   if (aIndex < A.size()) {
          result.addAll(A.subList(aIndex, A.size()));
     }
     if (bIndex < B.size()) {
          result.addAll(B.subList(bIndex, B.size()));
     }
     return result;
  }
```

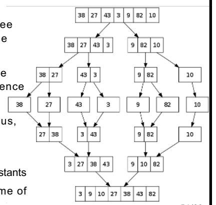
- Add elements from the two sequences in **increasing order**
- If there are elements left that you haven't added, **add the remaining elements** to your result

50/63

50

Recursive (Top Down) Merge Sort (7/7)

- Recursion to get down to base case is just halving each subarray: **$O(\log_2 N)$**
- Unwinding the recursion: Each level of the tree performs **N operations** to **merge** and **sort** the subproblems below it
- Each time you **merge**, you must handle all the elements in the sub-arrays you're merging, hence **$O(N)$**
- There are **$\log_2 N$** number of merge passes, thus, **$O(\log_2 N) + O(N)(\log_2 N) = O(N)(\log_2 N)$**
  - way better than **$O(N^2)$**
  - drop base (2) and say **$O(N\log N)$**, ignore constants
- Learn much more about how to find the runtime of these types of algorithms in CS200!

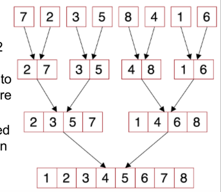


51/63

51

Iterative (Bottom Up) Merge Sort

- Merge sort can also be implemented **iteratively**… non-recursive!
- Loop through array of size N, sorting 2 items each. Loop through the array again, combining the 2 sorted items into sorted item of size 4. Repeat, until there is a single item of size N!
- Number of **iterations** is **$\log_2 N$**, rounded up to nearest integer. 1000 elements in the list, **only 10** iterations!
- Iterative merge sort avoids the nested method invocations caused by recursion!



52/63

52

53

Comparing Sorting Algorithms

Bubble Sort – $O(N^2)$ Insertion Sort – $O(N^2)$ Merge Sort - $(N\log_2 N)$

Click here to download interactive sorter

54/63

54

TopHat Question

Which sorting algorithm that we have looked at is the fastest (in terms of **worst-case** runtime)?

A. Bubble Sort
B. Insertion Sort
C. Merge Sort
D. Selection Sort

55/63

55

That's It!

- Runtime is a very important part of algorithm analysis!
  - worst case runtime is what we generally focus on
  - know the difference between constant, linear, and quadratic run-time
  - calculate/define runtime in terms of Big-O Notation
- Sorting!
  - runtime analysis is very significant for sorting algorithms
  - types of simple sorting algorithms - bubble, insertion, selection, merge sort
  - fancier sorts perform even better, but tough to analyze, e.g., QuickSort
  - different algorithms have different performances and time complexities

56/63

56

What's next?

- You have now seen how different approaches to solving problems can dramatically affect speed of algorithms
  - this lecture utilized arrays and loops to solve most problems
- Subsequent lectures will introduce more **data structures** beyond arrays and arraylists that can be used to handle collections of data
- We can use our newfound knowledge of algorithm analysis to strategically choose different data structures to further speed up algorithms!

57/63

57

Announcements

- DoodleJump late deadline tomorrow 11/3 @ 11:59pm
- DoodleJump Code Debriefs Coming Up
  - Keep an eye on your email to see if you were selected
  - If you are selected and miss this debrief you will receive a minus four deduction on your final grade!!!!
- Tetris out Saturday!!!
  - you do NOT want to procrastinate on this assignment!
  - the earlier you start, the shorter the lines at debugging hours ☺
  - Please reference the collaboration policy when working on Solo Tetris!!

58/63

58

Socially Responsible Computing

Dark & Addictive Design

CS15 Fall 2022

59

Definition

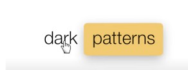


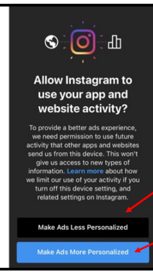
Image source: Evan Puschak, 2018

Dark patterns are features of interactive design crafted to **trick users into doing things they might not wish to do**, but which **benefit the business in question**.

