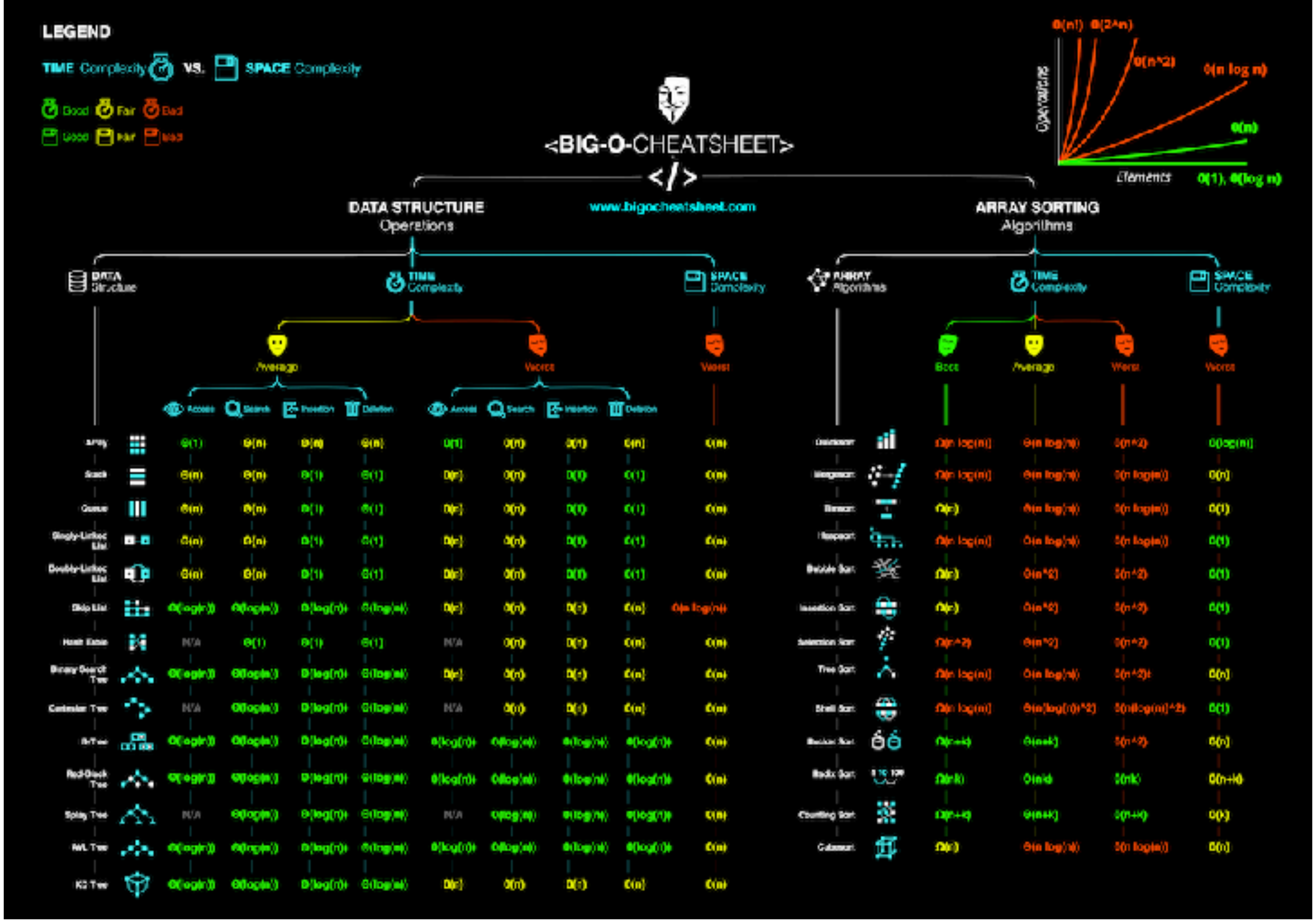

CSCI 241

Lecture 26

Review of Runtime Analysis Techniques

Max-flow / Min-cut

Announcements

- Final Exam: Study guide is updated with objectives from the second half of the quarter
- Study tips:

    1. Start now. You’ve taken 7 quizzes and 1 exam. There are 8 days between now and the exam.

    2. ABCD questions and other in-class assessments: these resemble the “easy” points on the exam.

    3. Flipping through slides, nodding, and pensively saying “ah yes, I remember this” is **not** a good study strategy. **Solve problems.** If you run out of problems, make up more.

Announcements

- No new material this week will be on the exam.
- There will be in-class exercises every day this week.
- Some fun advanced topics will be introduced at a high level.

