CSCI203
Algorithms and Data Structures

String Searching and Improving Sorting II

Lecturer: Dr. Xueqiao Liu
Room 3.117
Email: xueqiao@uow.edu.au

Outline

- String Search
- Lower Bound for Comparison Sort
  - Decision Tree Model
- Sorting in Linear Time
  - Bucket Sorting
  - Radix Sorting

Looking for Text (in all the right places)

- Consider the problem of String Searching:
  - Given a text, t, is the subtext, s, present in it?
- This problem occurs in many real-life applications:
  - grep;
  - find in a text editor;
  - Genome matching;
  - Google search.

String Search Problem

- Example
  - t = AGCATGCTGCAGTCATGCTTAGGCTA
  - s= GCT
  - s appear three times in t, starting from locations 6,17,23
- There are a wide number of techniques to achieve this.
- Let us look at a couple of examples.

The Naïve Approach: Linear Search

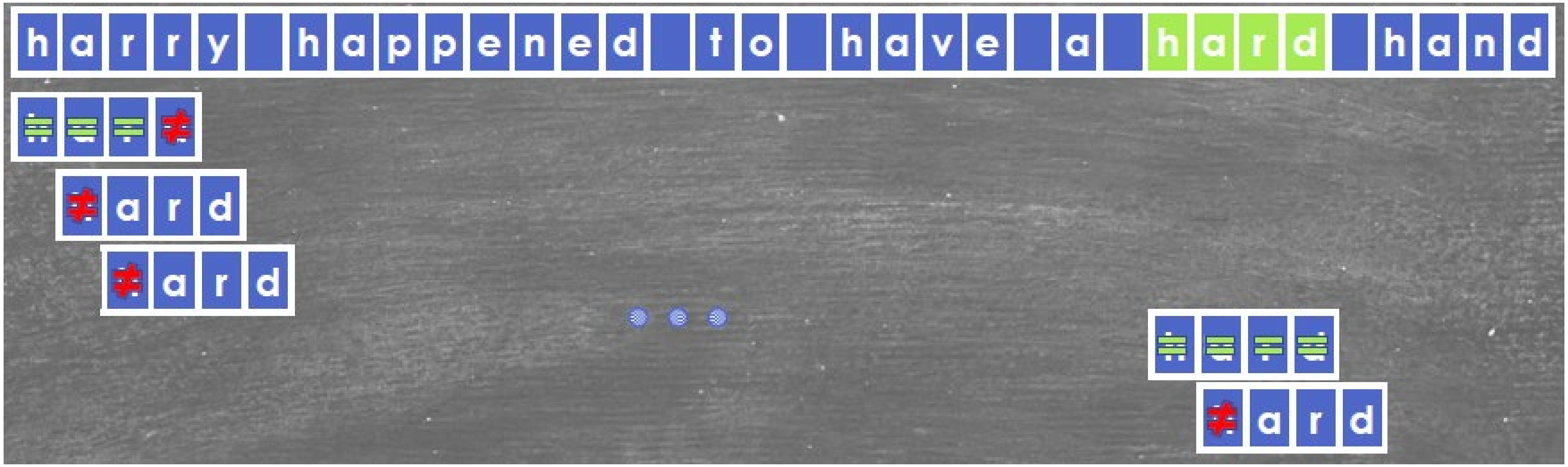
- The simplest possible approach is linear search:
  - Try to match s starting at each location in t.

```
for i from 0 to length(t) - length(s)
     j=0
     while j < length(s) do
          if (s(j) != t(i+j)) break
          j++
     od
     if j == length(s) print(" string found starting at location " i)
rof
```

- We can see this with an example.

Linear Search: An Example

- Let t be the string "*harry happened to have a hard hand*".
- Let s be the string "*hard*".
- The search proceeds as follows:



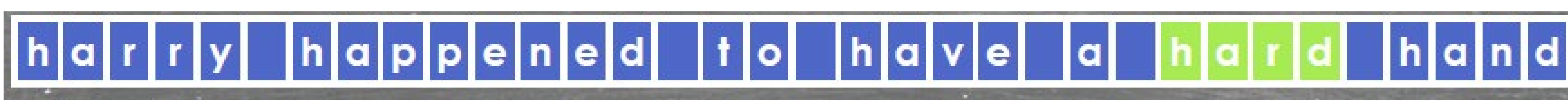
Linear Search $\neq$ Linear Time Search

- The outer loop in our algorithm is repeated $|t| - |s|$ times.
- Typically the string t is much longer than the string s, so this is $\Theta(t)$.
- The inner loop is repeated up to $|s|$ times for each time round the outer loop.
  - This is $\Theta(s)$
- The total number of comparisons is $\Theta(|s| \times |t|)$.
- Is this the best we can do?
- The best we can possibly do is $\Theta(|s| + |t|)$;
  - we have to at least look at each string!
- Can we actually achieve this goal of a linear time algorithm?

