Lecture 12/13: Insertion, Faster Sorts

BT 3051 – Data Structures and Algorithms for Biology

Karthik Raman
Department of Biotechnology
Bhupat and Jyoti Mehta School of Biosciences
Indian Institute of Technology Madras

Insertion Sort

Insertion Sort

```
def InsertionSort(a):
    n = len(a)
    for i in range(1,n):
        for j in range(i, 0, -1):
            if a[j] < a[j-1]:
                a[j], a[j-1] = a[j-1], a[j]
            else:
                break
    return a
```

Can we improve upon this?

Insertion Sort

```python
def InsertionSort(a):
    n = len(a)
    for i in range(1, n):
        pos = i
        currval = a[i]

        while pos > 0 and a[pos - 1] > currval:
            a[pos] = a[pos - 1]
            pos = pos - 1

        a[pos] = currval
```

Properties

- Stable, $O(1)$ extra space
- $O(n^2)$ comparisons and swaps
- Adaptive: $O(n)$ when nearly sorted
- Very low overhead

Insertion Sort

```python
def InsertionSort(a):
    n = len(a)
    for i in range(1, n):
        pos = i
        currval = a[i]

        while pos > 0 and a[pos - 1] > currval:
            a[pos] = a[pos - 1]
            pos = pos - 1

        a[pos] = currval
```

Properties

- Stable, $O(1)$ extra space
- $O(n^2)$ comparisons and swaps
- Adaptive: $O(n)$ when nearly sorted
- Very low overhead

Insertion Sort

```python
def InsertionSort(a):
    n = len(a)
    for i in range(1, n):
        pos = i
        currval = a[i]

        while pos > 0 and a[pos - 1] > currval:
            a[pos] = a[pos - 1]
            pos = pos - 1

        a[pos] = currval
```

Properties

- Stable, $O(1)$ extra space
- $O(n^2)$ comparisons and swaps
- Adaptive: $O(n)$ when nearly sorted
- Very low overhead

Insertion Sort

```
def InsertionSort(a):
    n = len(a)
    for i in range(1, n):
        pos = i
        currval = a[i]

        while pos > 0 and a[pos - 1] > currval:
            a[pos] = a[pos - 1]
            pos = pos - 1

        a[pos] = currval
```

Properties

- Stable, $O(1)$ extra space
- $O(n^2)$ comparisons and swaps
- Adaptive: $O(n)$ when nearly sorted
- Very low overhead

Insertion Sort

- Elementary sorting algorithm with $O(n^2)$ worst-case time; but
- Insertion sort is the algorithm of choice
  - either when the data is nearly sorted (because it is adaptive), or
  - when the problem size is small (because it has low overhead)
- It is also stable
- Insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort

Insertion Sort

- Elementary sorting algorithm with $O(n^2)$ worst-case time; but
- Insertion sort is the algorithm of choice
  - either when the data is nearly sorted (because it is adaptive), or
  - when the problem size is small (because it has low overhead)
- It is also stable
- Insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort

Insertion Sort

- Elementary sorting algorithm with $O(n^2)$ worst-case time; but
- Insertion sort is the algorithm of choice
  - either when the data is nearly sorted (because it is adaptive), or
  - when the problem size is small (because it has low overhead)
- It is also stable
- Insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort

Insertion Sort

- Elementary sorting algorithm with $O(n^2)$ worst-case time; but
- Insertion sort is the algorithm of choice
  - either when the data is nearly sorted (because it is adaptive), or
  - when the problem size is small (because it has low overhead)
- It is also stable
- Insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort

Insertion Sort

- Elementary sorting algorithm with $O(n^2)$ worst-case time; but
- Insertion sort is the algorithm of choice
  - either when the data is nearly sorted (because it is adaptive), or
  - when the problem size is small (because it has low overhead)
- It is also stable
- Insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort

Insertion Sort

- Elementary sorting algorithm with $O(n^2)$ worst-case time; but
- Insertion sort is the algorithm of choice
  - either when the data is nearly sorted (because it is adaptive), or
  - when the problem size is small (because it has low overhead)
- It is also stable
- Insertion sort is often used as the recursive base case (when the problem size is small) for higher overhead divide-and-conquer sorting algorithms, such as merge sort or quick sort

Overview

Sorting Algorithms

Algorithm	Complexity			In-place?
	Worst-case	Average-case	Best-case	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
?	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
?		$\Theta(n \lg n)$		yes
?	$\Theta(n \lg n)$			yes

Divide and Conquer

Algorithm Design: Divide and Conquer

- One of the important algorithm design strategies
- [Unfortunately] named after Roman (then British!) political strategies
  - Divide your enemies (develop distrust)
  - Conquer them individually (easier)
  - Combine the states together(?)

