10/19/2020

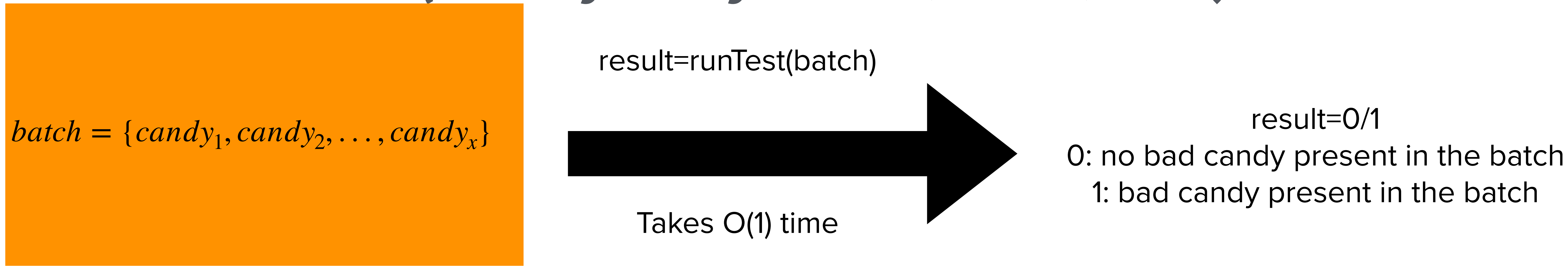
CS141: DISCUSSION WEEK 3

MASTER THEOREM AND AVERAGE CASE ANALYSIS

TABLE OF CONTENTS

HOMEWORK 1 QUESTION 3

- Input: n candies, some good, some bad
  - Number of bad candies: unknown but we guess it's small (Company still in business!)
- Want: Identify and get rid of bad candies
- Have: A bad candy detector that can run tests on batches of candy -> signature of the test: *boolean runTest(CandyArray batch, start, end)*



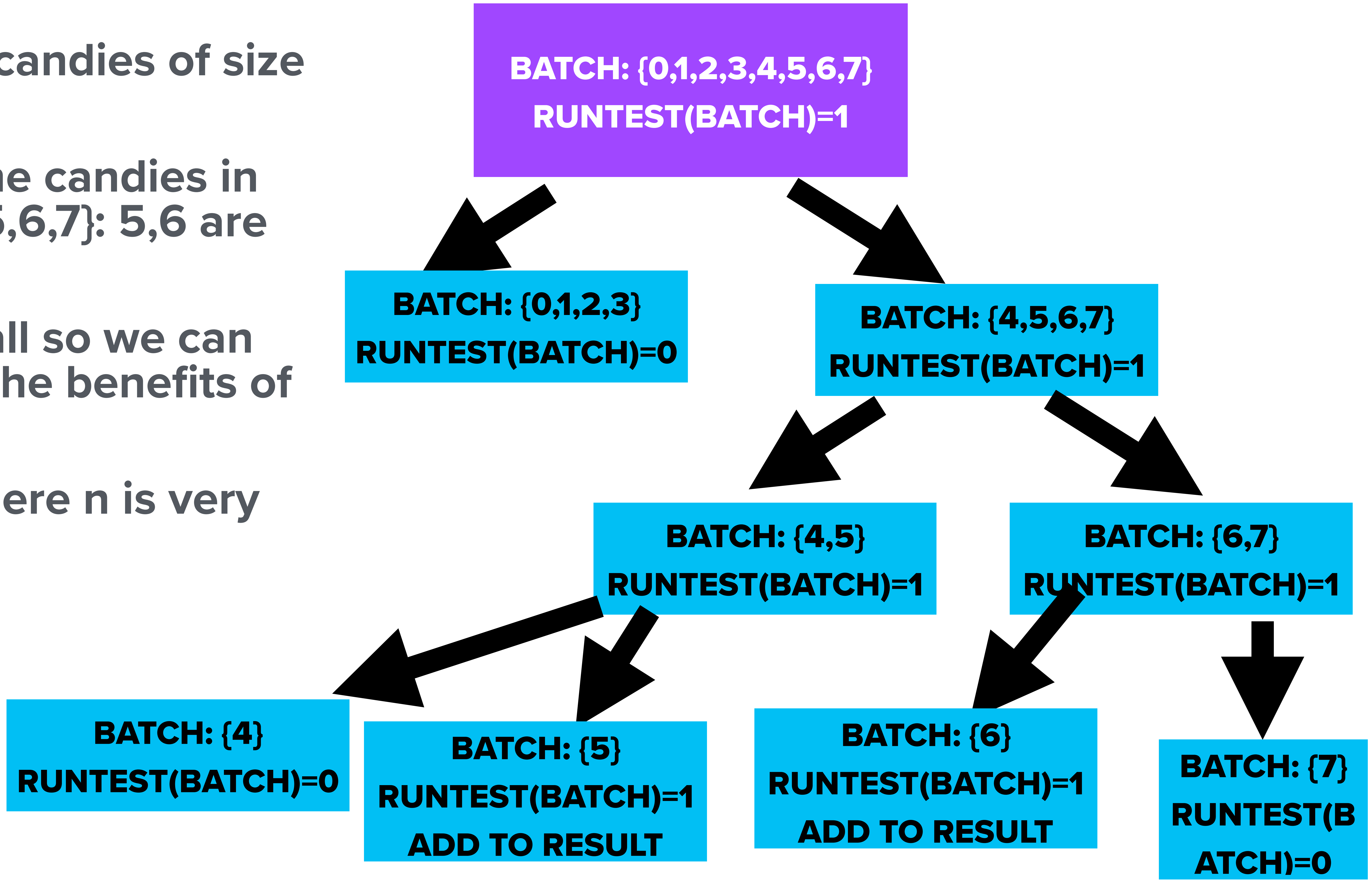
  - If test outputs 1, we only know there exists at least 1 bad candy in the batch; don't know how many