Linear Time Search

- To do this we will use hashing.
- We compare the hash of string *s* with the hash of each substring of *t* with the same length:

```
hash_s = hash(s)
for i in 0…length(t)-length(s)
    hash_t = hash(t[i..i+length(s)-1])
    if hash_s == hash_t then
        brute-force compare s and the substring
        if they match print(“ s found starting at location ” i)
    fi
rof
```



Linear Time Search

- This algorithm takes linear time, provided:
  - The hash function only collides rarely;
  - The hash function takes constant time to compute;
    - Independent of the length of string s!
- Surely, the second <u>requirement is impossible.</u>
- To hash a string of length $|s|$ must take $\Theta(|s|)$ operations.
  - Yes?
  - No!
- Not if we are clever.

Clever Hashing

- We note that we need $\Theta(|s|)$ time to compute $h(s)$.
- We also need $\Theta(|s|)$ time to compute $h(t[0..|s|-1)$, the initial substring of t.
- The trick is to compute the hash of each successive substring of t in constant time.
- If we look closely at these substrings, we see an interesting feature:
  - Successive substrings differ only by two characters.
- The first character of the first substring; **`harr`**
- The last character of the next substring. **`arry`**

Rolling Hash

- Maybe we can define a hash function which, given $h(\text{“}harr\text{”})$ can compute $h(\text{“}arry\text{”})$ in constant time.
- Let us define a rolling hash function, $r()$, so that:
  - $h(\text{“}arry\text{”}) = r(h(\text{“}harr\text{”}), \text{”}h\text{”}, \text{”}y\text{”})$
- We compute the hash of the next substring by removing the first and appending the new last characters;
- In this case we remove “h“ and append “y“.
- If we can compute a rolling hash in constant time then we can do string matching in linear time.
- How?

Karp-Rabin String Search

- The Karp-Rabin algorithm looks like this:

```
hash_s=hash(s)
hash_t=hash(t[0..length(s)-1])
for i in 0…length(t)-length(s)
    if hash_s == hash_t then
        brute-force compare s and the substring
        if they match print(" string found starting at location
        " i)
    fi
    hash_t=roll(hash_t,t[i],t[i+length(s)])
rof
```

- The function $roll(h, p, s)$ computes the rolling hash of the next substring given the hash of the existing substring, h, with the prefix p removed and the suffix, s, appended.
- We need only find a suitable function $roll()$.

How We Roll

- One popular way to compute $roll()$ is to use something called the Rabin fingerprint.
- We start by treating each symbol in the alphabet as an integer – use the ASCII code for example.
- We then find a random prime number > the size of the alphabet—let's pick 257.
- We now compute $h(\text{“}harr\text{”})$ as:
  - $257^3 * 104 + 257^2 * 97 + 257^1 * 114 + 257^0 * 114$
  - $= 1{,}771{,}793{,}837$

- Note: $\text{“}h\text{”} = 104$, $\text{“}a\text{”} = 97$ and $\text{“}r\text{”} = 114$.

The Next Hash

- Given that $h(\text{“}harr\text{”}) = 1{,}771{,}793{,}837$ how do we get $h(\text{“}arry\text{”})$?

- Simply compute $r(h, p, s) = 257 * (h - 257^3 * p) + s$
$$= 257.\,(1{,}771{,}793{,}837 - 257^3 * 104) + 121$$
- In this case the result is $1{,}654{,}094{,}526$ which is exactly the same as $h(\text{“}arry\text{”})$
$$257^3 * 97 + 257^2 * 114 + 257^1 * 114 + 257^0 * 121$$

- Note: if these values become too large, we can reduce them modulo m, where m is a convenient value—say 2^{15}or 2^{31}.

Efficient?

- We can compute our hash values for s and the initial substring of t using compact evaluation.

$$p^{k-1} * c_1 + p^{k-2} * c_2 + \ ... + \ p * c_{k-1} + c_k$$

- This requires a lot of multiplication!
- It can be re-written as…

$$h = \ c_k + p(c_{k-1} + p(c_{k-2} + \cdots \ + p(c_3 + \ p(c_2 + pc_1))..))$$

- Where $k = |s|$ and c_i is the i^{th} character of s.
- This requires $|s| - 1$ multiplications and $|s| - 1$ additions.

Efficiency!

- If we precompute $q = p^{k-1}$ we can find the next hash value, h' as:
- $h' = p * (h - q * c_i) + c_j$ where we remove character i and add character j.
- This requires only 1 multiplication and 1 addition.
  - Constant time.
- Thus we have $\Theta(|s|)$ operations to perform the initial hashes and $|t| - |s| \; * \Theta(1)$ operations to do the rehashing.
- Overall: our algorithm operates in $\Theta(|s| + |t|)$ time.
- We win!

Sorting

- We have seen few sorting algorithms
  - Insertion sort in worst-case takes $O(n^2)$
  - Merge Sort in worst-case takes $O(nlogn)$
  - Heap Sort in worst-case takes $O(nlogn)$
- Does this mean $O(nlogn)$ is the minimum cost?
- Can we do better?

