COMP - 285

Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 6: Sorts and Sort Properties

Chris Lucas (cflucas@ncat.edu)

HW2 was released!

Due 09/15 @ 11:59PM ET

HW2 was released!

With video walkthrough! (pilot)

Quiz 0, 1

Video walkthroughs also!

HW1 was due!

Last night @ 11:59pm ET!

HW1 was due!

Late days accepted until the 11th!

Recall where we ended last lecture…

Big Questions!

- Which data structures use hashing? How fast are they?
- *What is hashing?*
- What about sorting algorithms?

Big Questions!

- Which data structures use hashing? How fast are they?

- What about sorting algorithms?

Hash Sets/Maps

Hash Sets vs. Maps

Hash Sets vs. Maps

- A set holds a collection (i.e. unordered) of distinct elements (i.e. no duplicates)
  - Example: {“A”, “B”, “C”}

Hash Sets vs. Maps

- A set holds a collection (i.e. unordered) of distinct elements (i.e. no duplicates)
    - Example: {“A”, “B”, “C”}

- A map holds a collection (i.e. unordered) of distinct keys (i.e. no duplicates) and their values. Each element is a pair of a key *and its associated value*.
    - Example: {“A”: 2, “B”: 4, “C”: 2}.

Hash Sets vs. Maps

- A set holds a collection (i.e. unordered) of distinct elements (i.e. no duplicates)
  - Example: {“A”, “B”, “C”}

- A map holds a collection (i.e. unordered) of distinct keys (i.e. no duplicates) and their values. Each element is a pair of a key *and its associated value*.
  - Example: {“A”: 2, “B”: 4, “C”: 2}.

- Implementations of these use hash functions, which helps us quickly decide where to insert/lookup key-value pairs in amortized O(1) time.

Really O(1) Time?

Set Operation	Runtime*
`insert(x)`	O(1)
`remove(x)`	O(1)
`contains(x)`	O(1)
`empty()`	O(1)
`size()`	O(1)

Map Operation	Runtime*
`put(k, v)`	O(1)
`remove(k)`	O(1)
`contains(k)`	O(1)
`get(k)`	O(1)
`empty()`	O(1)
`size()`	O(1)

Hash Set/Map Operations Chart

Data Structure	Access	Search	Insertion	Deletion
Hash Set/Map				

Hash Set/Map Operations Chart

Data Structure	Access	Search	Insertion	Deletion
Hash Set/Map	N/A	O(1)	O(1)	O(1)

Concrete Example

Intersecting Arrays

Given two vector<string> which contain lists of names vecA and vecB. Return true whether any name appears in both vecA and vecB.

```
A = {"bob", "sally", "jill"}
B = {"alice", "john", "sally"}
return true

A = {"bob", "sally", "jill"}
B = {"alice", "john"}
return false
```

Intersecting Arrays

Given two vector<string> which contain lists of names vecA and vecB. Return true whether any name appears in both vecA and vecB.

```
A = {"bob", "sally", "jill"}
B = {"alice", "john", "sally"}
return true

A = {"bob", "sally", "jill"}
B = {"alice", "john"}
return false
```

Let’s code it!!!

Kahoot!

www.kahoot.it, Code: 440 6904

Enter your @aggies.ncat email

Finding the Mode

Given a vector, identify the mode. The mode is defined as the value that occurs the most number of times. If there is a tie, select any of the valid mode answers.

```
A = {1, 2, 2, -13, 100, 3}
return 2

B = {-1, -1, -1, 0, 10, 0, 0}
return (either -1 or 0)
```

Finding the Mode

Given a vector, identify the mode. The mode is defined as the value that occurs the most number of times. If there is a tie, select any of the valid mode answers.

```
A = {1, 2, 2, -13, 100, 3}
return 2

B = {-1, -1, -1, 0, 10, 0, 0}
return (either -1 or 0)
```

Let’s code it!!!

Runtime Comparison

Data Structure	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)
Queue	O(n)	O(n)	O(1)	O(1)
Hash Set/Map	N/A	O(1)	O(1)	O(1)

Big Questions!

- Which data structures use hashing? How fast are they?
- What about sorting algorithms?

Motivation

- Sorts are classic algorithms that have building blocks that may be useful elsewhere (i.e. may be useful to internalize some of these code snippets/approaches).
- Sorts are a great place to practice our time and space complexity analysis.
- Divide & conquer can help break down complex problems efficiently.

Sorting

Let’s define the problem of sorting as follows:

- Input: vector of ints in some order
- Output: vec with original vec’s elements arranged in non-decreasing order

Sorting

Let’s define the problem of sorting as follows:

- Input: vector of ints in some order
- Output: vec with original vec’s elements arranged in non-decreasing order

[4, 5, 0, 12, -3, 1]

Sorting

Let's define the problem of sorting as follows:

- Input: vector of ints in some order
- Output: vec with original vec's elements arranged in non-decreasing order

[4, 5, 0, 12, -3, 1]

Sorting

Let’s define the problem of sorting as follows:

- Input: vector of ints in some order
- Output: vec with original vec’s elements arranged in non-decreasing order

[4, 5, 0, 12, -3, 1]

[-3, 0, 1, 4, 5, 12]

Sorting

When comparing different sorting algorithms, these are some of the properties we care about:

Sorting

When comparing different sorting algorithms, these are some of the properties we care about:

- **Best-case/worst-case/average-case time complexity**

Sorting

When comparing different sorting algorithms, these are some of the properties we care about:

- Best-case/worst-case/average-case time complexity

- In-place: can we use only O(1) additional space?