HW1-Q3: SOLUTION PART 1

```
Array resultArray; \\empty candy array
void findBadCandies(candyArray, start, end)
    if(start==end) \\we are just checking 1 candy
        if(runTest(CandyArray, start, end))\\it is bad
            resultArray.add(candyArray)
        return
    boolean val = runTest(candyArray, start, end)
    if(val)\\there are some bad candies in the batch, make two recursive calls of input size half
        findBadCandies(candyArray, start, (start+end)/2)
        findBadCandies(candyArray, (start+end)/2+1, end)
    if(!val)\\there are not bad candies in the batch
        return
```

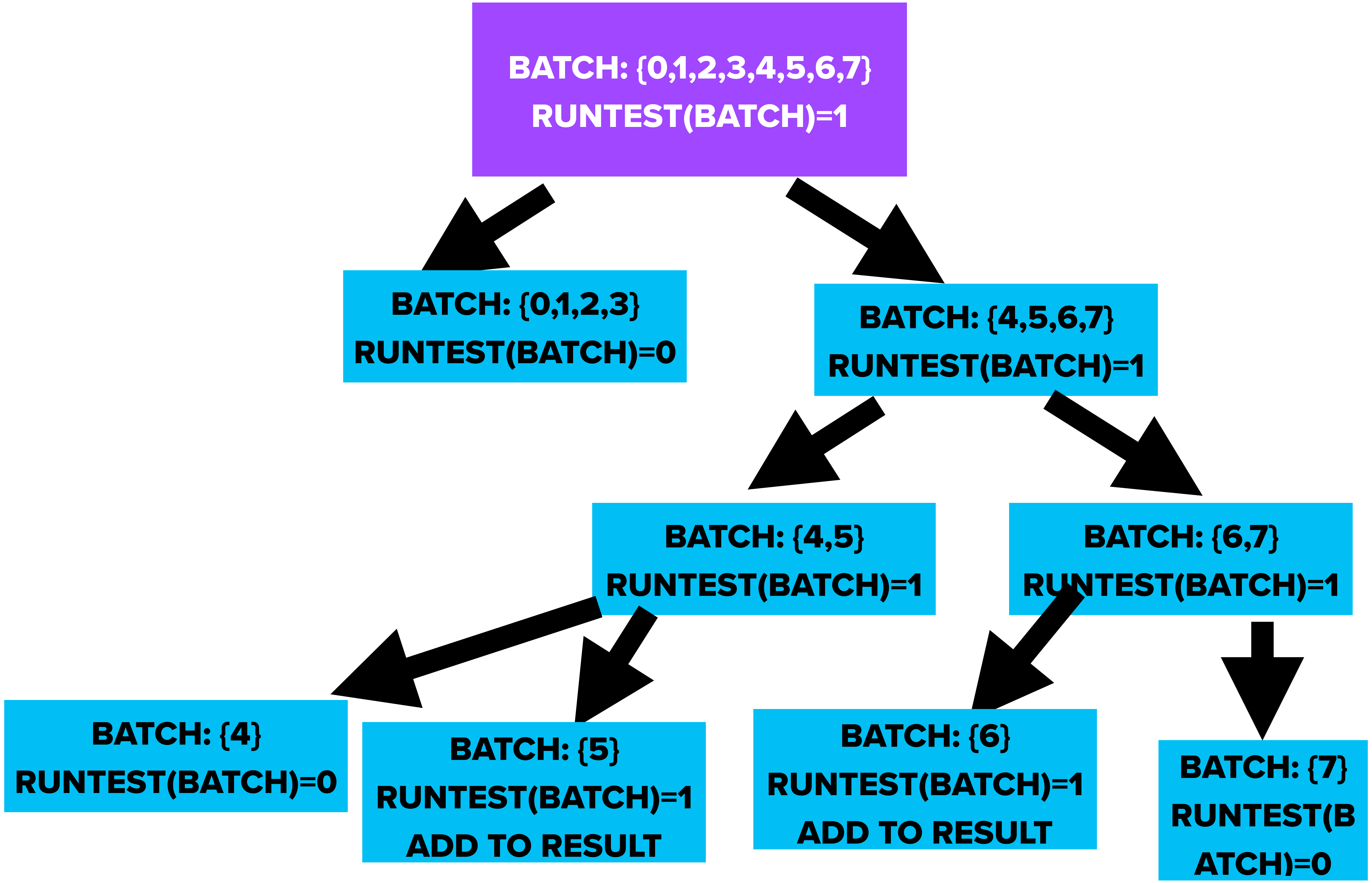
VISUALIZATION OF HW1-Q3:PART1

- Assume we have a batch of candies of size 8.
- Use numbers to represent the candies in the batch: batch= {0,1,2,3,4,5,6,7}: 5,6 are bad
- In this example n is very small so we can visualize. But, we don't see the benefits of this algorithm that well.
  - Think of the scenarios where n is very big!



HW1-Q3: PART 2

- Think of the recursive computation as a tree
- Traverse tree from root to leaf only if the candy in the leaf is bad
- Going from root to one leaf takes logn time
- If number of bad candies is constant: traversal takes c*logn time => O(logn)



HW1-Q3: PART 3

```
Array resultArray; \\empty candy array