Two Models for Sorting

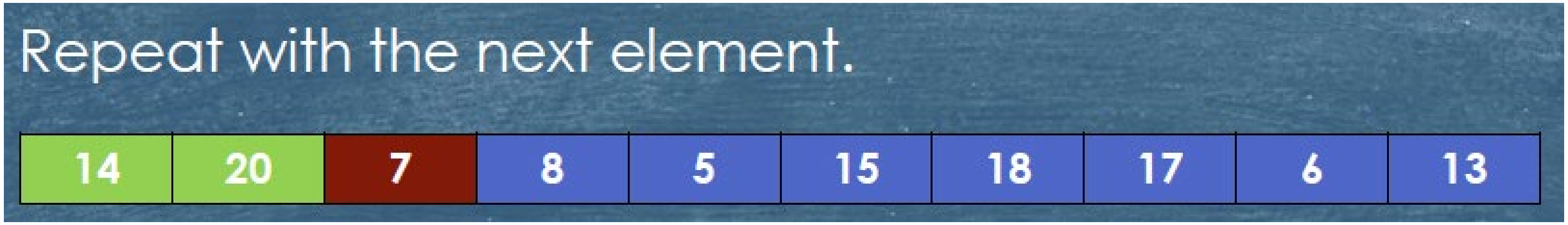
- Comparison-based sorting model
  - We will see that all comparison sort must take $\Omega(nlogn)$ comparisons.
- Other Sorts without relying on comparison
  - Bucket sort takes $O(n)$ on average
  - Radix sort also works in linear time

Lower Bound for Comparison Sort

- Comparison Sort
  - Only use comparisons between elements to gain order
    - E.g., given two inputs a_i and a_j, we perform one of comparisons to get the order between them
      - $a_i < a_j$
      - $a_i = a_j$
      - $a_i > a_j$
  - Examples include insertion sort, merge sort, heap sort, etc.

Examples-Insertion Sort

- Insertion sort uses the following strategy:
  - Start with the second element in the list.
  - Insert it in the right place in the preceding list.
  - Repeat with the next unsorted element.
  - Keep going until we have placed the last element in the list.



Examples-Merge Sort

- Takes the strategy that recursively divides the unsorted array into two parts and merging them in order.

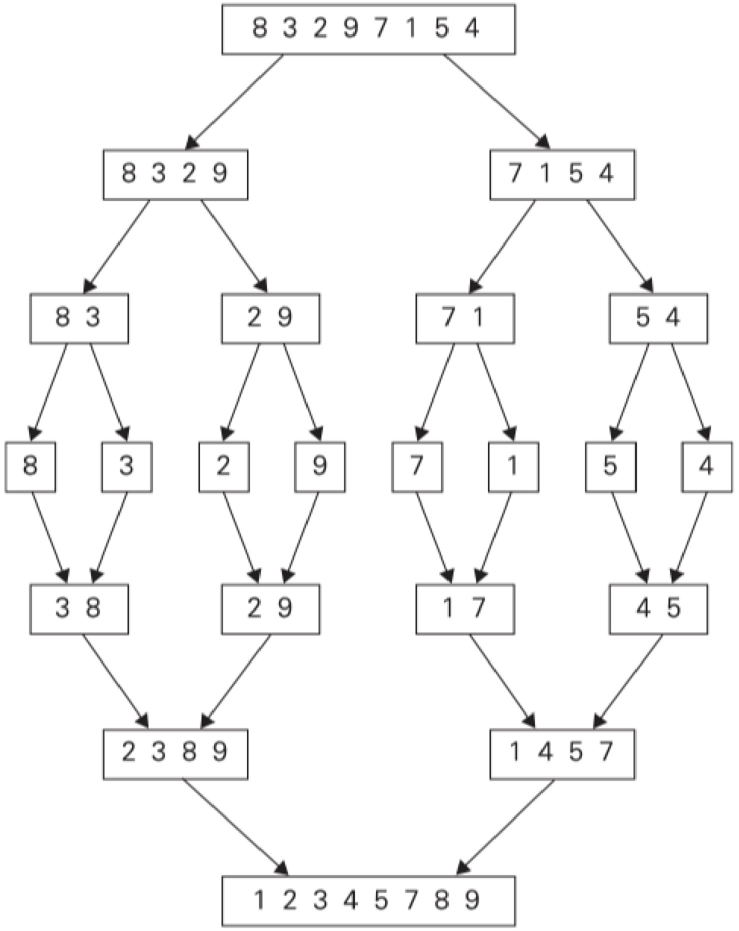


FIGURE 5.2 Example of mergesort operation.