Algorithm Design: Divide and Conquer

- One of the important algorithm design strategies
- [Unfortunately] named after Roman (then British!) political strategies
  - Divide your enemies (develop distrust)
  - Conquer them individually (easier)
  - Combine the states together(?)

Algorithm Design: Divide and Conquer

- One of the important algorithm design strategies
- [Unfortunately] named after Roman (then British!) political strategies
  - Divide your enemies (develop distrust)
  - Conquer them individually (easier)
  - Combine the states together(?)

Algorithm Design: Divide and Conquer

- One of the important algorithm design strategies
- [Unfortunately] named after Roman (then British!) political strategies
  - Divide your enemies (develop distrust)
  - Conquer them individually (easier)
  - Combine the states together(?)

Algorithm Design: Divide and Conquer

- One of the important algorithm design strategies
- [Unfortunately] named after Roman (then British!) political strategies
  - Divide your enemies (develop distrust)
  - Conquer them individually (easier)
  - Combine the states together(?)

Divide, Conquer and Combine!

Three major steps

- Split a problem into multiple sub-problems
- Solve individual sub-problems (independently)
- Combine solutions to individual sub-problems

- Huge number computational problems can be solved efficiently using divide-and-conquer
- First avenue of search for an efficient algorithm, once a brute-force solution is understood
- Divide-and-conquer algorithms are typically recursive — since the conquer part involves invoking the same technique on a smaller sub-problem
- Analysis of the running times of recursive programs is ~tricky
- When merging takes less time than solving the two subproblems, we get an efficient algorithm!

Divide, Conquer and Combine!

Three major steps

- Split a problem into multiple sub-problems
- Solve individual sub-problems (independently)
- Combine solutions to individual sub-problems

- Huge number computational problems can be solved efficiently using divide-and-conquer
- First avenue of search for an efficient algorithm, once a brute-force solution is understood
- Divide-and-conquer algorithms are typically recursive — since the conquer part involves invoking the same technique on a smaller sub-problem
- Analysis of the running times of recursive programs is ~tricky
- When merging takes less time than solving the two subproblems, we get an efficient algorithm!

Divide, Conquer and Combine!

Three major steps

- Split a problem into multiple sub-problems
- Solve individual sub-problems (independently)
- Combine solutions to individual sub-problems

- Huge number computational problems can be solved efficiently using divide-and-conquer
- First avenue of search for an efficient algorithm, once a brute-force solution is understood
- Divide-and-conquer algorithms are typically recursive — since the conquer part involves invoking the same technique on a smaller sub-problem
- Analysis of the running times of recursive programs is ~tricky
- When merging takes less time than solving the two subproblems, we get an efficient algorithm!

Divide, Conquer and Combine!

Three major steps

- Split a problem into multiple sub-problems
- Solve individual sub-problems (independently)
- Combine solutions to individual sub-problems

- Huge number computational problems can be solved efficiently using divide-and-conquer
- First avenue of search for an efficient algorithm, once a brute-force solution is understood
- Divide-and-conquer algorithms are typically recursive — since the conquer part involves invoking the same technique on a smaller sub-problem
- Analysis of the running times of recursive programs is ~tricky
- When merging takes less time than solving the two subproblems, we get an efficient algorithm!

Divide, Conquer and Combine!

Three major steps

- Split a problem into multiple sub-problems
- Solve individual sub-problems (independently)
- Combine solutions to individual sub-problems

- Huge number computational problems can be solved efficiently using divide-and-conquer
- First avenue of search for an efficient algorithm, once a brute-force solution is understood
- Divide-and-conquer algorithms are typically recursive — since the conquer part involves invoking the same technique on a smaller sub-problem
- Analysis of the running times of recursive programs is ~tricky
- When merging takes less time than solving the two subproblems, we get an efficient algorithm!