Sorting

When comparing different sorting algorithms, these are some of the properties we care about:

- Best-case/worst-case/average-case time complexity

- In-place: can we use only O(1) additional space?

- Adaptive: does it run faster if the array is partially sorted?

Sorting

When comparing different sorting algorithms, these are some of the properties we care about:

- Best-case/worst-case/average-case time complexity
- In-place: can we use only O(1) additional space?
- Adaptive: does it run faster if the array is partially sorted?
- Stable: will elements of the same value stay ordered relative to each other?

Sort Properties: Stability

Sort Properties: Stability

[4, 5, 1, 12, -3, 1]

Sort Properties: Stability

[4, 5, 1, 12, -3, 1]

[-3, 1, 1, 4, 5, 12]

Sort Properties: Instability

[4, 5, 1, 12, -3, 1]

Sort Properties: Instability

[4, 5, 1, 12, -3, 1]

[-3, 1, 1, 4, 5, 12]

Selection Sort!!

Selection Sort

For each index (0, 1, 2, ..., N-1), repeatedly pick the next smallest element from the rest of the array and swap spots.

[9, 4, 5, 1, 5, 2] : start

[1, 4, 5, 9, 5, 2] : swap(0, 3)

[1, 2, 5, 9, 5, 4] : swap(1, 5)

[1, 2, 4, 9, 5, 5] : swap(2, 5)

[1, 2, 4, 5, 9, 5] : swap(3, 4)

[1, 2, 4, 5, 5, 9] : swap(4, 5)

[1, 2, 4, 5, 5, 9] : finished

Stable?

In-place?

Selection Sort

For each index (0, 1, 2, ..., N-1), repeatedly pick the next smallest element from the rest of the array and swap spots.

[9, 4, 5, 1, 5, 2] : start

[1, 4, 5, 9, 5, 2] : swap(0, 3)

[1, 2, 5, 9, 5, 4] : swap(1, 5)

[1, 2, 4, 9, 5, 5] : swap(2, 5)

[1, 2, 4, 5, 9, 5] : swap(3, 4)

[1, 2, 4, 5, 5, 9] : swap(4, 5)

[1, 2, 4, 5, 5, 9] : finished

Stable? NO

In-place? YES

Selection Sort Pseudocode

```
algorithm selectionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 0, 1, 2, ..., N-2
    min_index = i
    for j = i+1, i+2, ..., N-1
      if vec[j] < vec[min_index]
        min_index = j
    temp = vec[i]
    vec[i] = vec[min_index]
    vec[min_index] = temp
```

What’s the tight upper-bound on the:

- Best-case runtime?
- Worst-case runtime?
- Average-case runtime?
- Worst-case space complexity?

Is this adaptive?

Selection Sort Pseudocode

```
algorithm selectionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 0, 1, 2, ..., N-2
    min_index = i
    for j = i+1, i+2, ..., N-1
      if vec[j] < vec[min_index]
        min_index = j
    temp = vec[i]
    vec[i] = vec[min_index]
    vec[min_index] = temp
```

What’s the tight upper-bound on the:

- Best-case runtime? $O(n^2)$
- Worst-case runtime? $O(n^2)$
- Average-case runtime? $O(n^2)$
- Worst-case space complexity? $O(1)$

Is this adaptive? No

Insertion Sort!!

Insertion Sort: Intuition

Maintain a vector of sorted elements so far. We pick the element by element from the input vector, and figure out where it goes in our sorted vector. We will insert before the first element that is greater. This is similar to how you might organize playing cards in your hand.

[9, 4, 5, 1, 5, 2] : []

[~~9~~, 4, 5, 1, 5, 2] : [9]

[~~9~~, ~~4~~, 5, 1, 5, 2] : [4, 9]

[~~9~~, ~~4~~, ~~5~~, 1, 5, 2] : [4, 5, 9]

[~~9~~, ~~4~~, ~~5~~, ~~1~~, 5, 2] : [1, 4, 5, 9]

[~~9~~, ~~4~~, ~~5~~, ~~1~~, ~~5~~, 2] : [1, 4, 5, 5, 9]

[~~9~~, ~~4~~, ~~5~~, ~~1~~, ~~5~~, ~~2~~] : [1, 2, 4, 5, 5, 9]

Stable?

In-place?

Kahoot!

www.kahoot.it, Code: 440 6904

Enter your @aggies.ncat email

Insertion Sort: Intuition

Maintain a vector of sorted elements so far. We pick the element by element from the input vector, and figure out where it goes in our sorted vector. We will insert before the first element that is greater. This is similar to how you might organize playing cards in your hand.

[9, 4, 5, 1, 5, 2] : []

[~~9~~, 4, 5, 1, 5, 2] : [9]

[~~9~~, ~~4~~, 5, 1, 5, 2] : [4, 9]

[~~9~~, ~~4~~, ~~5~~, 1, 5, 2] : [4, 5, 9]

[~~9~~, ~~4~~, ~~5~~, ~~1~~, 5, 2] : [1, 4, 5, 9]

[~~9~~, ~~4~~, ~~5~~, ~~1~~, ~~5~~, 2] : [1, 4, 5, 5, 9]

[~~9~~, ~~4~~, ~~5~~, ~~1~~, ~~5~~, ~~2~~] : [1, 2, 4, 5, 5, 9]

Stable? Yes (insert before first greater element)

In-place? No

Insertion Sort: Intuition

Maintain a vector of sorted elements so far. We pic

ctor, and

figure out where it goes in our vector so th

he first

element that is greater. This is si

ur hand.