Example-Heap Sort

```
Procedure heapsort(T[1..n])
    makeheap(T)
    for i = n to 2 step -1 do
        swap T[1] and T[i]
        siftdown(T[1 .. i - 1], 1)
```

```
procedure siftdown(Heap, i)
//move element i down to its correct position
    c = i * 2
    //Heap[c] < Heap[c+1] for a max-heap
    if Heap[c] > Heap[c + 1]
        c = c +1
    //for max-heap, the condition should be
    //changed to Heap[i] < Heap[c]
    if Heap[i] > Heap[c]
        swap (Heap[i], Heap[c])
        siftdown(Heap,c)
    endif
end
```

A lower bound for the worst case

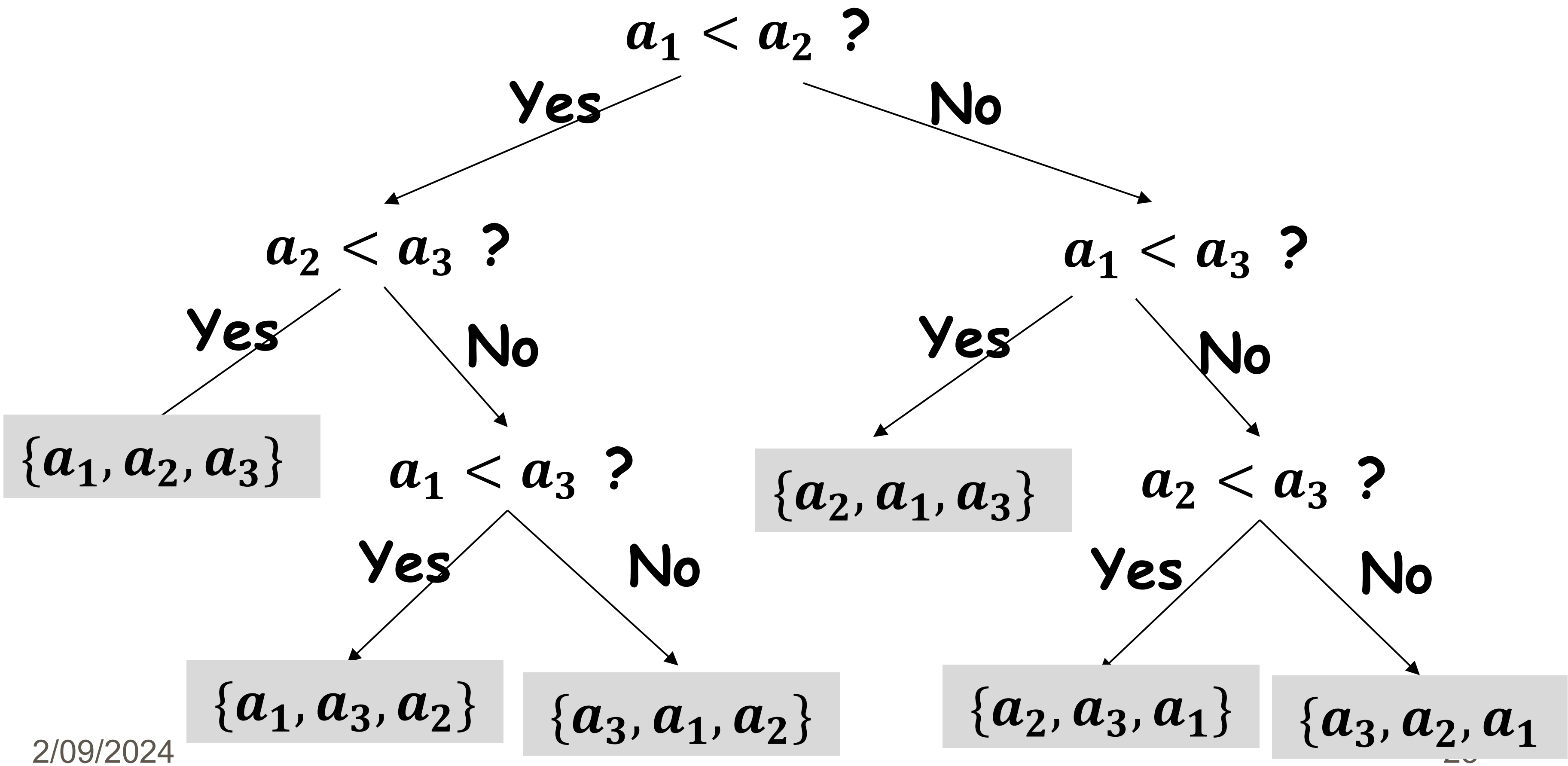
- Theorem:
  - All comparison sort must take $\Omega(nlogn)$ comparisons in the worst case.
- Next, we will show how to prove this theorem.

The Decision Tree Model

- A decision tree is a full binary tree that only represent the comparisons between elements.
- The comparisons order is determined by specific sorting algorithms.
- Control, data movement and all other inspect of algorithms are ignored.