Divide, Conquer and Combine!

Three major steps

- Split a problem into multiple sub-problems
- Solve individual sub-problems (independently)
- Combine solutions to individual sub-problems

- Huge number computational problems can be solved efficiently using divide-and-conquer
- First avenue of search for an efficient algorithm, once a brute-force solution is understood
- Divide-and-conquer algorithms are typically recursive — since the conquer part involves invoking the same technique on a smaller sub-problem
- Analysis of the running times of recursive programs is ~tricky
- When merging takes less time than solving the two subproblems, we get an efficient algorithm!

Divide, Conquer and Combine!

Three major steps

- Split a problem into multiple sub-problems
- Solve individual sub-problems (independently)
- Combine solutions to individual sub-problems

- Huge number computational problems can be solved efficiently using divide-and-conquer
- First avenue of search for an efficient algorithm, once a brute-force solution is understood
- Divide-and-conquer algorithms are typically recursive — since the conquer part involves invoking the same technique on a smaller sub-problem
- Analysis of the running times of recursive programs is ~tricky
- When merging takes less time than solving the two subproblems, we get an efficient algorithm!

Divide, Conquer and Combine!

Three major steps

- Split a problem into multiple sub-problems
- Solve individual sub-problems (independently)
- Combine solutions to individual sub-problems

- Huge number computational problems can be solved efficiently using divide-and-conquer
- First avenue of search for an efficient algorithm, once a brute-force solution is understood
- Divide-and-conquer algorithms are typically recursive — since the conquer part involves invoking the same technique on a smaller sub-problem
- Analysis of the running times of recursive programs is ~tricky
- When merging takes less time than solving the two subproblems, we get an efficient algorithm!

Divide and Conquer: Advantages

Advantages

- Efficiency
- Parallelism
- Memory access patterns

Advantages

- Efficiency
- Parallelism
- Memory access patterns

Advantages

- Efficiency
- Parallelism
- Memory access patterns

Merge Sort

Merge Sort

- Invented by John von Neumann in 1945
- Classic divide-and-conquer problem
- Also useful for on-line sorting (as data come in)

Merge Sort

- Invented by John von Neumann in 1945
- Classic divide-and-conquer problem
- Also useful for on-line sorting (as data come in)

Merge Sort

- Invented by John von Neumann in 1945
- Classic divide-and-conquer problem
- Also useful for on-line sorting (as data come in)

Merge Sort
Algorithm

```
def MergeSort(a):
    n=len(a)

    if(len(a)==1):
        return a
    else:
        return merge(MergeSort(a[:n//2]),MergeSort(a[n//
                                              2:]))
```

Where's all the work happening?

Merge Sort

merge()

```
def merge(a,b):
    c=[None]*(len(a)+len(b))
    i=j=k=0
    while (i<len(a) and j<len(b)):
        if a[i]<b[j]:
            c[k]=a[i]
            i+=1
        else:
            c[k]=b[j]
            j+=1
        k+=1
    if (i<len(a)):
        c[k:]=a[i:]
    else:
        c[k:]=b[j:]
    return c
```

Merge Sort

Time Complexity

$$T(n) = 1, \text{if } n = 1$$

- Time to `sort` the first half of the array
- Time to `sort` the second half of the array
- Time to `merge` the sorted arrays

Merge Sort
Time Complexity

$T(n) = 1$, if $n = 1$

Otherwise,
$T(n) = T\left\lfloor \frac{n}{2} \right\rfloor +$

- Time to `sort` the first half of the array
- Time to `sort` the second half of the array
- Time to `merge` the sorted arrays

Merge Sort
Time Complexity

$T(n) = 1$, if $n = 1$

Otherwise,
$T(n) = T\left\lfloor \frac{n}{2} \right\rfloor + T\left\lceil \frac{n}{2} \right\rceil +$

- Time to `sort` the first half of the array
- Time to `sort` the second half of the array
- Time to `merge` the sorted arrays