Can we do it in place?

[9, 4, 5, 1, 5

5, 9]

[~~9~~, 4,

4, 5, 5, 9]

[~~9~~, ~~4~~, 5

es (insert before first greater element)

[~~9~~, ~~4~~, ~~5~~,

in-place? No

[~~9~~, ~~4~~, ~~5~~, ~~1~~,

] :

[1, 4, 5, 9]

Insertion Sort (in-place): Intuition

Insertion Sort (in-place): Intuition

BEFORE: Maintain a vector of sorted elements so far. We pick the next element from the vector, and figure out where it goes in our vector so that it stays sorted. We will insert before the first element that is greater. This is similar to how you might organize playing cards in your hand.

[9, 4, 5, 1, 5, 2] : []

[~~9~~, 4, 5, 1, 5, 2] : [9]

Insertion Sort (in-place): Intuition

BEFORE: Maintain a vector of sorted elements so far. We pick the next element from the vector, and figure out where it goes in our vector so that it stays sorted. We will insert before the first element that is greater. This is similar to how you might organize playing cards in your hand.

[9, 4, 5, 1, 5, 2] : []

[~~9~~, 4, 5, 1, 5, 2] : [9]

AFTER: Maintain an index of the sorted elements so far. We pick the next element from the vector, figure out where it goes, and insert by shifting all other elements to the right by one. Increment the index tracking sorted elements so far.

[1, 4, 5, 9, 5, 2] : i=3 | https://visualgo.net/en/sorting

Insertion Sort In-Place Pseudocode

```
algorithm insertionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 1, 2, ..., N-1
    next = vec[i]
    j = i-1
    while j >= 0 and vec[j] > next
      vec[j+1] = vec[j]
      j = j-1
    vec[j+1] = next
```

What's the tight upper-bound on the:

- Best-case runtime?
- Worst-case runtime?
- Average-case runtime?
- Worst-case space complexity?

Is this adaptive?

Insertion Sort In-Place Pseudocode

```
algorithm insertionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 1, 2, ..., N-1
    next = vec[i]
    j = i-1
    while j >= 0 and vec[j] > next
      vec[j+1] = vec[j]
      j = j-1
    vec[j+1] = next
```

What’s the tight upper-bound on the:

- Best-case runtime? O(n)
- Worst-case runtime? $O(n^2)$
- Average-case runtime? $O(n^2)$
- Worst-case space complexity? O(1

Is this adaptive?

Insertion Sort In-Place Pseudocode

```
algorithm insertionSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  for index i = 1, 2, ..., N-1
    next = vec[i]
    j = i-1
    while j >= 0 and vec[j] > next
      vec[j+1] = vec[j]
      j = j-1
    vec[j+1] = next
```

What’s the tight upper-bound on the:

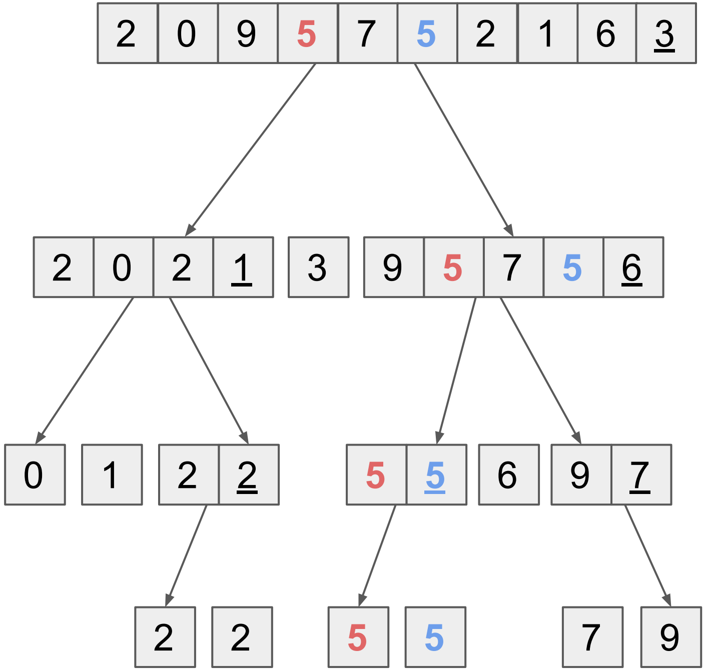
- Best-case runtime? O(n)
- Worst-case runtime? $O(n^2)$
- Average-case runtime? $O(n^2)$
- Worst-case space complexity? O(1

Is this adaptive? Yes

QuickSort!!