Decision trees

- Sort $\{ a_1, a_2, a_3\}$



Decision Tree

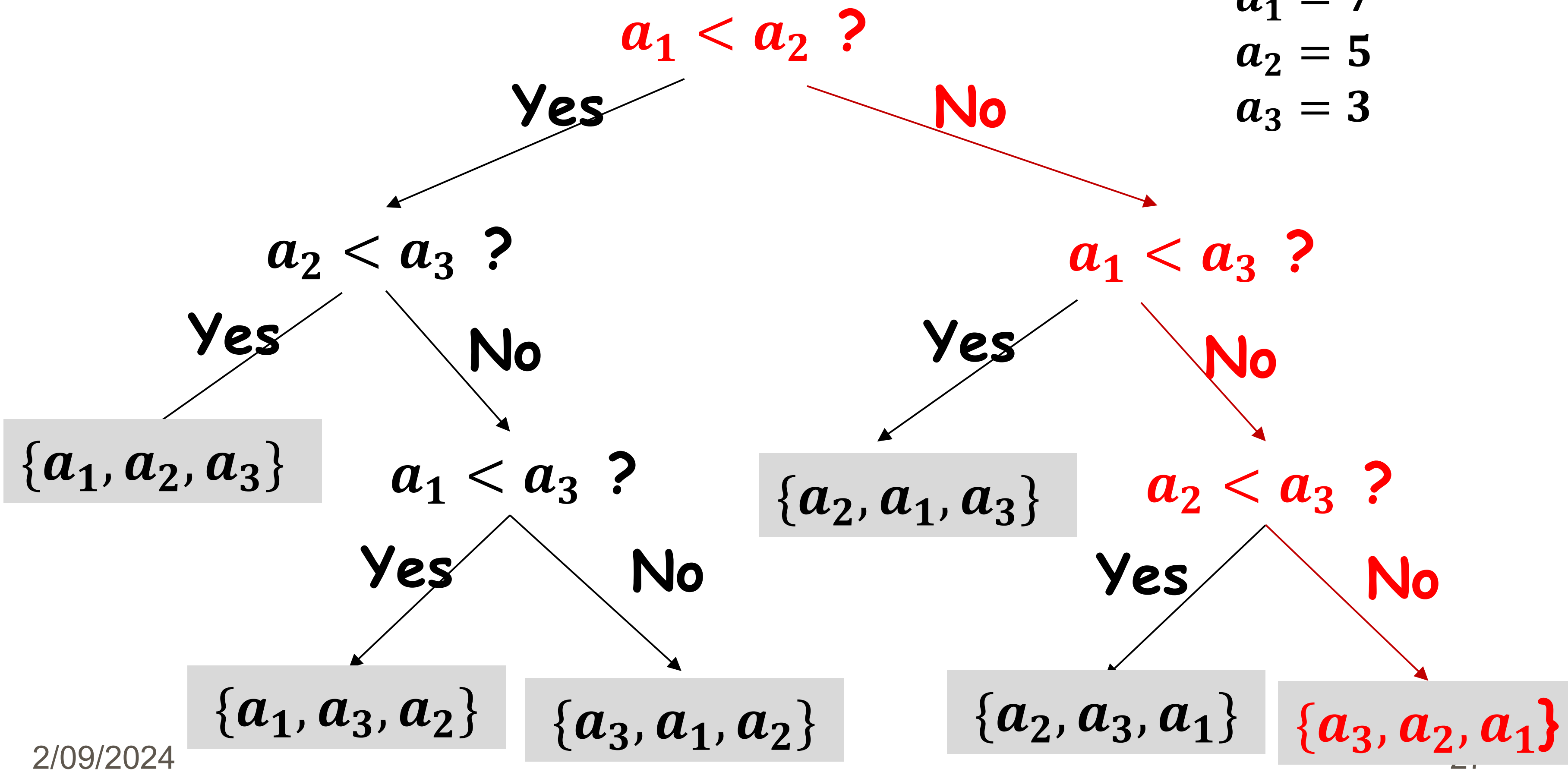
- Each internal node is a comparison of a_i and a_j
- All leaves nodes are all possible orderings of the items
- The execution of a sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf.
- All comparison-based algorithms have an associated decision tree