Term coined by Harry Brignull (UX Specialist) in 2010

60

Nudging users to give up privacy



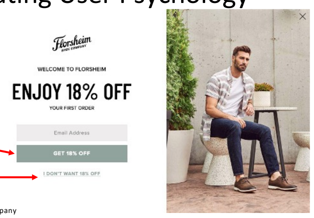
Using terms like “personalized” instead of “targeting” or “tracking”

More intuitive to tap on the blue box

Source: Meta

61

Manipulating User Psychology



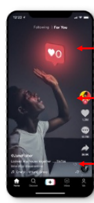
More intuitive to tap the big, dark box

Manipulating user psychology

Source: Florsheim Shoe Company

62

Addictive patterns in action: TikTok



Variable rewards/slot machine tactic: not knowing what you will get when you swipe

Infinite scrolling/Auto-loop feature encourages you to passively engage with the app & makes it hard to stop

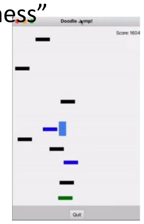
Restricts user choice by limiting the ways in which you can interact with the app, so you can only scroll

Hick's Law: The more choices a user has, the less likely they are to make one

Source: Ankit Sherke Design

63

Infinite scrolling to stimulate “stickiness”

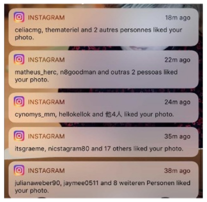


Source: @callmetochi on TikTok

64

Impact of addictive patterns on the brain

Push-notifications and “likes” increase **dopamine,** the same chemical released with drug/alcohol usage



Source: Apple Tool Blox blog

Abnormal White Matter Integrity in Adolescents with Internet Addiction Disorder: A Tract-Based Spatial Statistics Study

Unravelling the Enigma: How Behavioural Addictions Alter the Developing Child Brain?

Source: NIH (2012, 2023)

Internet addiction in children leads to:
- inhibited capabilities in brain regions responsible for reward processing and inhibitory control
- impairments in the network that allows for communication between different areas of your brain

65

Meta Lawsuit

TECHNOLOGY

States Sue Meta Alleging Harm to Young People on Instagram, Facebook

More than 40 attorneys general filing suits in federal and state courts seek changes to product features

Source: WSJ (October 2023)

- States accuse Meta of damaging mental health with “dopamine-manipulating” features, including likes and infinite scrolling
- As evidence, plaintiffs cite internal Meta documents made public by whistleblower Frances Haugen
- Legal experts say Meta is likely to invoke Section 230

66

Regulation

FTC Report Shows Rise in Sophisticated Dark Patterns Designed to Trick and Trap Consumers

Tactics Include Disguised Ads, Difficult-to-Cancel Subscriptions, Buried Terms, and Tricks to Obtain Data

September 15, 2022

Source: FTC.gov (2022)

Source: Center for Humane Technology (2022)

67

FTC Regulation

$245 million FTC settlement alleges Fortnite owner Epic Games used digital dark patterns to charge players for unwanted in-game purchases

By: Lesley Fair | December 19, 2022

Source: FTC.gov (2022)

FTC Sends Nearly $100 Million in Refunds to Vonage Consumers Who Were Trapped in Subscriptions By Dark Patterns and Junk Fees

October 30, 2023

Source: FTC.gov (2023)

68

Issue with defining dark patterns

"All design has a level of persuasion to it. The difference is, if you're designing to trick people, you're an asshole."

Victor Yocco, author of *Design for the Mind: Seven Psychological Principles of Persuasive Design (2016)*

69

Join Dash!

What is Dash?

- MERN stack web application (MongoDB, Express, React, Node.js), hypertext/hypermedia system
- produce, annotate, and consume digital documents

What would you be doing?

- This semester:
  - using Dash and providing feedback
  - becoming familiar with the system, codebase, and technologies used
- Winter break:
  - complete the starter project to join as full member
- Next semester:
  - work on Dash as an independent study, building your very own feature!

Interest form:

70/63

70