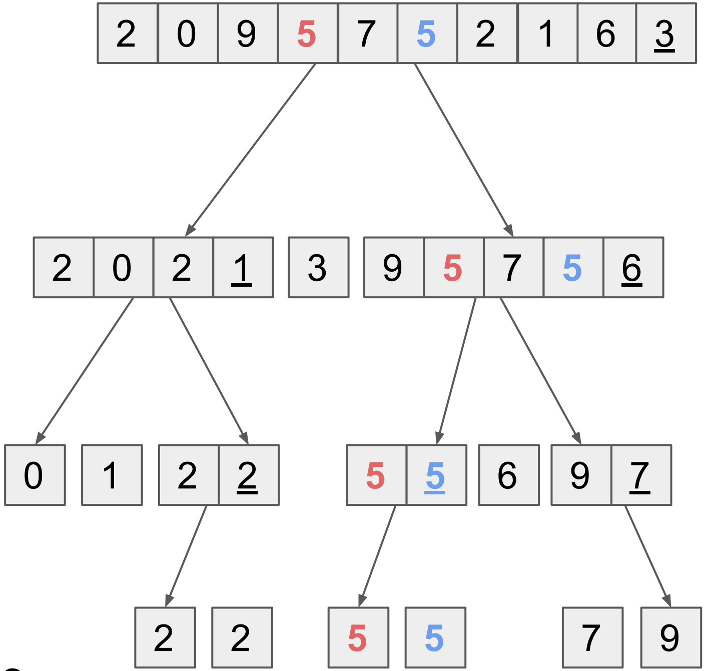
QuickSort: Intuition

- Pick the last element in the list. (Pivot)
- Put the rest of the elements into two partitions (vectors)
  - "all elements <= pivot"
  - "all elements > pivot"
- Then do the same steps on the two partitions until the vectors are small enough to not need sorting. (recursion!)
- Once we've sorted the smaller vectors, glue them back together at each level, along with the pivot.



QuickSort: Intuition

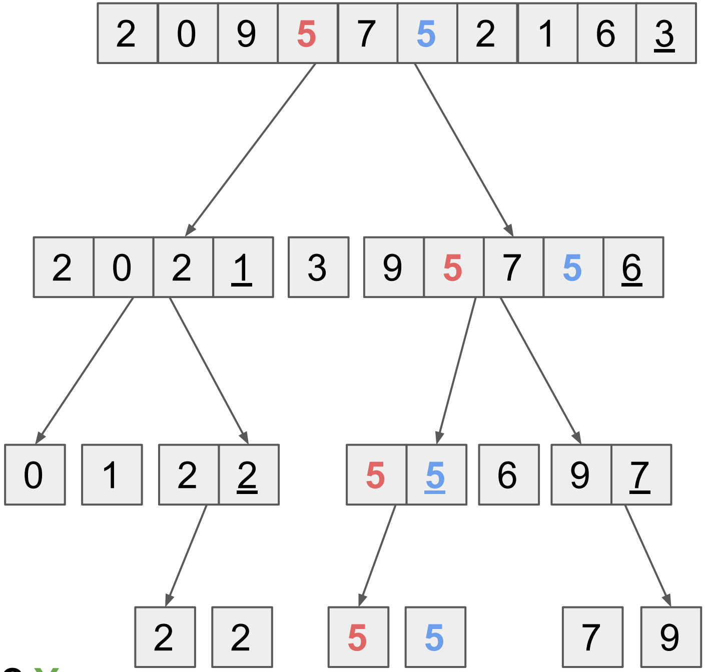
- Pick the last element in the list. (Pivot)
- Put the rest of the elements into two partitions (vectors)
    - "all elements <= pivot"
    - "all elements > pivot"
- Then do the same steps on the two partitions until the vectors are small enough to not need sorting. (recursion!)
- Once we've sorted the smaller vectors, glue them back together at each level, along with the pivot.



Stable?
In-Place?

QuickSort: Intuition

- Pick the last element in the list. (Pivot)
- Put the rest of the elements into two partitions (vectors)
  - "all elements <= pivot"
  - "all elements > pivot"
- Then do the same steps on the two partitions until the vectors are small enough to not need sorting. (recursion!)
- Once we've sorted the smaller vectors, glue them back together at each level, along with the pivot.



Stable? Yes

In-Place? No

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

- Pick the last element in the list. (Pivot)
- Put the rest of the elements into two partitions (lists)
  - "all elements <= pivot"
  - "all elements > pivot"
- Then do the same steps on the two partitions until the lists are small enough to not need sorting. (recursion!)
- Once we've sorted the smaller lists, glue them back together at each level, along with the pivot.

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

- Pick the last element in the list. (Pivot)
- Put the rest of the elements into two partitions (lists)
  - "all elements <= pivot"
  - "all elements > pivot"
- Then do the same steps on the two partitions until the lists are small enough to not need sorting. (recursion!)
- Once we've sorted the smaller lists, glue them back together at each level, along with the pivot.

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What’s the tight upper-bound on the:

- Best-case runtime?
- Worst-case runtime?
- Average-case runtime?
- Worst-case space complexity?

Is this adaptive?