Merge Sort
Time Complexity

$T(n) = 1$, if $n = 1$

Otherwise,
$T(n) = T\left\lfloor \frac{n}{2} \right\rfloor + T\left\lceil \frac{n}{2} \right\rceil + n$

- Time to `sort` the first half of the array
- Time to `sort` the second half of the array
- Time to `merge` the sorted arrays

Merge Sort

Recurrence Equation

$$T(1) = 1$$
$$T(2) = T(1) + T(1) + 2 = 4$$
$$T(3) = T(1) + T(2) + 3 = 8$$
$$T(4) = T(2) + T(2) + 4 = 12$$
$$\ldots$$
$$T(8) = T(4) + T(4) + 8 = 32$$
$$\ldots$$
$$T(16) = T(8) + T(8) + 16 = 80$$
$$\ldots$$
$$T(32) = T(16) + T(16) + 32 = 192$$

Merge Sort

Recurrence Equation

$$T(1) = 1$$
$$T(2) = T(1) + T(1) + 2 = 4$$
$$T(3) = T(1) + T(2) + 3 = 8$$
$$T(4) = T(2) + T(2) + 4 = 12$$
$$\ldots$$
$$T(8) = T(4) + T(4) + 8 = 32$$
$$\ldots$$
$$T(16) = T(8) + T(8) + 16 = 80$$
$$\ldots$$
$$T(32) = T(16) + T(16) + 32 = 192$$

Can you see a pattern?

Merge Sort

Recurrence Equation

$$T(1) = 1$$
$$T(2) = T(1) + T(1) + 2 = 4$$
$$T(3) = T(1) + T(2) + 3 = 8$$
$$T(4) = T(2) + T(2) + 4 = 12$$
$$\ldots$$
$$T(8) = T(4) + T(4) + 8 = 32$$
$$\ldots$$
$$T(16) = T(8) + T(8) + 16 = 80$$
$$\ldots$$
$$T(32) = T(16) + T(16) + 32 = 192$$

Can you see a pattern?

$$T(n)/n = lg\, n + 1$$

Merge Sort

Recurrence Equation

$$
\begin{aligned}
&T(1) = 1 \\
&T(2) = T(1) + T(1) + 2 = 4 \\
&T(3) = T(1) + T(2) + 3 = 8 \\
&T(4) = T(2) + T(2) + 4 = 12 \\
&\ldots \\
&T(8) = T(4) + T(4) + 8 = 32 \\
&\ldots \\
&T(16) = T(8) + T(8) + 16 = 80 \\
&\ldots \\
&T(32) = T(16) + T(16) + 32 = 192
\end{aligned}
$$

Can you see a pattern?

$$T(n)/n = \lg n + 1 \Rightarrow T(n) \in \Theta(n \lg n)$$

Quick Sort

Quick Sort

- Merge sort is very good, but does not operate in place!
- Quick sort is another divide-and-conquer algorithm
- Jon Bentley's "*Three Beautiful Quicksorts*" : https://www.youtube.com/watch?v=QvgYAQzg1z8

Quick Sort

- Merge sort is very good, but does not operate in place!
- Quick sort is another divide-and-conquer algorithm
- Jon Bentley's "*Three Beautiful Quicksorts*" : https://www.youtube.com/watch?v=QvgYAQzg1z8

Quick Sort

- Merge sort is very good, but does not operate in place!
- Quick sort is another divide-and-conquer algorithm
- Jon Bentley's "*Three Beautiful Quicksorts*" :
  https://www.youtube.com/watch?v=QvgYAQzg1z8

Quick Sort

- Central idea: Partition the array
- About a pivot (choice is critical!)
- All elements less than the pivot go into the *left subarray*
- All elements greater than the pivot go into the *right subarray*
- Pivot fits in between — in its place in the sorted array!
- Quicksort either half and we have the answer ...
- What is the `combine()` procedure here?

Quick Sort

- Central idea: Partition the array
- About a pivot (choice is critical!)
- All elements less than the pivot go into the *left subarray*
- All elements greater than the pivot go into the *right subarray*
- Pivot fits in between — in its place in the sorted array!
- Quicksort either half and we have the answer …
- What is the `combine()` procedure here?