Decision trees

- Sort $\{ a_1, a_2, a_3\}$

Insertion Sort for {7,5,3}

$a_1 = 7$
$a_2 = 5$
$a_3 = 3$



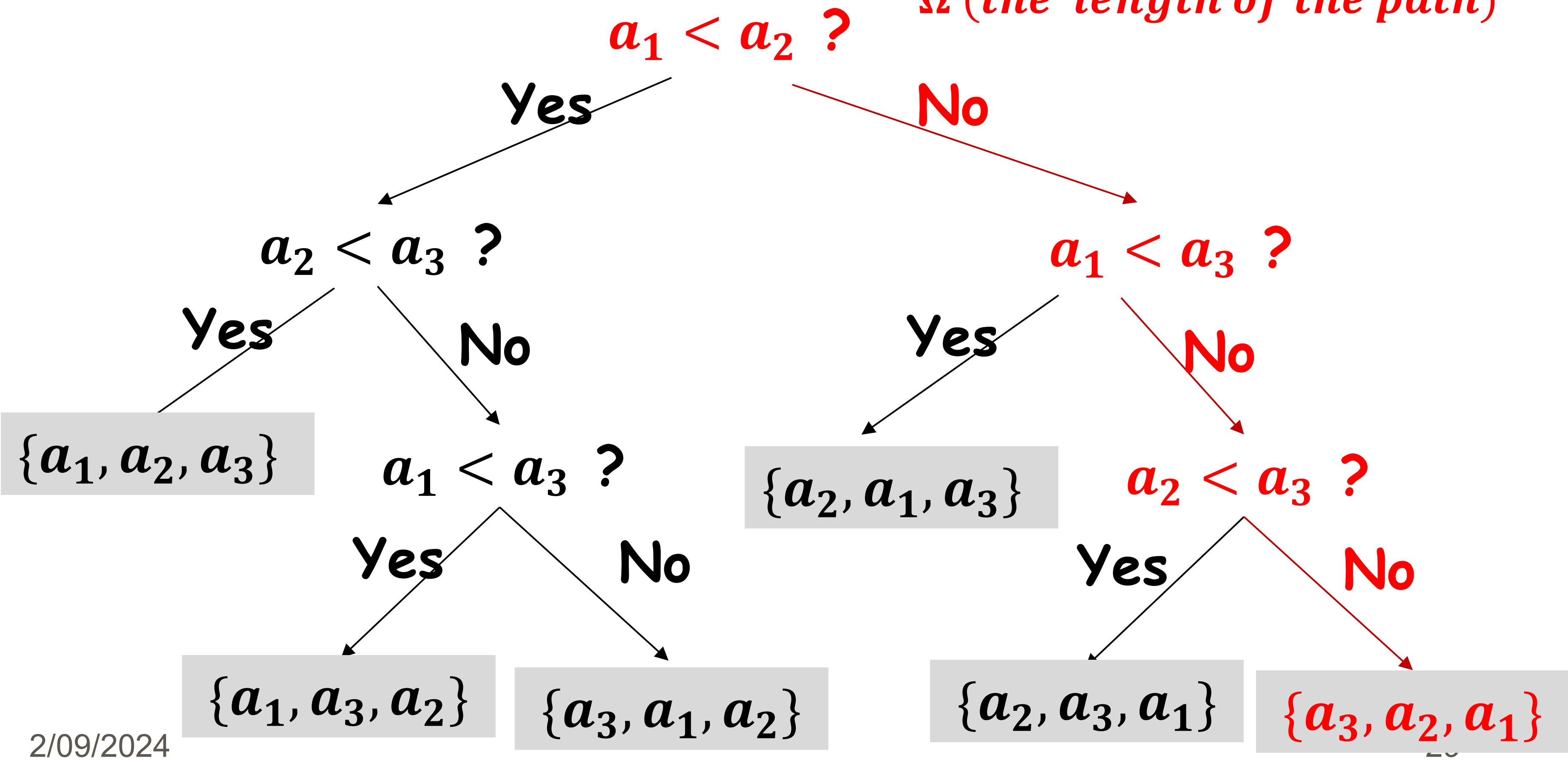
Decision Tree

- To sort n elements,
  - There will be n! permutations for n elements
  - There will be n! leaves in the decision tree
  - Any comparison algorithm must be able to produce each permutation of its input with size n, where each permutation will be a leaf node in the decision tree
  - Each of the leaf node is reachable from the root

What's the running time on a particular input?