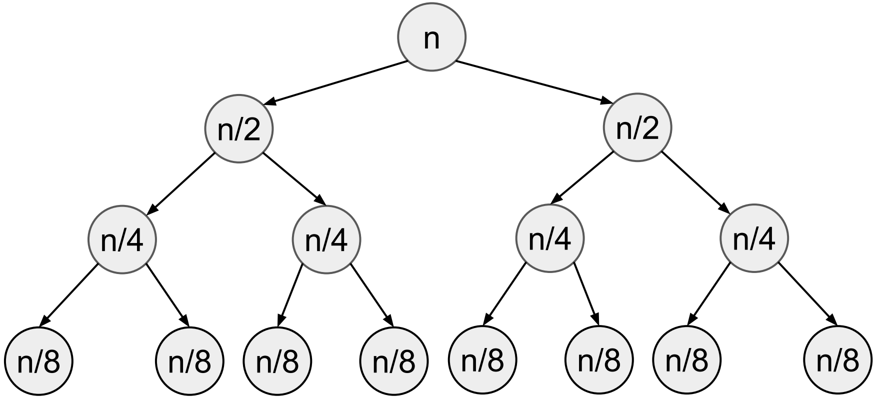
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime?



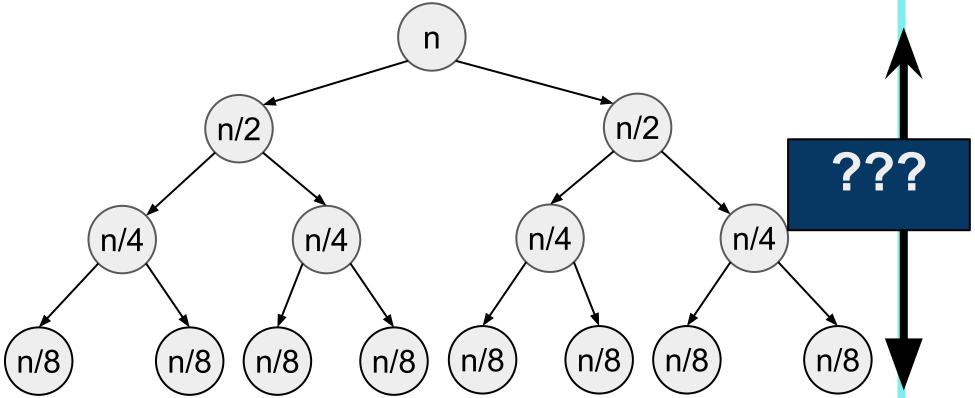
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime?



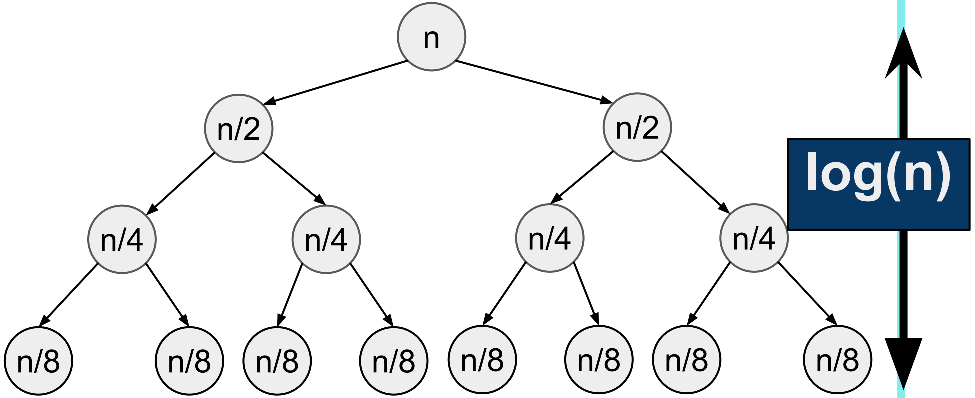
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime?



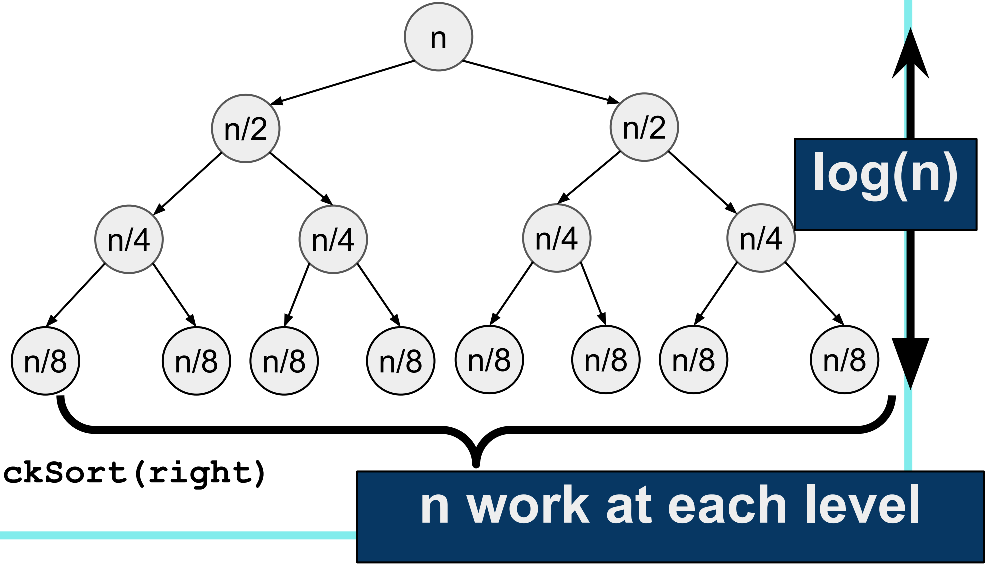
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime?