void findBadCandies(candyArray, start, end)
    if(start==end) \\we are just checking 1 candy
        if(runTest(CandyArray, start, end))\\it is bad
            resultArray.add(candyArray)
        return
    boolean val = runTest(candyArray, start, end)
    if(val)
        findBadCandies(candyArray, start, (start+end)/2)
        findBadCandies(candyArray, (start+end)/2+1, end)
    if(!val)
        return
```

- If all candies are bad, go down to the leaf for each candy
- Two recursive calls with half of the input size
- During each recursive call, do O(1) amount of work
- Recurrence relation: $T(n)=2T(n/2)+1$
- Solve it with master theorem!

MASTER THEOREM

- A powerful method to solve a common type of recurrence relations
- Can be applied to recurrence relations of the form:
  - $T(n) = aT(n/b) + f(n),\ a \geq 1,\ b > 1,\ f$:asymptotically positive
  - Let $y = log_b a$ and constant $k \geq 0.$
- Case 1: $f(n) = O(n^{y'})$ for $y' < y \Rightarrow T(n) = \Theta(n^y)$: more work as we keep dividing the input
- Case 2: $f(n) = \Theta(n^y log^k n) \Rightarrow T(n) = \Theta(n^y log^{k+1} n)$: same amount of work as we keep dividing the input
- Case 3: $f(n) = \Omega(n^{y'})$ for $y' > y$ and $af(n/b) \leq cf(n) \Rightarrow T(n) = \Theta(f(n))$:less work as we keep dividing the input