Quick Sort

- Central idea: Partition the array
- About a pivot (choice is critical!)
- All elements less than the pivot go into the *left subarray*
- All elements greater than the pivot go into the *right subarray*
- Pivot fits in between — in its place in the sorted array!
- Quicksort either half and we have the answer ...
- What is the `combine()` procedure here?

Quick Sort

- Central idea: Partition the array
- About a pivot (choice is critical!)
- All elements less than the pivot go into the *left subarray*
- All elements greater than the pivot go into the *right subarray*
- Pivot fits in between — in its place in the sorted array!
- Quicksort either half and we have the answer ...
- What is the `combine()` procedure here?

Quick Sort

- Central idea: Partition the array
- About a pivot (choice is critical!)
- All elements less than the pivot go into the *left subarray*
- All elements greater than the pivot go into the *right subarray*
- Pivot fits in between — in its place in the sorted array!
- Quicksort either half and we have the answer …
- What is the `combine()` procedure here?

Quick Sort

- Central idea: Partition the array
- About a pivot (choice is critical!)
- All elements less than the pivot go into the *left subarray*
- All elements greater than the pivot go into the *right subarray*
- Pivot fits in between — in its place in the sorted array!
- Quicksort either half and we have the answer ...
- What is the `combine()` procedure here?

Quick Sort

- Central idea: Partition the array
- About a pivot (choice is critical!)
- All elements less than the pivot go into the *left subarray*
- All elements greater than the pivot go into the *right subarray*
- Pivot fits in between — in its place in the sorted array!
- Quicksort either half and we have the answer ...
- What is the `combine()` procedure here?

Sorting Algorithms

Algorithm	Complexity			In-place?
	Worst-case	Average-case	Best-case	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
Quick Sort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	yes
Heap Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	yes

Sorting Algorithms

Algorithm	Complexity			In-place?
	Worst-case	Average-case	Best-case	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
Quick Sort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	yes
Heap Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	yes

Self-assessment Exercise: Read about heaps and heap-sort ...

Sorting Algorithms

Algorithm	Complexity			In-place?
	Worst-case	Average-case	Best-case	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	yes
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	yes
Merge Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	no
Quick Sort	$\Theta(n^2)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	yes
Heap Sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$	yes

Self-assessment Exercise: Read about heaps and heap-sort ...

Many hybrid algorithms exist, e.g. quick + heap etc.

Why is Quick Sort so popular?

- *"When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort!"* (Steven Skiena)
- How do we compare two $\Theta(n \lg n)$ algorithms?
- Since the difference between the two programs will be limited to a multiplicative constant factor, details of how you program each algorithm *will make a big difference*!
- Also, operations in the innermost loop are simpler
- Worst-case of quick sort is still $\Theta(n^2)$, but …
- If you shuffle the input array, "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."
- Instead, if you pick a pivot at random, "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time"
- Worst-case of quick sort can be conquered by a randomisation!

Why is Quick Sort so popular?

- "*When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort!*" (Steven Skiena)
- How do we compare two $\Theta(n \lg n)$ algorithms?
- Since the difference between the two programs will be limited to a multiplicative constant factor, details of how you program each algorithm *will make a big difference*!
- Also, operations in the innermost loop are simpler
- Worst-case of quick sort is still $\Theta(n^2)$, but ...
- If you shuffle the input array, "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."
- Instead, if you pick a pivot at random, "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time"
- Worst-case of quick sort can be conquered by a randomisation!

Why is Quick Sort so popular?

- "*When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort!*" (Steven Skiena)
- How do we compare two $\Theta(n \lg n)$ algorithms?
- Since the difference between the two programs will be limited to a multiplicative constant factor, details of how you program each algorithm *will make a big difference*!
- Also, operations in the innermost loop are simpler
- Worst-case of quick sort is still $\Theta(n^2)$, but ...
- If you shuffle the input array, "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."
- Instead, if you pick a pivot at random, "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time"
- Worst-case of quick sort can be conquered by a randomisation!

Why is Quick Sort so popular?

- "*When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort!*" (Steven Skiena)
- How do we compare two $\Theta(n \lg n)$ algorithms?
- Since the difference between the two programs will be limited to a multiplicative constant factor, details of how you program each algorithm *will make a big difference*!
- Also, operations in the innermost loop are simpler
- Worst-case of quick sort is still $\Theta(n^2)$, but ...
- If you shuffle the input array, "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."
- Instead, if you pick a pivot at random, "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time"
- Worst-case of quick sort can be conquered by a randomisation!