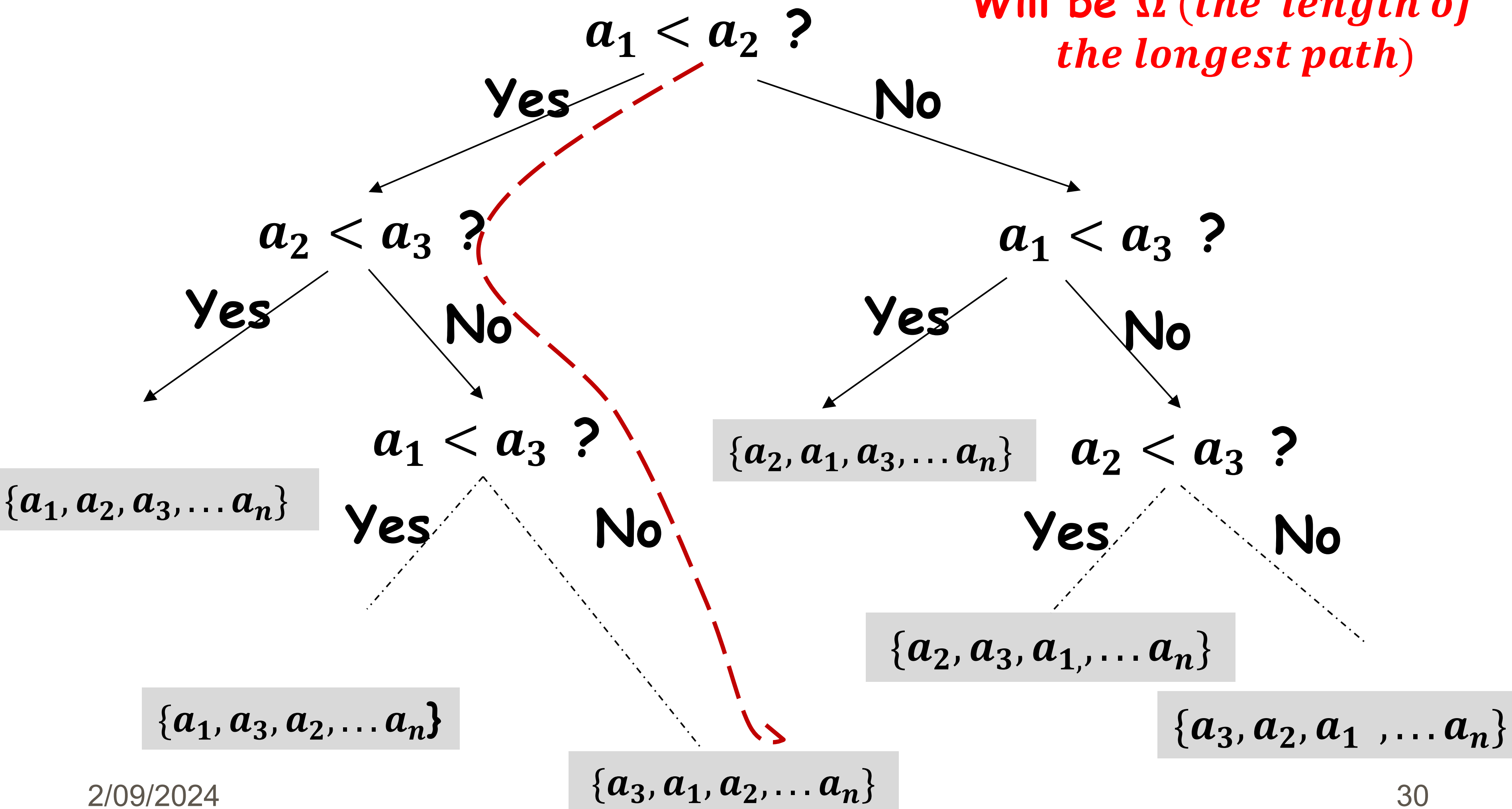
- Sort $\{ a_1, a_2, a_3 \}$

If we take the red path, the running time is $\Omega\,(\textit{the length of the path})$



What's the running time in the worst case?

Will be $\Omega(\textit{the length of the longest path})$



How long is the longest path?

- The tree has n! leaves
- It is a binary tree
- We know that for a complete binary tree, the height will be $h = log_2^{(No.\,of\ leaves)}$
- The longest path is at least $log_2^{n!}$
- Using Stirling's formula $n! \sim \left(\frac{n}{e}\right)^n$
  - $log_2^{n!} \sim \log_2 \left(\frac{n}{e}\right)^n$ =n $log_2^{n/e} = \Omega(nlogn)$

Proof for the lower bound

- Any comparison sorting algorithm can be represented as a decision tree with n! leaves
- The worst running time is the longest length of path in that tree
- All decision tree with n! leaves have depth $\Omega(nlogn)$

Corollary

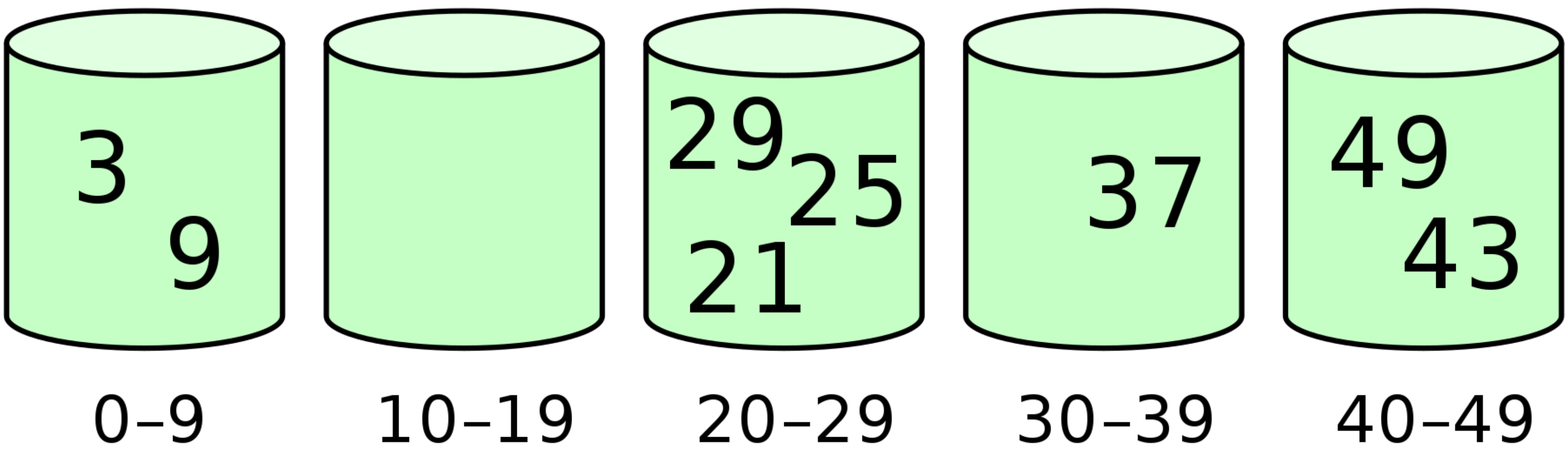
- Heapsort and merge sort are asymptotically optimal comparison sorts.
  - The upper bound for these two types of sorting is $O(nlogn)$, which matches the worst-case lower bound of comparison sorts.