Goals

- Review the following techniques we've used for runtime analysis up to this point:
  - Counting operations
  - Aggregate analysis
- Be able to analyze the runtime of Prim's algorithm as implemented in class.
- Be able to analyze the runtime of Dijkstra's algorithm as implemented in A4.

Runtime Analysis: Review

- Why? We want a measure of performance that is

  - **Independent** of what computer we run it on.

    Solution: count **operations** instead of clock time.

  - Dependence on **problem size** is made explicit.

    Solution: express runtime as a function of **n**
    (or whatever variables define problem size)

  - **Simpler** than a raw count of operations and focuses on performance on **large problem sizes**.

    Solution: ignore constants, analyze **asymptotic** runtime.

Runtime Analysis: Review

- How?

  1. Count the number of primitive (constant-time) operations that occur over the entire execution of the algorithm.

  2. Drop constants and lower-order terms to find the **asymptotic runtime class**.

Counting Operations

What’s a constant-time operation?

- Anything that doesn’t depend on the input size:
  - Reading/writing from/to a variable or array location.
  - Evaluating an arithmetic or boolean expression.
  - Returning from a method.

Counting Operations

What's a constant-time operation?

- Anything that doesn't depend on the input size:
  - Reading/writing from/to a variable or array location.
    ```
    int i = 2; int b = 4; a[i] = b;
    ```
  - Evaluating an arithmetic or boolean expression.
    ```
    int i = 0; int j = i+4; int k = i*j;
    ```
  - Returning from a method. `return k;`

Key intuition:

- These don't take identical amounts of time, but the times are within a **constant factor** of each other.
- Same for running the **same** operation on a **different** computer.

Counting Operations

What's a constant-time operation?

- Anything that doesn't depend on the input size:
  - Creating a new object? Depends:
    - if you only have to initialize O(1) memory, it's O(1)

      ```
      Node n = new Node();
      ```

    - If you have to initialize O(n) memory, it's O(n).

      ```
      int[] a = new int[n];
      ```

      java has to set all n entries to 0

      ```
      AList<Node> a = new AList<Node>(n);
      ```

      constructor creates an array of size n!

Counting Operations

What's **not** a constant-time operation?

- Anything that **does** depend on the input size:
  - Looping over all values in an array of size n.
  - Recursing over a tree of height log(n).
  - Searching a graph of |V| nodes and |E| edges.
  - Most nontrivial algorithms / data structure operations we've covered in this class.

Counting Operations

What happens when the number of times executed is variable / depends on the data?

- We have to specify whether we want worst-case, average-case (aka expected-case), or best-case runtime.

```
public int findMax(int[] a) {
  int currentMax = a[0];
  for (int i = 1; i < a.length; i++) {
    if (currentMax < a[i]) {
      currentMax = a[i];
    }
  }
}
```

times executed depends on contents of a!

Counting Operations

What happens when the number of times executed is variable / depends on the data?

- Worst-case is usually the important one, with notable exceptions for algorithms that beat asymptotically faster algorithms in practice:
  - Quicksort generally beats Mergesort in practice.
  - HashMaps generally beat TreeMaps unless keys need to be sorted.

Counting Strategies:
1. Simple counting

```
/** A singly linked list node */
public class Node {
  int value;
  Node next;
  public Node(int v) {
    value = v;
  }
}

/** Insert val into the list in after pred.
 * Precondition: pred is not null */
public void addAfter(Node pred, int val) {
  Node newNode = new Node(val);   ———— 1
  new_node.next = pred.next;      ———— 1
  pred.next = newNode;            ———— 1
}
```

Counting Strategies:
1. Simple counting - for loop

```
for (int i = 0; i < n; i++) {
    loopBody(i);
}

// is equivalent to:

int i = 0;            ———— 1
while (i < n) {       ———— 1 per iteration
  loopBody(i);        ———— 1 per iteration
  i++;                ———— 1 per iteration
}
```

How many iterations?

i takes on values 0..n, of which there are n.

Counting Strategies:
1. Simple counting - for loop

```
for (int i = 0; i < n; i++) {
    loopBody(i);
}
```

Total runtime:

```
// is equivalent to:
```

1 + 2n + n*[runtime of loopBody]

```
int i = 0;            ——— 1
while (i < n) {       ——— n
   loopBody(i);       ——— n * runtime of loopBody
   i++;               ——— n
}
```

How many iterations?

i takes on values 0..n, of which there are n.