GOING BACK TO HW1-Q3:PART 3

- $T(n) = aT(n/b) + f(n),\ a \geq 1,\ b > 1,\ f$:**asymptotically positive**
- **Let** $y = log_b a$ **and constant** $k \geq 0$.
- **Case 1:** $f(n) = O(n^{y'})$ **for** $y' < y \Rightarrow T(n) = \Theta(n^y)$
- **Case 2:** $f(n) = \Theta(n^y log^k n) \Rightarrow T(n) = \Theta(n^y log^{k+1} n)$
- **Case 3:** $f(n) = \Omega(n^{y'})$ **for** $y' > y$ **and** $af(n/b) \leq cf(n) \Rightarrow T(n) = \Theta(f(n))$

- **Recurrence relation of our candy solving problem: T(n)=2T(n/2)+1**
- **Which case does it belong to? Answer: a=2,b=2,f(n)=1, y=1.**
  $f(n) = 1 = O(n^{y'})$ **for** $y' < 1$. **We are in case 1** $\Rightarrow T(n) = \Theta(n)$

MASTER THEOREM: EXAMPLES

- $T(n) = 3T(n/2) + n^2$
- **First find parameters:**
  - **a=3, b=2, f(n)=**n^2**, y=**$log_2 3 \approx 1.6$
  - **f(n)=**$n^2 = \Omega(n^{y'})$ **for y'>**$log_2 3$
  - $3(n/2)^2 = (3n^2)/4 \leq cn^2$ **for** $c \geq 3/4$
  - **We are in case 3!**
    - $T(n) = \Theta(n^2)$

- $T(n) = aT(n/b) + f(n),\ a \geq 1,\ b > 1,$ **f: asymptotically positive**
- **Let** $y = log_b a$ **and constant** $k \geq 0.$
- **Case 1:** $f(n) = O(n^{y'})$ **for** $y' < y \Rightarrow T(n) = \Theta(n^y)$
- **Case 2:** $f(n) = \Theta(n^y log^k n) \Rightarrow T(n) = \Theta(n^y log^{k+1} n)$
- **Case 3:** $f(n) = \Omega(n^{y'})$ **for** $y' > y$ **and** $af(n/b) \leq c(fn) \Rightarrow T(n) = \Theta(f(n))$

MASTER THEOREM: EXAMPLES

- $T(n) = 2T(n/2) + n$
- **Which algorithm we learned has this recurrence relation?**
- **First find parameters:**
  - **a=2, b=2, f(n)=**n**, y=**$log_2 2 = 1$
  - **f(n)=**$n = \Theta(n)$**.**
  - **We are in case 2 with k=0.**
    - $T(n) = \Theta(nlogn)$

- $T(n) = aT(n/b) + f(n),\ a \geq 1,\ b > 1,$ **f: asymptotically positive**
- **Let** $y = log_b a$ **and constant** $k \geq 0$.
- **Case 1:** $f(n) = O(n^{y'})$ **for** $y' < y \Rightarrow T(n) = \Theta(n^y)$
- **Case 2:** $f(n) = \Theta(n^y log^k n) \Rightarrow T(n) = \Theta(n^y log^{k+1} n)$
- **Case 3:** $f(n) = \Omega(n^{y'})$ **for** $y' > y$ **and** $af(n/b) \leq c(fn) \Rightarrow T(n) = \Theta(f(n))$

WHEN CAN'T WE USE MASTER THEOREM?

- When f is not asymptotically positive
  - Example: $T(n) = 64T(n/8) - n^2 \Rightarrow f(n) = -n^2$
- When we are in case 3 and $af(n/b) \leq cf(n)$ does not hold
  - Example: T(n)=T(n/2)+n(2-sin(n)): a=1, b=2, $y = log_2 1 = 0$
  - Are we in case 3?
    - af(n/b)=n/2(2-sin(n/2))<?cn(2-sin(n))=>2-sin(n/2)<? 2c(2-sin(n)): No because sine function oscillates
- When we are in case 2 and k<0
  - Example: T(n)=2T(n/2)+n/logn => a=2, b=2, y=1, f(n)=n/logn=>$n/logn = \Theta(nlog^{-1}n)$, k=-1.
  - We cannot use master theorem because k<0!

- $T(n) = aT(n/b) + f(n),\ a \geq 1,\ b > 1,$ f: asymptotically positive
- Let $y = log_b a$ and constant $k \geq 0$.
- Case 1: $f(n) = O(n^{y'})$ for $y' < y \Rightarrow T(n) = \Theta(n^y)$
- Case 2: $f(n) = \Theta(n^y log^k n) \Rightarrow T(n) = \Theta(n^y log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{y'})$ for $y' > y$ and $af(n/b) \leq c(fn) \Rightarrow T(n) = \Theta(f(n))$

WHEN CAN'T WE USE MASTER THEOREM?