Sorting in Linear Time

- Some sorting algorithm can run in linear time with specific requirement on its input.
- Bucket Sorting
  - Assume all inputs are drawn from uniform distribution with an average-case running time $O(n)$
  - works as follows
    - divide input space into n buckets
    - distribute n inputs into the bucket

Bucket Sort

29 25 3 49 9 37 21 43

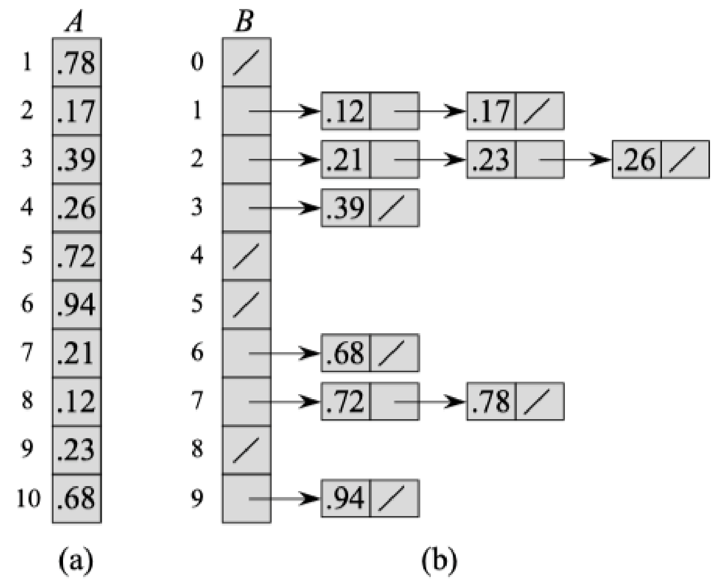


Sorted in O(n)

Bucket Sort

BUCKET-SORT(A)

```
n = A.length
let B[0 .. n − 1] be a new array
for i = 0 to n − 1
        make B[i] an empty list
for i = 1 to n
        insert A[i] into list B[⌊n A[i]⌋]
for i = 0 to n − 1
        sort list B[i] with insertion sort
concatenate the lists B[0], B[1], ..., B[n − 1] together in order
```



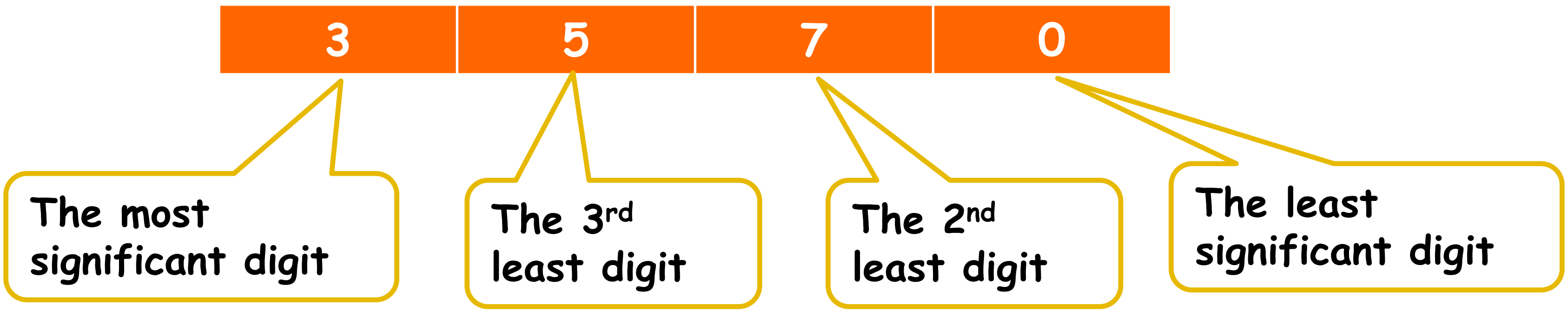
Each element is chosen from interval [0,1)

Bucket Sort

- We can analyze that the total running time of bucket sort is $\Theta(n)$
- We can implement each bucket as a linked list, which maintains the order for the element in the list
- Some issues:
  - Need to know the domain of elements to be sorted ahead of time

Radix Sort

- Works on decimal numbers
  - Each decimal digital has 10 possible values, 0,1,...,9
  - First sort the least significant digit
  - Then sort the 2^{nd},3^{rd} until the most significant bits
  - It requires the sort to be stable



Radix Sort

- Assume there are n elements in an array *A* each with d digits

RADIX-SORT(A, d)

```
for i = 1 to d
    use a stable sort to sort array A on digit i
```

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix Sort

- To sort n d-digit numbers where each digit is chosen from k possible values, radix sort will take $\theta(d(n+k))$ time if the stable sort it uses takes $\theta(n+k)$ time.
- Proof Analysis
  - There are d iterations for sorting
  - Each iteration needs to sort n digits into k buckets, which is $\theta(n+k)$
- When d is constant and k=O(n), we make radix sort run in linear time

Related Reference

- Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
  - Chapters 8
- Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
  - Chapters 11.2