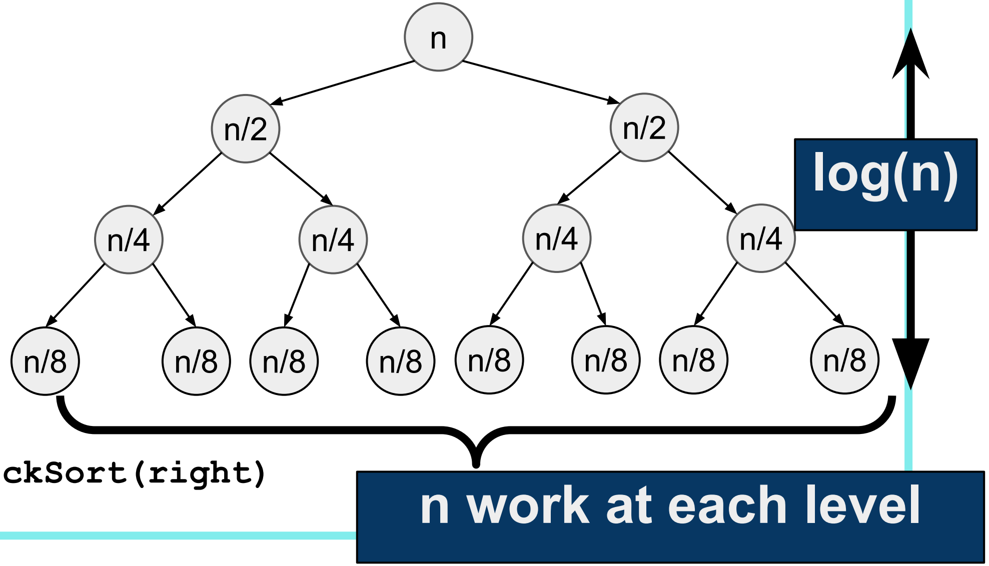
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)



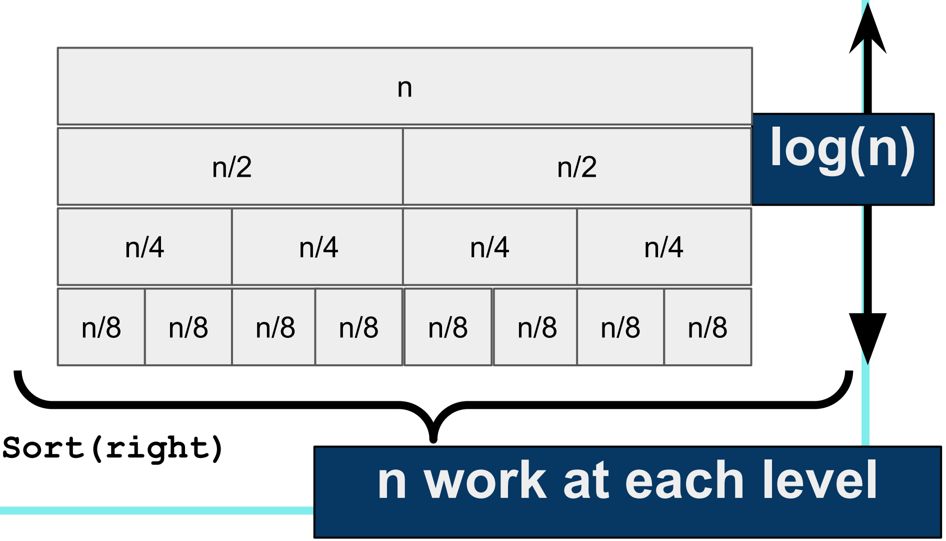
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)



n

n/2 n/2

n/4 n/4 n/4 n/4

n/8 n/8 n/8 n/8 n/8 n/8 n/8 n/8

log(n)

n work at each level

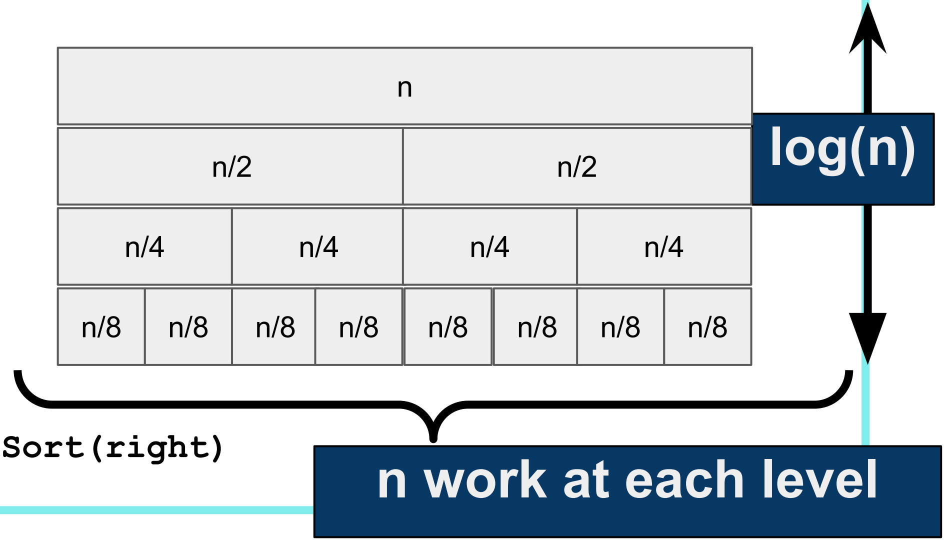
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)



n

n/2 n/2

n/4 n/4 n/4 n/4

n/8 n/8 n/8 n/8 n/8 n/8 n/8 n/8

log(n)

n work at each level

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What’s the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime?