Counting Strategies:
1. Simple counting

```
/** An ArrayList-like growable array. */
public class AList<T> {
  int size;
  T[] a;

  // other methods
  public void addFirst(T val) {
    growIfNeeded(size+1);
    for (int i = 0; i < size; i++) {
      a[size-i] = a[size-i-1];
    }
    a[0] = val;
  }
}
```

Let n = size.

growIfNeeded(size+1): 1 or n

for loop: happens n times

a[size-i] = a[size-i-1]: 1

a[0] = val: 1

Worst-case: n + n + 1 = 2n+1, which is O(n)

Best-case: 1 + n + 1 = n+2, which is O(n)

Counting Strategies:
2. Counting in Aggregate

```
/** sorts A using LSD radix sort */
public void radixSortQueue(int[] A) {
  for (int d = 0; d < 10; d++) {
    for (int i = 0; i < A.length; i++) {
      int key = getDigit(A[i], d);          O(1)
      buckets[key].add(A[i]);               O(1)
    }
    int k = 0;                              1
    for (int i = 0; i < 10; i++) {
      while (buckets[i].peek() != null) {   O(1)
        A[k] = buckets[i].remove();         O(1)
        k++;                                1
      }
    }
  }
}
```

Counting Strategies: 2. Counting in Aggregate

```
/** sorts A using LSD radix sort */
public void radixSortQueue(int[] A) {
  for (int d = 0; d < 10; d++) {                    *10
    for (int i = 0; i < A.length; i++) {
      int key = getDigit(A[i], d);                   O(1)
      buckets[key].add(A[i]);                        O(1)
    }
    int k = 0;                                       1
    for (int i = 0; i < 10; i++) {
      while (buckets[i].peek() != null) {            O(1)
        A[k] = buckets[i].remove();                  O(1)
        k++;                                         1
      }
    }
  }
```

Counting Strategies:
2. Counting in Aggregate

```
/** sorts A using LSD radix sort */
public void radixSortQueue(int[] A) {
  for (int d = 0; d < 10; d++) {                *10
    for (int i = 0; i < A.length; i++) {        *n
      int key = getDigit(A[i], d);              O(1)
      buckets[key].add(A[i]);                   O(1)
    }
    int k = 0;                                  1
    for (int i = 0; i < 10; i++) {
      while (buckets[i].peek() != null) {       O(1)
        A[k] = buckets[i].remove();             O(1)
        k++;                                    1
      }
    }
  }
}
```

Counting Strategies: 2. Counting in Aggregate

```
/** sorts A using LSD radix sort */
public void radixSortQueue(int[] A) {
  for (int d = 0; d < 10; d++) {                  *10
    for (int i = 0; i < A.length; i++) {          *n
      int key = getDigit(A[i], d);                O(1)
      buckets[key].add(A[i]);                     O(1)
    }
    int k = 0;                                    1
    for (int i = 0; i < 10; i++) {
      while (buckets[i].peek() != null) {         O(1)
        A[k] = buckets[i].remove();               O(1)
        k++;                                      1
      }
    }
  }
```

How many times is this actually done?

Counting Strategies:
2. Counting in Aggregate

```
/** sorts A using LSD radix sort */
public void radixSortQueue(int[] A) {
  for (int d = 0; d < 10; d++) {                *10
    for (int i = 0; i < A.length; i++) {        *n
      int key = getDigit(A[i], d);              O(1)
      buckets[key].add(A[i]);                   O(1)
    }
    int k = 0;                                  1
    for (int i = 0; i < 10; i++) {
      while (buckets[i].peek() != null) {       O(1)
        A[k] = buckets[i].remove();             O(1)
        k++;                                    1
      }
    }
  }
```

How many times is this actually done?

n elements went into buckets, so only n elements can be removed. The whole red box does 3n ops.

Counting Strategies: 2. Counting in Aggregate

```
/** sorts A using LSD radix sort */
public void radixSortQueue(int[] A) {
  for (int d = 0; d < 10; d++) {
    for (int i = 0; i < A.length; i++) {
      int key = getDigit(A[i], d);
      buckets[key].add(A[i]);
    }
    int k = 0;
    for (int i = 0; i < 10; i++) {
      while (buckets[i].peek() != null) {
        A[k] = buckets[i].remove();
        k++;
      }
    }
  }
}
```

*10

*n

O(1)

O(1)

1

O(n)

Counting Strategies:
2. Counting in Aggregate

```
/** sorts A using LSD radix sort */
public void radixSortQueue(int[] A) {
  for (int d = 0; d < 10; d++) {                  *10
    for (int i = 0; i < A.length; i++) {
      int key = getDigit(A[i], d);
      buckets[key].add(A[i]);
    }                                          O(n)
    int k = 0;                                 1
    for (int i = 0; i < 10; i++) {
      while (buckets[i].peek() != null) {
        A[k] = buckets[i].remove();
        k++;
      }                                        O(n)
    }
  }
}
```

Overall: 10 * O(n) + 1 + O(n) => O(n)

Analyzing Prim’s Algorithm