- When a is not a constant
  - Example: $T(n) = 2^n T(n/8) + n^2 \Rightarrow a = 2^n$
- When $a < 1$ or $b \leq 1$
- Basically, we cannot use master theorem if the conditions on parameters are violated!
  - Carefully check if parameters are valid

- $T(n) = aT(n/b) + f(n),\ a \geq 1,\ b > 1,$ f: asymptotically positive
- Let $y = log_b a$ and constant $k \geq 0$.
- Case 1: $f(n) = O(n^{y'})$ for $y' < y \Rightarrow T(n) = \Theta(n^y)$
- Case 2: $f(n) = \Theta(n^y log^k n) \Rightarrow T(n) = \Theta(n^y log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{y'})$ for $y' > y$ and $af(n/b) \leq c(fn) \Rightarrow T(n) = \Theta(f(n))$

AVERAGE CASE ANALYSIS

- Why average case analysis?
  - Worst case is too pessimistic
    - Think about the bad candy problem and our solution
    - Worst case performance is O(n), which seems like we are not getting improved performance by cleverly using the test mechanism
    - However, on an average input, which is the case most of the time, runtime is O(logn): we are in fact better off!
  - Worst case analysis might miss these details, which are crucial!

SOME PROBABILITY BACKGROUND

- If you are familiar, discrete random variables can help perform average case analysis!
- Bernoulli Trials:
  1. Each trial results in one of two possible outcomes, denoted success (S) or failure (F).
  2. The probability of S remains constant from trial-to-trial and is denoted by p.
  3. The trials are independent.
  - Example: Coin flip.
- Geometric distribution: Represents the number of failures before you get a success in a series of Bernoulli trials
  - Expected value of number of trials before we get first success: 1/p.

AVERAGE CASE ANALYSIS: AN EXAMPLE

```
RANDOM-SEARCH(x, A, n)
    v = Ø\\v can contain each value once
    while |v| != n
        i = RANDOM(1, n)
        if A[i] = x
            return i
        else
            Add i to v
    return NIL
```

5-2 ***Searching an unsorted array***

This problem examines three algorithms for searching for a value x in an unsorted array A consisting of n elements.

Consider the following randomized strategy: pick a random index i into A. If $A[i] = x$, then we terminate; otherwise, we continue the search by picking a new random index into A. We continue picking random indices into A until we find an index j such that $A[j] = x$ or until we have checked every element of A. Note that we pick from the whole set of indices each time, so that we may examine a given element more than once.

a. Write pseudocode for a procedure RANDOM-SEARCH to implement the strategy above. Be sure that your algorithm terminates when all indices into A have been picked.

AVERAGE CASE ANALYSIS: SEARCHING AN UNSORTED ARRAY

- Each index picking event can be modeled as Bernoulli trials
  - Success probability of each trial is p=1/n. (have n values and one of them is x)
- Whole process can be modeled with geometric random variable G
  - Success: Finding x->want how many trials we will have before we have the ***first*** success
- Thus, expected number of indices we hit before RANDOM-SEARCH terminates=E[G]=1/p=n.

```
RANDOM-SEARCH(x, A, n)
  v = Ø
  while |v| != n
    i = RANDOM(1, n)
    if A[i] = x
      return i
    else
      Add i to v
  return NIL
```

b. Suppose that there is exactly one index i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates?

AVERAGE CASE ANALYSIS: SEARCHING AN UNSORTED ARRAY

- Each index picking event can be modeled as Bernoulli trials
  - Success probability of each trial is p=k/n. (have n values and one of them is x)
- Whole process can be modeled with geometric random variable G
  - Success: Finding x->want how many trials we will have before we have the ***first*** success
- Thus, expected number of indices we hit before RANDOM-SEARCH terminates=E[G]=1/p=n\k.

```
RANDOM-SEARCH(x, A, n)
  v = Ø
  while |v| != n
    i = RANDOM(1, n)
    if A[i] = x
      return i
    else
      Add i to v
  return NIL
```

c. Generalizing your solution to part (b), suppose that there are $k \geq 1$ indices i such that $A[i] = x$. What is the expected number of indices into A that we must pick before we find x and RANDOM-SEARCH terminates? Your answer should be a function of n and k.

Any Questions