Why is Quick Sort so popular?

- "*When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort!*" (Steven Skiena)
- How do we compare two $\Theta(n \lg n)$ algorithms?
- Since the difference between the two programs will be limited to a multiplicative constant factor, details of how you program each algorithm *will make a big difference*!
- Also, operations in the innermost loop are simpler
- Worst-case of quick sort is still $\Theta(n^2)$, but ...
- If you shuffle the input array, "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."
- Instead, if you pick a pivot at random, "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time"
- Worst-case of quick sort can be conquered by a randomisation!

Why is Quick Sort so popular?

- "*When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort!*" (Steven Skiena)
- How do we compare two $\Theta(n\, lg\, n)$ algorithms?
- Since the difference between the two programs will be limited to a multiplicative constant factor, details of how you program each algorithm *will make a big difference*!
- Also, operations in the innermost loop are simpler
- Worst-case of quick sort is still $\Theta(n^2)$, but ...
- If you shuffle the input array, "If you give me random input data, quicksort runs in expected $\Theta(n\, lg\, n)$ time."
- Instead, if you pick a pivot at random, "With high probability, randomized quicksort runs in $\Theta(n\, lg\, n)$ time"
- Worst-case of quick sort can be conquered by a randomisation!

Why is Quick Sort so popular?

- "*When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort!*" (Steven Skiena)
- How do we compare two $\Theta(n \lg n)$ algorithms?
- Since the difference between the two programs will be limited to a multiplicative constant factor, details of how you program each algorithm *will make a big difference*!
- Also, operations in the innermost loop are simpler
- Worst-case of quick sort is still $\Theta(n^2)$, but ...
- If you shuffle the input array, "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."
- Instead, if you pick a pivot at random, "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time"
- Worst-case of quick sort can be conquered by a randomisation!

Why is Quick Sort so popular?

- "*When Quicksort is implemented well, it is typically 2-3 times faster than mergesort or heapsort!*" (Steven Skiena)
- How do we compare two $\Theta(n \lg n)$ algorithms?
- Since the difference between the two programs will be limited to a multiplicative constant factor, details of how you program each algorithm *will make a big difference*!
- Also, operations in the innermost loop are simpler
- Worst-case of quick sort is still $\Theta(n^2)$, but ...
- If you shuffle the input array, "If you give me random input data, quicksort runs in expected $\Theta(n \lg n)$ time."
- Instead, if you pick a pivot at random, "With high probability, randomized quicksort runs in $\Theta(n \lg n)$ time"
- Worst-case of quick sort can be conquered by a randomisation!

Quick Sort

Importance of Randomisation (Steven Skiena)

- Since the time bound how does not depend upon your input distribution, this means that unless we are extremely unlucky (as opposed to ill-prepared) we will certainly get good performance
- Randomisation is a general tool to improve algorithms with bad worst-case but good average-case complexity
- The worst-case is still there, but we almost certainly won't see it

Quick Sort

Importance of Randomisation (Steven Skiena)

- Since the time bound how does not depend upon your input distribution, this means that unless we are extremely unlucky (as opposed to ill-prepared) we will certainly get good performance
- Randomisation is a general tool to improve algorithms with bad worst-case but good average-case complexity
- The worst-case is still there, but we almost certainly won't see it

Quick Sort

Importance of Randomisation (Steven Skiena)

- Since the time bound how does not depend upon your input distribution, this means that unless we are extremely unlucky (as opposed to ill-prepared) we will certainly get good performance
- Randomisation is a general tool to improve algorithms with bad worst-case but good average-case complexity
- The worst-case is still there, but we almost certainly won't see it

Other Sorts

Other Sorts

Can we beat $\Omega(n \lg n)$?

- Bucket sort
- Radix sort

Other Sorts

Can we beat $\Omega(n\,lg\,n)$?

- Bucket sort
- Radix sort

Other Sorts

Can we beat $\Omega(n \, lg \, n)$?

- Bucket sort
- Radix sort