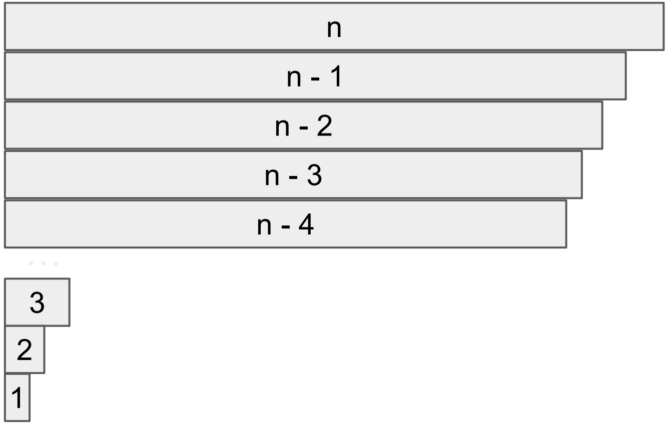
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime?



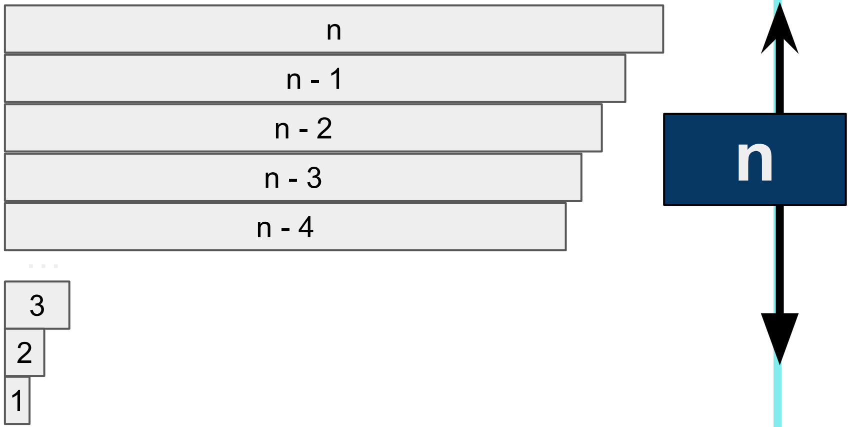
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What’s the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime?



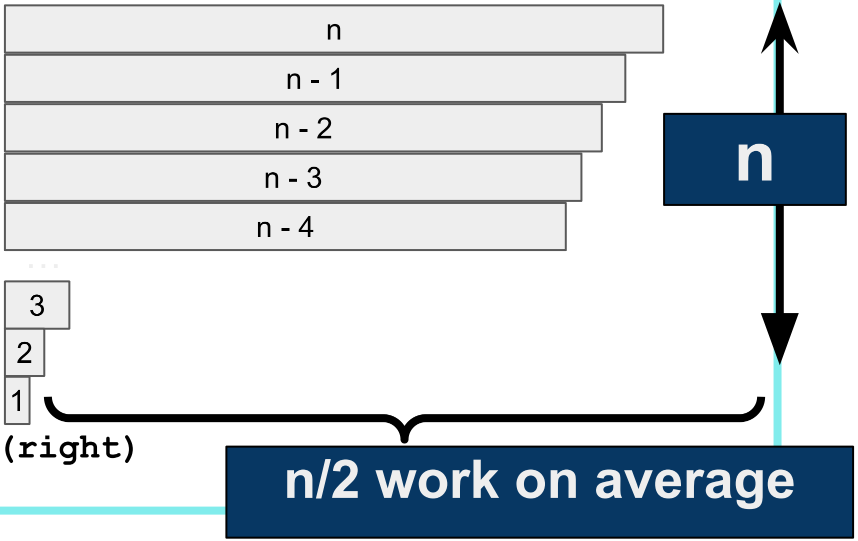
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime?



n

n/2 work on average

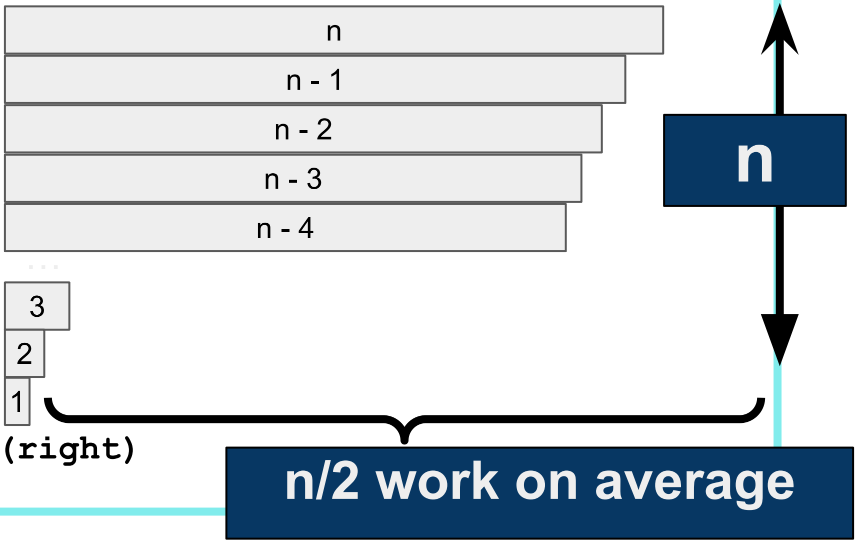
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? O(n^2)



QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? O(n^2)
- Average-case runtime?

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivo
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? O(n^2)
- Average-case runtime? O(nlogn)

- **Average will avoid the worst-case pivots (min or max each time).**
- **Even if we don't divide 50-50 (c = 2), we still wind up with O(n log(n)).**
- **Why? Because even picking a pivot that splits 90-10 each time is still dividing by some multiplicative constant (c = 1.11), so there is still O(log(n)) levels (as the base can be changed by multiplying out the log-base cleverly using the log change-of-base formula).**

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What’s the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? $O(n^2)$
- Average-case runtime? O(nlogn)
- Worst-case space complexity?

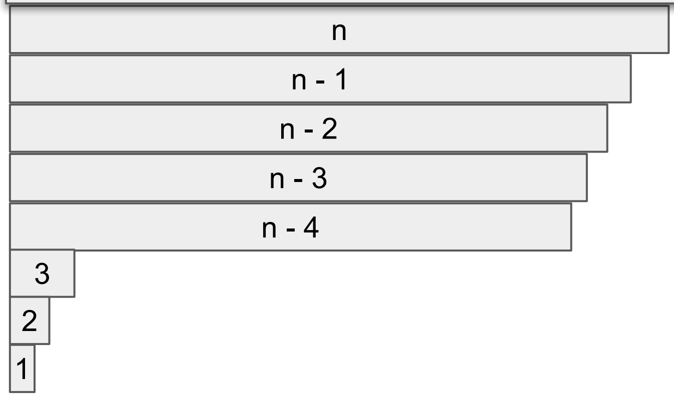
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? O(n^2)
- Average-case runtime? O(nlogn)
- Worst-case space complexity?



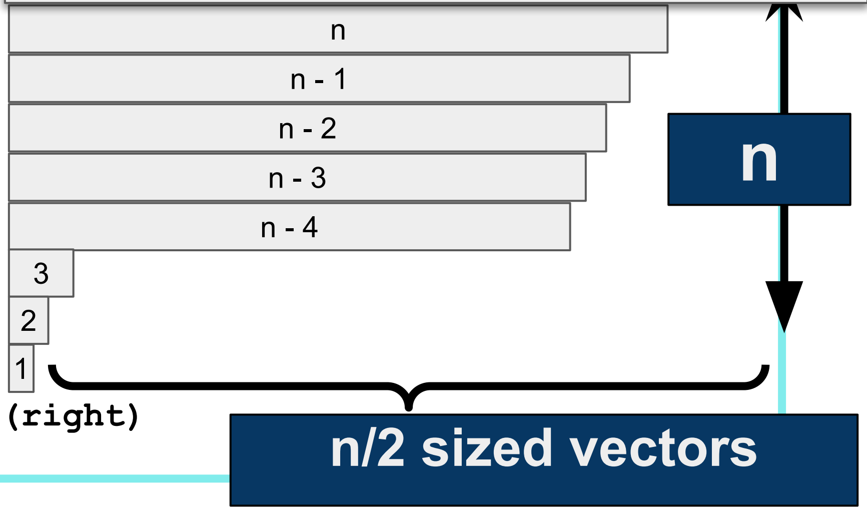
QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What’s the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? O(n^2)
- Average-case runtime? O(nlogn)
- Worst-case space complexity?O(n^2)



QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What’s the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? O(n^2)
- Average-case runtime? O(nlogn)
- Worst-case space complexity?O(n^2)

Is it adaptive?

QuickSort: Pseudocode

```
algorithm quickSort
  Input: vector<int> vec of size N
  Output: vector<int> with sorted elements

  if N < 2
    return vec
  pivot = vec[N-1]
  left = new empty vec
  right = new empty vec
  for index i = 0, 1, 2, ... N-2
    if vec[i] <= pivot
      left.push_back(vec[i])
    else
      right.push_back(vec[i])
  return quickSort(left) + [pivot] + quickSort(right)
```

What's the tight upper-bound on the:

- Best-case runtime? O(nlogn)
- Worst-case runtime? $O(n^2)$
- Average-case runtime? O(nlogn)
- Worst-case space complexity?$O(n^2)$

Is it adaptive? NO

Key Takeaways

- Insertion Sort is slightly less intuitive than Selection Sort, but faster in the best case. Quick Sort is the least intuitive of the three, but also the fastest in practice. It's best case and average case are very good, and further optimizations can be made to convert the algorithm into the in-place version.
    - All of the above algorithms take O(N2) in the worst case
- Divide and Conquer is a pattern in recursive algorithms that is usually efficient
- Divide and Conquer algorithms have two defining characteristics
    - At each step, there are 2 or more recursive calls
    - The problem is being reduced by some multiplicative factor at each call

COMP - 285

Advanced Analysis of Algorithms

Welcome to COMP 285

Lecture 6: Sorts and Sort Properties

Chris Lucas (cflucas@ncat.edu)